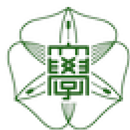

HOKKAIDO UNIVERSITY

Title	真偽維持システムを用いた項書換えシステムの推論手続きに関する研究
Author(s)	近藤, 久
Citation	北海道大学. 博士(工学) 甲第3397号
Issue Date	1994-03-25
DOI	10.11501/3076668
Doc URL	http://hdl.handle.net/2115/50015
Type	theses (doctoral)
File Information	000000272701.pdf

Instructions for use

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

真偽維持システムを用いた項書換えシステムの推論手続きに関する研究

近藤　久

真偽維持システムを用いた項書換えシステムの推論手続きに関する研究

近藤　　久

目次

1 序　論 **1**

1.1 背　景 1

1.2 項書換えシステム 3

1.3 真偽維持システム 6

1.4 本研究の目的 7

2 項書換えシステムと完備化手続き **10**

2.1 抽象書換えシステム 10

2.2 項書換えシステム 16

2.3 停止性検証 19

2.3.1 停止性検証の難しさ 19

2.3.2 停止性検証 22

2.4 完備化手続き 26

3 真偽維持システム **29**

3.1 問題解決器と TMS 29

3.2 探　索 30

3.3 ATMS 33

3.3.1 問題解決器と ATMS 33

3.3.2 基本定義 34

3.3.3 ラベル 35

3.3.4 基本データ構造 36

3.3.5 環境束 38

3.3.6 基本動作 40

3.3.7 基本アルゴリズム 41

4 停止性検証の効率化 44

4.1 停止性検証アルゴリズム 44

4.2 停止性検証における問題点 46

4.3 停止性検証システムのための TMS の設計 50

4.3.1 TMS ノードと正当化 50

4.3.2 正当化の計算 52

4.3.3 *nogood* の計算 52

4.3.4 TMS への登録 61

4.3.5 TMS への問い合わせ 63

4.4 TMS 利用停止性検証システムの動作 64

4.5 実験結果と TMS 利用の問題点 70

4.5.1 実験結果 70

4.5.2 TMS 利用の問題点 74

4.6 議　論 74

4.7 まとめ 77

5 完備化手続きの効率化 78

5.1 完備化手続きの問題点 79

5.2 並行完備化手続き 82

5.3 複数簡約順序完備化手続き 83

5.3.1 ノード 83

5.3.2 MKB の推論規則 85

5.4 基本アルゴリズム 89

5.5 複数簡約順序完備化手続きの動作 92

5.6 複数簡約順序完備化手続きの実験結果 93

5.7 議 論 100

5.8 まとめ 101

6 結 論 102

6.1 各章のまとめ 102

6.1.1 第 1 章 102

6.1.2 第 2 章 102

6.1.3 第 3 章 103

6.1.4 第 4 章 103

6.1.5 第 5 章 104

6.2 まとめ 104

謝 辞 106

参考文献　107

付録 A プログラムリスト　111

図目次

2.1 抽象書換えシステム $\mathcal{A}$ 11

2.2 合流性と弱合流性 13

2.3 簡約グラフ $\mathcal{A}_1$ 14

2.4 ARS の諸性質の関連 15

2.5 完備化アルゴリズム 28

3.1 ATMS と問題解決器 34

3.2 環境束 39

4.1 通常のバックトラック法による探索木 48

4.2 P 選択点における新 *halfgood* の計算 55

4.3 S 選択点における新 *halfgood* の計算 57

4.4 *nogood* の計算例 59

4.5 TMS 利用の場合の探索木 66

4.6 実験 1(x に関する微分) 72

4.7 実験 2(論理回路の結線情報とブール代数公理) 73

4.8 停止性検証法の包含関係 76

5.1 複数簡約順序完備化 (MKB) の推論規則 86

5.2 複数簡約順序完備化アルゴリズム 90

5.3 実行時間と簡約順序数 (表 5.1) 97

5.4 簡約順序数とノード数 (危険対数) 98

5.5 実行時間と簡約順序数 (表 5.2) 99

表目次

4.1 停止性検証システムの実験結果 71

5.1 複数簡約順序完備化の実験結果 1 95

5.2 複数簡約順序完備化手続きの実験結果 2 96

第 1 章

序　論

1.1　背　景

近年，計算機処理能力の向上と安価なハードウェアの出現は，ソフトウェアに対する要求の多用化や高度な知識情報処理への期待をもたらした．しかし，FORTRAN，COBOL，C，PASCAL などに代表される手続き型言語には，これらに応えるソフトウェアの開発に十分対処できない場合がある．手続き型言語によるプログラムは，簡単に言えば，データの取り出し，加工，格納指示とそれらの実行順序の記述の集まりであるが，実行時において，実行順序は格納されたデータ値に依存して決定され，実行順序の変化は格納するデータ値に影響を及ぼすというように両者が複雑に絡んで計算が進められる．したがって，プログラムの動きは静的にとらえにくく，大規模化の進むソフトウェアに対して全体的な正しさを保証することは，ほぼ不可能である．また，計算機を“いかに動かすか”という記述が中心の言語では，高度な処理対象が持つ論理的性質を十分表現できないこともある．

このような状況で，これまでと違う新しいパラダイムに基づくプログラミング言語が提

案されている．これらの言語に共通する特徴は，ベースとなる論理的体系が明確に定まっていることである．このことにより，演繹的手法によるプログラムの検証が可能となる．また，“何をするのか” という宣言的な記述によりプログラムが簡潔で読み易くなるなど，手続き型言語の欠点を補うことができる．

一方，人工知能，ソフトウェア工学においてはプログラムの変換，検証，合成は重要な問題として位置づけられ，その自動化は早急に望まれる技術である．

このような言語の計算モデル及びプログラムの変換，検証，合成といった技術の理論的基礎として，項書換えシステムがある [1, 2]．項書換えシステムは等式を左辺から右辺への向きの付いた書換え規則とみなし，項と呼ばれる式を書換えることによって計算を行う．その応用として，ソフトウェア工学では，先に述べたプログラムの変換 [3]，合成 [4]，検証 [5]，代数的仕様記述 [6] や関数型プログラミング [3] に，また，人工知能の分野では定理自動証明 [7] などに用いられている．

項書換えシステムの持つべき望ましい性質はいろいろなものがあるが，その中でも停止性 (*termination*) と合流性 (*confluence*) は特に重要である．書換え規則を項に適用する順序や場所によらず，停止性は無限に書換えが続かないこと，合流性は計算結果が一意であることを保証する．一般に，これらの性質の検証は決定不能であるがいくつかの十分条件が知られている [2, 8]．停止性と合流性をみたす項書換えシステムを完備であるという．与えられた等式仕様から完備な項書換えシステムを生成する推論手続きは完備化手続きと呼ばれ，プログラムの合成や定理自動証明などに応用されている [8]．従来，停止性検証 [9] 及び完備化 [10, 11, 12] を行う推論手続きがいくつか提案されているが，その実現手法では検証の際に推論の重複を多く生じるなど，非効率的である．

そのような推論の高速化技法として，人工知能の分野では真偽維持システムが提案され

ている [13, 14]．真偽維持システムは推論手続き (問題解決器) の行った推論をその根拠と共に保持することによって，その後の推論手続きを効率よく実行させるサブシステムである．ただし，推論手続きが真偽維持システムにどのような推論を与え，また，どのような問い合わせを行えばよいかを設計することは一般に難しく，問題領域ごとに考察する必要がある．

以下，本研究の対象となる項書換えシステムと完備化手続きおよび真偽維持システムについて簡単に紹介し，本研究の目的，本論文の構成について述べる．

1.2　項書換えシステム

自然数の加法，乗法，後者 (*succssor*) の等式による仕様が次のように与えられたと考える．

$$\begin{cases} x + 0 & \leftrightarrow & x & (e1) \\ x + s(y) & \leftrightarrow & s(x+y) & (e2) \\ x * 0 & \leftrightarrow & 0 & (e3) \\ x * s(y) & \leftrightarrow & x * y + x & (e3) \end{cases}$$

自然数は項 $0, s(0), s(s(0)), \cdots, s^n(0), \cdots$ によって表される．等式推論 (*equational reasoning*) によって $2 * 2$ は 4 を得る．

$$
\begin{aligned}
s(s(0)) * s(s(0)) &\leftrightarrow s(s(0)) * s(0) + s(s(0)) && (e4)\\
&\leftrightarrow (s(s(0)) * 0 + s(s(0))) + s(s(0)) && (e4)\\
&\leftrightarrow (0 + s(s(0))) + s(s(0)) && (e3)\\
&\leftrightarrow s(0 + s(0)) + s(s(0)) && (e2)\\
&\leftrightarrow s(s(0 + 0)) + s(s(0)) && (e2)\\
&\leftrightarrow s(s(0)) + s(s(0)) && (e1)\\
&\leftrightarrow s(s(s(0))) + s(0)) && (e2)\\
&\leftrightarrow s(s(s(s(0)))) + 0) && (e2)\\
&\leftrightarrow s(s(s(s(0)))) && (e1)
\end{aligned}
$$

この導出は等式 $(e1), \cdots, (e4)$ が左から右に用いられている．等式をこのように一方向に利用することは自然と項書換えシステムに導く．項書換えシステム (*term rewriting system* : TRS) は向き付けられた等式の集合であり，そのような等式を書換え規則と呼ぶ．向きを表すために$\to$を用いる．もし項 s が書換え規則 $l \to r$の左辺の項 lのある代入例を持つならば，s に $l \to r$を適用することにより，その代入例が右辺 rの対応する代入例と置換わり，項 t が得られる．このような書換えステップは $s \to t$ によって示される．書換えステップは有限あるいは無限の簡約列 $t_1 \to t_2 \to \cdots$ となるかもしれない．上記の導出は等式 $(e1), \cdots, (e4)$ を左から右に向き付けることによって得られたTRSによって項 $s(s(0)) * s(s(0))$ から項 $s(s(s(s(0))))$ への簡約列とみなすことができる．ただし，項 $s(s(0)) * s(s(0))$ から項 $s(s(s(s(0))))$ への唯一の簡約列ではないことに注意する．

TRS の持つべき望ましい重要な性質である停止性と合流性について述べる．上記の例では，すべての簡約列は有限になる．もし無限の簡約列が存在しないならば，その TRS は停

止性を持つという．

停止性を保証するためには，項の大小を比較する簡約順序と呼ばれる半順序 ($\succ$で表す) を導入し，$s \to t$ ならば必ず $s \succ t$ であることを確認すればよい．停止性を持つ TRS のすべての最大簡約列は必ず正規形で終わる．上記で用いた例ではすべての最大簡約列は同じ項 $s(s(s(s(0))))$ で終わる．与えられた項がたかだか 1 つ正規形を持つ性質を合流性あるいはチャーチ-ロッサー性という．この性質は分岐する簡約列 $t_1 \leftarrow \cdots \leftarrow t \to \cdots \to t_2$ が再び $t_1 \to \cdots \to t_3 \leftarrow \ldots \leftarrow t_2$のように合流することを示している．

もし合流性を持つ TRS においてある項が正規形を持つが，無限の書換えをも許す場合，どのようにして正規形を計算するかは重要な問題である．上記の例では，書換え規則 $x+s(y) \to s(x+y)$ を項 $s(s(0)) * s(0) + s(s(0))$ に適用するか，書換え規則 $x * s(y) \to x * y + x$ をその部分項 $s(s(0)) * s(0)$ に適用しても，この TRS は停止性を持つため問題はない．しかし，一般には正規形を計算する適切な戦略を採用することが重要である．適切な簡約戦略なしには，無限に書換えが継続し停止しないかもしれない．

Knuth と Bendix によって提案された完備化手続き [10] は与えられた等式集合と適切な簡約順序から上記で述べた 2 つの性質をみたす完備な TRS を生成する．例えば，次の群論の公理 $\mathcal{E}$と適当な簡約順序を完備化手続きに与えると完備な TRS:$\mathcal{R}$ を得る．

$$\mathcal{E} = \begin{cases} e \cdot x \leftrightarrow x \\ x^{-} \cdot x \leftrightarrow e \\ (x \cdot y) \cdot z \leftrightarrow x \cdot (y \cdot z) \end{cases}$$

$$\mathcal{R} = \begin{cases} e \cdot x \to x & e^- \to e \\ x^- \cdot x \to e & x^{--} \to x \\ (x \cdot y) \cdot z \to x \cdot (y \cdot z) & x \cdot x^- \to e \\ x^- \cdot (x \cdot y) \to y & x \cdot (x^- \cdot y) \to y \\ x \cdot e \to x & (x \cdot y)^- \to y^- \cdot x^- \end{cases}$$

$\mathcal{R}$ は停止性と合流性をみたすため，すべての項は $\mathcal{R}$ に関して一意な正規形を持つ．$\mathcal{E}$の等式理論は $\mathcal{R}$ による書換えで決定することができる．つまり，ある等式 $s \leftrightarrow t$ は $\mathcal{R}$ によって両辺が同じ正規形を持つときかつそのときに限って，$\mathcal{E}$から導出される (すなわち，$\mathcal{E}$をみたす，すべての解釈のもとで $s \leftrightarrow t$ は真となる)．例えば，すべての群は等式 $(x^- \cdot y)^- \leftrightarrow (y \cdot e)^- \cdot x$ を満足する．なぜなら両辺が正規形 $y^- \cdot x$ に書換えられるからである．

1.3　真偽維持システム

真偽維持システム (*truth maintenance system* : TMS) を利用することによって推論システム (*overall problem solver*) は “問題領域に関する推論を行う問題解決器 (*problem solver*)” と “問題解決器の推論を記録する部分 (TMS)” に明確に分けられる．つまり，TMS は問題解決器の推論を記録し，問題解決器の推論を効率よく行わせるためのサブシステムの役割を果たす．問題解決器は TMS に問い合わせを行うことによって，これまでに記録された推論から得られるいくつかの情報を受けとることができる．問題解決器は受けとった情報を用いて効率よく推論を行うことが可能になる．

真偽維持システムとして，Doyle の TMS[13] と de Kleer の ATMS(*assumption-based* TMS) [14, 15, 16] がよく知られている．TMS は正当化 (*justification*) に基づくシステムで

あるのに対し，ATMS は仮定 (*assumption*) に基づいている．

TMS(ATMS) は，

(1) 探索木の知的枝刈り (無駄なバックトラック，推論の再検知，矛盾の再検知の回避)

(2) 非単調推論

(3) (文脈ごとの) データベースの無矛盾性管理

の 3 つの基本機能を持つ．(1) はそれまでに行った推論を保持することによって，一度行った推論，失敗原因を後の問題解決器の推論から避ける機能である．(2) と (3) は非単調推論を行う際の不完全な事実とその管理を行う機能である．

TMS を利用した問題解決において重要なことは，TMS に解決しようとしている問題領域のどのような推論を与えるのか，また，どのような問い合わせを行えばよいのかということである．この考察を十分に行わなければ，TMS の利点を生かすことはできない．

1.4　本研究の目的

背景でも述べたように，項書換えシステムの応用は幅広く，その性質の検証は非常に重要である．本研究では，問題領域として項書換えシステムの停止性検証と完備化を対象とし，そこに含まれる計算効率上の問題点を明確にし，その問題点を効率よく解決することを目的とする．本論文は，そのために設計・開発した真偽維持システムとそれを用いた推論手続きについて述べたものである．

その成果は以下のように要約できる．

1. 従来の停止性検証手続きの問題点である，(1) 無駄なバックトラック，(2) 推論の再検知，及び (3) 矛盾の再検知を避けることにより，実行時間が非常に短縮された．
2. 複数の簡約順序を同時に扱うことによる推論の重複を避けることができ，完備化手続きの実行時間が非常に短縮された．

以下，本論文の構成について述べる．

第 2 章は本論の準備として，抽象書換えシステムを通して項書換えシステムの諸性質を述べている．また，項書換えシステムの停止性検証法と完備化手続きについて紹介している．

第 3 章では，真偽維持システムについて述べている．特に真偽維持システムとしてよく知られている ATMS について紹介している．

第 4 章では，項書換えシステムの停止性検証手続きの計算効率上の問題点を説明し，それを解決するために設計・開発した真偽維持システム及びそれを用いた検証手続きについて述べている．項書換えシステムの停止性を検証するためには，項を構成している関数記号の集合上に，ある制約をみたす半順序を決定しなければならない．この半順序は，従来の実現手法のようにバックトラック法を用いて決定することが可能であるが，その際，無駄なバックトラック，推論の再検知，矛盾の再検知という効率上の問題点を数多く生ずる．本章では，本手続きがいかにしてこれらの問題点を避けているのかを明らかにし，その有効性を論じている．

第 5 章では，完備化手続きの問題点を解決するために設計した真偽維持システムを用いた完備化手続きについて述べている．一般に，完備化手続きは等式の集合及び適切な簡約順序を入力とし，完備な項書換えシステムを出力する．特に本論文で対象としている手続きは，入力として簡約順序の候補を複数個受けとり，その中から適切なものを自動的に選択するものとしている．この手続きは，論理的には，入力として簡約順序を 1 つだけ受け

とる手続きを1つのプロセスとし，これらを並行に動作させたものと等価である．しかし，その場合には，各プロセス間で多くの推論の重複が生じ，無駄が多い．本完備化手続きでは，その問題のために真偽維持システムに基づく特別なデータ構造をノードとして持つことにより，それらの重複が生じないように推論を管理して，この問題を解決し効率を改善している．

第6章では，本研究の結論及び今後の展望について述べている．

第 2 章

項書換えシステムと完備化手続き

記号列を書換えて，意味がよりはっきりと把握できる記号列を導出するという操作は，計算系を特徴付ける最も基本的な操作である．$1+1 \to 2$ というのは，我々が最初に習う計算であり，$1+1$ という記号列から 2 という記号列への書換えである．

本章では，抽象書換えシステムを通して，書換えシステムの一般的諸性質について述べ，応用分野が広く，かつ深く性質が調べられている項書換えシステムについて述べる．項書換えシステムに関連して，項書換えシステムの停止性を検証する方法と与えられた等式集合から停止性及び合流性をみたす完備な項書換えシステムを生成する完備化手続きについて紹介する．

2.1 抽象書換えシステム

定義 2.1.1 (抽象書換えシステム) 抽象書換えシステム (*abstract reduction system* : ARS) は任意の集合 A と A 上の二項関係$\to$からなる対 $\mathcal{A} = \langle A, \to \rangle$ である．$(a, b) \in \to$ の代りに

$a \to b$ と書く．

例えば，$A = \{a_1, a_2, a_3\}$，$\to = \{a_1 \to a_2, a_1 \to a_3, a_2 \to a_3\}$ とすると $\mathcal{A} = \langle A, \to \rangle$ は1つのARSを定義する．$\mathcal{A}$ を簡約グラフで表すと図2.1のようになる．このような簡約グラフは一般には非連結グラフになる．すべての要素が簡約グラフの節となるように簡約グラフを描くことによってARSを表現することができ，性質を理解しやすい．

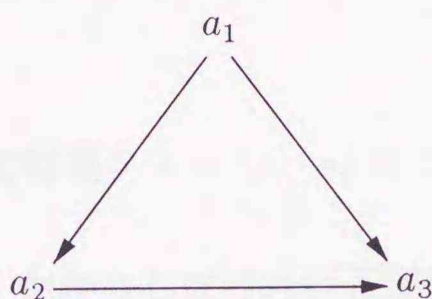


図 2.1: 抽象書換えシステム $\mathcal{A}$

定義 2.1.2 $\mathcal{A} = \langle A, \to \rangle$ をARSとする．

(1) $a, b \in A$ が同値であるとき $a \equiv b$ と表す．

(2) $\to$ の反射推移閉包を $\to^*$ $(\to \circ \cdots \circ \to)$，推移閉包を $\to^+$ $(\to \circ \to^*)$，対称閉包を $\leftrightarrow$ $(\leftarrow \cup \to)$，反射閉包を $\to^{\equiv}$ $(\to \circ \equiv)$ で表す．$\leftrightarrow$ の反射推移閉包を $\leftrightarrow^*$ $(\leftrightarrow \circ \cdots \circ \leftrightarrow)$ と表す ($\circ$ は関係の合成)．

(3) $\exists a, b, c \in A$ に対して，$a \to^* c \leftarrow^* b$ ならば $a \downarrow b$ と表し，c を a，b の共通簡約子 (*common reduct*) という．

$a \to^+ b$ とは $a_0 (\equiv a) \to a_1 \to \cdots \to a_n (\equiv b)$ となる $n \geq 1$ が存在する a と b の二項関係，$a \to^* b$ とは $a_0 (\equiv a) \to a_1 \to \cdots \to a_n (\equiv b)$ となる $n \geq 0$ が存在する a と b の二項関係を表している．

任意の $a, b \in A$ が $a \rightarrow^* b$ なる関係にあるとき，少なくとも1つ

$$a_0(\equiv a) \rightarrow a_1 \cdots \rightarrow a_n(\equiv b),\ n \geq 0 \tag{2.1}$$

なる書換え列が存在する．式2.1を a から b の簡約経路といい，n，つまり→の個数を簡約経路の長さという．

以下，ARS の諸性質を定義する．ARS:$\mathcal{A} = \langle A, \rightarrow \rangle$ がある性質を持つとき，関係→がその性質を持つともいう．

定義 2.1.3 (抽象書換えシステムの性質) $\mathcal{A} = \langle A, \rightarrow \rangle$ を ARS とする．

(1) $a \in A$ は $a \rightarrow b$ となる $b \in A$ が存在しなければ**正規形** (*normal form*) である．$a \in A$ は，ある正規形 b に対し $a \rightarrow^* b$ ならば正規形を持つという．$\mathcal{A}$ の正規形の集合を $NF(\mathcal{A})$ または文脈から $\mathcal{A}$ が明らかなとき $NF(\rightarrow)$ と表す．また，a の正規形を $a\downarrow$ と表す．

(2) $\mathcal{A}$ は，もしすべての $a \in A$ が正規形を持つとき**弱正規化性** (*weakly normalizing*(WN)) を持つという．

(3) $\mathcal{A}$ は，もし A の要素の $a_1 \rightarrow a_2 \rightarrow \cdots$ なる無限の簡約経路が存在しなければ**強正規化性** (*strongly normalizing*(SN)) を持つという．

(4) $\mathcal{A}$ は以下の性質を持つとき**UN性** (*unique normal forms*) を持つという．

$$\forall a, b \in A \text{ if } a \leftrightarrow^* b \text{ and } a, b \in NF(\mathcal{A}) \text{ then } a \equiv b$$

(5) $\mathcal{A}$ は以下の性質を持つとき**UN$^{\rightarrow}$性** (*unique normal forms with respect to reduction*) を持つという．

$$\forall a, b, c \in A \text{ if } a \rightarrow^* b, a \rightarrow^* c \text{ and } b, c \in NF(\mathcal{A}) \text{ then } b \equiv c$$

(6) $\mathcal{A}$ は以下の性質を持つとき **NF 性** (*normal forms property*) を持つという．

$$\forall a, b \in A \text{ if } a \leftrightarrow^* b \text{ and } b \in NF(\mathcal{A}) \text{ then } a \rightarrow^* b$$

(3) の SN 性の代りに**停止性** (*termiation*)，*Noetherian* ということもある．この性質は，どのような書換えを行っても必ず正規形になるという性質である．

(4) の UN 性はある 2 つの正規形 a，b が $a \leftrightarrow^* b$ なる関係にあるとき a と b は同値であるという性質である．

(5) の UN$^{\rightarrow}$ は任意の a に対して，a が正規形をもてば必ず唯一であるという性質である．

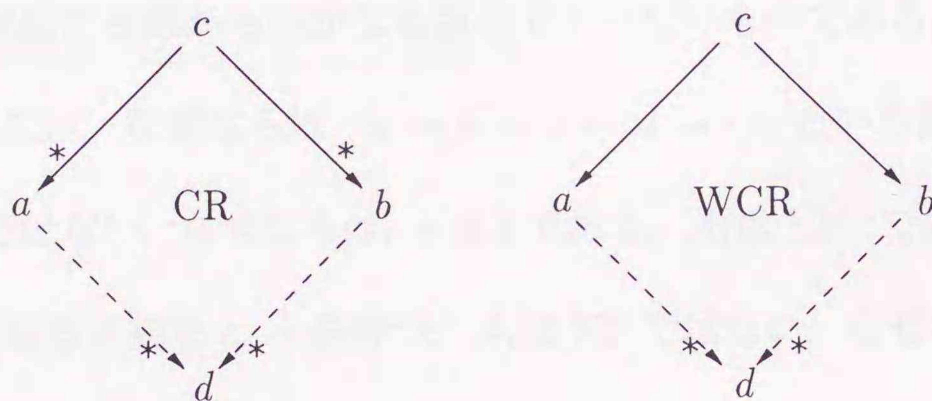


図 2.2: 合流性と弱合流性

定義 2.1.4 (合流性) $\mathcal{A} = \langle A, \rightarrow \rangle$ を ARS とする．

(1) $\mathcal{A}$ は，もし A 中の $a \leftarrow^* c \rightarrow^* b$ なるすべての要素 a, b, c に対して，$a \downarrow b$ ならば**合流性** (*confluence*) あるいは**チャーチ-ロッサー性** (*Church-Rossser property*(CR)) を持つという．

(2) $\mathcal{A}$ は，もし A 中の $a \leftarrow c \rightarrow b$ なるすべての要素 a, b, c に対して，$a \downarrow b$ ならば**局所合流性** (*localy confluence*) あるいは**弱チャーチ-ロッサー性** (*weakly Church-Rosser property*(WCR)) を持つという．

局所合流性と合流性を図 2.2に示す．ここで，2 つの図の違いに注意する．局所合流性では c が 1 回書換えられて a,b になり必ず $a \downarrow b$ で合流するが，合流性では c は何回も書換えられて a，b になっても必ず $a \downarrow b$ で合流することを示している．

例題 2.1.1 $\mathcal{A}_1 = \langle A_1, \rightarrow \rangle$ とする．ここで，$A_1 = \{a, b, c, d, e\}$，$\rightarrow = \{a \rightarrow b, b \rightarrow c, b \rightarrow d, d \rightarrow d, d \rightarrow e\}$ である．図 2.3は $\mathcal{A}_1$の簡約グラフである．

c，e は正規形である．正規形であるか否かは簡約グラフをみれば明らかである．正規形とは，簡約グラフで対応する節からいかなる弧もでていないものである．$\mathcal{A}_1$は WN である．しかし，SN ではない．なぜならば，$a \rightarrow b \rightarrow d \rightarrow d \rightarrow \cdots$ という無限の簡約経路が存在する．$\mathcal{A}_1$は UN ではない．なぜならば，$c \not\equiv e$ である．$\mathcal{A}_1$は UN$^{\rightarrow}$ではない，なぜならば，a と b は 2 つの異なる正規形 c，e を持つ．$\mathcal{A}_1$は NF ではない．なぜならば，e は正規形であり，$c \leftrightarrow^* e$ であるが，$c \rightarrow^* e$ とはならない．

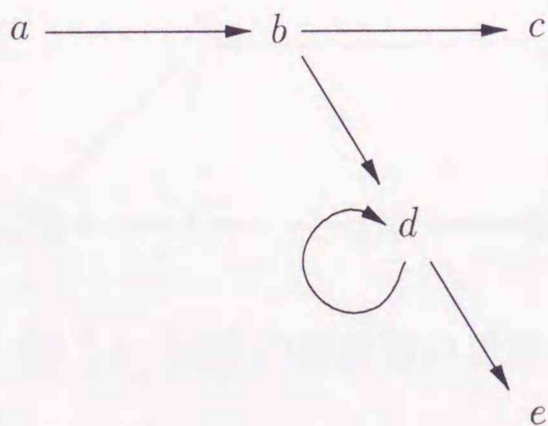


図 2.3: 簡約グラフ $\mathcal{A}_1$

これまでに示した ARS の諸性質がどのように関連しているのかを示すのが次の補題である.

補題 2.1.1 任意の ARS に対して

(1) SN 性を持つならば WN 性を持つ.

(2) CR 性を持つならば WCR 性を持つ.

(3) CR 性を持つならば NF 性を持つ.

(4) NF 性を持つならば UN 性を持つ.

(5) UN 性を持つならば $\mathrm{UN}^{\rightarrow}$ 性を持つ.

(6) WN かつ $\mathrm{UN}^{\rightarrow}$ 性を持つならば CR 性を持つ.

(7) SN かつ WCR 性を持つならば CR 性を持つ (Newman の補題).

補題 2.1.1を図 2.4に示す.

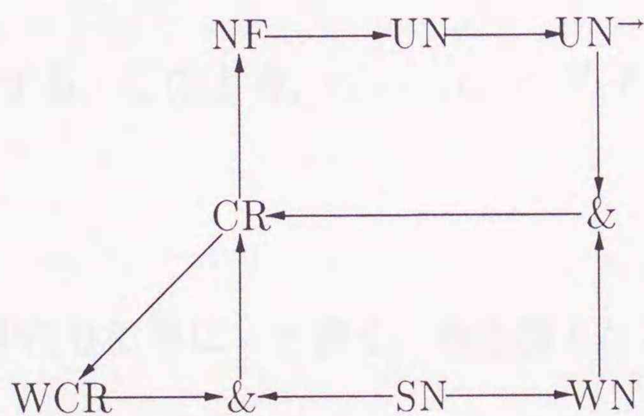


図 2.4: ARS の諸性質の関連

これまで述べた諸性質のうち，特に重要なのは CR 性と SN 性である．CR 性は書換え系を一種のプログラムとみなしたとき，そのプログラムの解の一意性を保証する．SN 性

は書換えによる計算が必ず停止することを保証する．CR かつ SN である書換え系は完備 (*complete*) であるという．

2.2 項書換えシステム

書換え計算系の中でも，応用範囲が広くかつその性質が深く研究されているものに項書換えシステムがある．項書換えシステムは関数型プログラミング，定理自動証明，代数的仕様記述と深くかかわっている．

定義 2.2.1 (関数記号) $\mathcal{F}$を関数記号 (*function symbol*) の集合とする．すべての $f \in \mathcal{F}$は**項数** (*arity*) と呼ばれる自然数を持つ．項数 0 の関数記号は**定数** (*constant*) である．

定義 2.2.2 (項) $\mathcal{T}(\mathcal{F},\mathcal{V})$ を関数記号の集合 $\mathcal{F}$と無限加算個の変数の集合 $\mathcal{V}$ からつくられる以下の条件をみたす項 (*term*) の集合とする．ここで，$\mathcal{F} \cap \mathcal{V} = \{\}$.

(1) $\mathcal{V} \subset \mathcal{T}(\mathcal{F},\mathcal{V})$

(2) $f \in \mathcal{F}$の項数を n とする．このとき，$t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{V})$ ならば $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{V})$

c が定数のとき $c()$ と表す代りに単に c と書く．ある項 $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ 中の変数の集合を $\mathcal{V}(t)$ と表す．変数を含まない項を**基礎項** (*ground term*) あるいは**閉じた項** (*closed term*) という．基礎項の集合 ($\mathcal{T}(\mathcal{F},\mathcal{V})$ の部分集合) を $\mathcal{T}(\mathcal{F})$ と表す．

定義 2.2.3 (書換え規則) $\mathcal{T}(\mathcal{F},\mathcal{V})$ 上の書換え規則は項 $l, r \in \mathcal{T}(\mathcal{F},\mathcal{V})$ の順序対 (l,r) であり，$l \rightarrow r$と表す．ただし，$\mathcal{V}(r) \subseteq \mathcal{V}(l)$.

定義 2.2.4 (項書換えシステム) **項書換えシステム** (*term rewriting system* : TRS) は関数記号の集合 $\mathcal{F}$ と項の集合 $\mathcal{T}(\mathcal{F},\mathcal{V})$ 上の書換え規則の集合 $\mathcal{R}$ の対 $\langle\mathcal{F},\mathcal{R}\rangle$ である．すべての書換え規則 $l \to r$ は以下の条件をみたすものとする．

(1) 左辺 l は変数ではない．

(2) 右辺 r で現れる変数は左辺 l にも現れる．

以後，TRS : $\langle\mathcal{F},\mathcal{R}\rangle$ を単に TRS : $\mathcal{R}$ と書く．

定義 2.2.5 (代入) 代入 θ は $\mathcal{V}$ から $\mathcal{T}(\mathcal{F},\mathcal{V})$ への写像である．θ は $\mathcal{T}(\mathcal{F},\mathcal{V})$ から $\mathcal{T}(\mathcal{F},\mathcal{V})$ へ拡張される．すなわち，$t \equiv f(t_1,\ldots,t_n)$ とすると $\theta(t) \equiv f(\theta(t_1),\ldots,\theta(t_n))$ である．$\theta(t)$ を t の**代入例** (*instance*) という．以後，$\theta(t)$ を $t\theta$ と表す．書換え規則の左辺の代入例を**可簡約項** (*redex*) という．

定義 2.2.6 (文脈) 特別な定数記号 $\Box$ を 1 つ含む項を**文脈** (*context*) といい，$c[\ ]$ で表す．$c[\ ]$ の $\Box$ を項 s で置換えて得られる項を $c[s]$ と表す．ここで，項 s を $c[s]$ の**部分項** (*subterm*) という．特に $c \not\equiv \Box$ のとき項 s は $c[s]$ の**真部分項** (*proper subterm*) である．

定義 2.2.7 (書換え関係) TRS : $\mathcal{R}$ は $\mathcal{T}(\mathcal{F},\mathcal{V})$ 上に**書換え関係** (*rewrite relation*) $\to_{\mathcal{R}}$ を定義する．つまり，もし，$l \to r$ が $\mathcal{R}$ 中の書換え規則であり，$s \equiv c[l\theta]$，$t \equiv c[r\theta]$ となるような文脈 $c[\ ]$ と代入が存在するとき $s \to_{\mathcal{R}} t$ と表し，項 s は項 t に書換えられるという．$s \to_{\mathcal{R}} t$ を**書換えステップ** (*rewrite step*) または**簡約ステップ** (*reduction step*) という．$\to_{\mathcal{R}}$ の推移的閉包を $\to^{+}_{\mathcal{R}}$，反射推移閉包を $\to^{*}_{\mathcal{R}}$ と表す．

定義 2.2.8 (正規形) 項 t はそれ以上書換えられなければ既約 (*irreducible*) であるという．もし $s \to^{*}_{\mathcal{R}} t$ かつ t が既約ならば t は s の**正規形**である．

例題 2.2.1 関数記号の集合を $\mathcal{F} = \{0, s, a, m\}$，変数の集合を $\mathcal{V} = \{x, y\}$ とし，以下の TRS : $\mathcal{R}$ を考える．なお，関数記号の項数はそれぞれ 0,1,2,2 である．

$$\mathcal{R} = \begin{cases} r_1 \;:\; a(0,x) & \to \; x \\ r_2 \;:\; a(s(x),y) & \to \; s(a(x,y)) \\ r_3 \;:\; m(0,x) & \to \; 0 \\ r_4 \;:\; m(s(x),y) & \to \; a(m(x,y),y) \end{cases}$$

$m(s(s(0)), s(s(0)))$ は，書換え規則 $r_1 \sim r_4$ を適用することにより以下の書換えステップの列を得る．

$$\begin{aligned} \underline{m(s(s(0)), s(s(0)))} &\to_{\mathcal{R}} a(\underline{m(s(0), s(s(0)))}, s(s(0))) && (r_4) \\ &\to_{\mathcal{R}} a(a(\underline{m(0, s(s(0)))}, s(s(0))), s(s(0))) && (r_4) \\ &\to_{\mathcal{R}} a(\underline{a(0, s(s(0)))}, s(s(0))) && (r_3) \\ &\to_{\mathcal{R}} \underline{a(s(s(0)), s(s(0)))} && (r_1) \\ &\to_{\mathcal{R}} s(\underline{a(s(0), s(s(0)))}) && (r_2) \\ &\to_{\mathcal{R}} s(s(\underline{a(0, s(s(0)))})) && (r_2) \\ &\to_{\mathcal{R}} s(s(s(s(0)))) && (r_1) \end{aligned}$$

ここで，下線を引いた項が可簡約項である．

定義 2.2.9 (TRS の停止性) 任意の TRS:$\mathcal{R}$ は $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ のような無限の簡約列が存在しなければ停止する．

定義 2.2.10 (TRS の合流性) 任意の TRS:$\mathcal{R}$ は任意の項 s, t に対し，$s \leftarrow^*_{\mathcal{R}} \circ \to^*_{\mathcal{R}} t$ ならば $s \to^*_{\mathcal{R}} \circ \leftarrow^*_{\mathcal{R}} t (s \downarrow_{\mathcal{R}} t)$ が成り立つとき $\mathcal{R}$ は合流性をみたすという．$s \leftarrow_{\mathcal{R}} \circ \to_{\mathcal{R}} t$ ならば $s \to^*_{\mathcal{R}} \circ \leftarrow^*_{\mathcal{R}} t$ が成り立つとき $\mathcal{R}$ は弱合流性をみたすという．$s \leftarrow_{\mathcal{R}} u \to_{\mathcal{R}} t$ の形をした 2 ステップの ($s \leftrightarrow_{\mathcal{R}} t$ であることの) 証明をピーク (*peak*) という．

停止性を示すことのできたTRSに関しては合流性は比較的容易に行うことができる．これは以下の補題に基づく．

補題 2.2.1 (Newman の補題) 停止性をみたすTRS:$\mathcal{R}$ が，弱合流性をみたすならば合流性もみたす．

節 **2.1**で ARS に対して定義したすべての性質は，TRS に対しても成立する．つまり，ARS : $\langle \mathcal{T}(\mathcal{F},\mathcal{V}), \rightarrow_{\mathcal{R}} \rangle$ をすべての TRS に対応させることにより明らかである．

2.3　停止性検証

本節では，TRS の停止性検証がなぜ困難であるかを例を用いて述べたあと，単純化順序を用いた停止性検証法について述べる．

2.3.1　停止性検証の難しさ

一般にプログラムの正当性 (*correctness*) を検証するためには，次の２つの性質を示さなければならない．

1. 停止性 (*termination*) : プログラムは必ず停止する．
2. 部分正当性 (*partial correctness*) : プログラムが停止したならば，得られた最終結果は正しいものである．

TRS では部分正当性を示すことはしばしば容易である [2]．これは TRS では得られた最終結果が望まれた結果でない場合，さらに書換えが続行することを容易に示すことができ

るためである．したがって，TRS の正当性を示すためには停止性を示すことが重要なポイントとなる．

例題 2.3.1 次の 3 本の書換え規則をからなる簡単な書換えシステムを考える．

$$\begin{cases} white, red \to red, white \\ blue, red \to red, bule \\ blue, white \to white, blue \end{cases}$$

この書換えシステムは “Dutch National Flag” ゲームを行なう．このゲームは赤，青，白のマーブル (碁石) を不特定に並べた列が与えられたとき，赤の碁石を左に，青の碁石を右に，白の碁石を中央に並べ換えるものである．

例えば，最初の書換え規則は，与えられた碁石列の任意の隣接する 2 つの碁石の左が白で右が赤ならば，赤を左に，白を右に置換えることができるということを示している．

与えられた初期列に関わらず，上記の書換え規則を任意の順序で適用した結果が常に正しく並べ換えられた碁石の列となることを示すことは難しくはない．

ある停止性の検証は青は白より大きく，白は赤より大きいという順序に基づいて行なわれる．各書換え規則は 2 つの碁石を並べ換える．つまり，“大きい” 色を持つ左にある碁石が “小さい” 色を持つ右の碁石と置換えられる．

例題 2.3.2 次の選言標準型への TRS を考える．この例は TRS が停止するかどうか，そしてなぜ停止するかを決定することの難しさを例示する．

$$\begin{cases} --\alpha \to \alpha \\ -(\alpha+\beta) \to -\alpha \times -\beta \\ -(\alpha\times\beta) \to -\alpha + -\beta \\ \alpha\times(\beta+\gamma) \to (\alpha\times\beta)+(\alpha\times\gamma) \\ (\beta+\gamma)\times\alpha \to (\beta\times\alpha)+(\gamma\times\alpha) \end{cases}$$

最初の書換え規則は二重否定を削除し，2，3 本目の書換え規則は否定を論理和と論理積に変換するためにド・モルガンの法則を適用し，最後の 2 本の書換え規則は分配則を適用する．

このシステムが停止することは明らかではない．つまり，ある書換えは項の長さを減じ，例えば，

$$(0\times(-1+-1)) \to \cdots \to -0+(1\times 1)$$

一方，他の書換えでは項の長さが増える．例えば，

$$-(0\times(1+1)) \to \cdots \to (((-0\times 0)+(-0\times -1))+((-1\times -0)+(-1\times -1)))$$

さらに，ある部分項への書換え規則の適用はその部分項の構造に影響を与えるばかりでなく，同様にその部分項を含む項の構造にも影響を与える．

停止性の検証は書換え規則と部分項の非決定的な選択によって作られる多くの異なる書換え列を考慮に入れなければならない．このため TRS の停止性検証は非常に難しい問題である．

2.3.2 停止性検証

TRS の停止性を検証する一般的な方法は存在しないがいくつかの十分条件が知られている [2, 8, 17]．本論文では，Dershowitz によって提案された単純化順序 [2] を用いた停止性検証法を用いる．

定義 2.3.1 (簡約順序) 簡約順序 (*reduction ordering*)$\succ$は置換え，代入に関して閉じた，項の集合$\mathcal{T}(\mathcal{F},\mathcal{V})$ 上の整礎な推移的かつ非反射的な半順序である (*well-founded pertial ordering*).

置換え，代入に関して閉じているとは，$s \succ t$ ならば任意の文脈 $c[\,]$ に対して，$c[s] \succ c[t]$, 任意の代入θに対して $s\theta \succ t\theta$である．$\succ$が整礎であるとは$\mathcal{T}(\mathcal{F},\mathcal{V})$ 中の項の無限の下降列 $t_1 \succ t_2 \succ \cdots$ が存在しないことである．このとき以下の定理により停止性が保証される．

定理 2.3.1 ある簡約順序$\succ$に対し，TRS : $\mathcal{R}$ のすべての書換え規則 $l \to r$ が $l \succ r$ならば $\mathcal{R}$ は停止する．

簡約順序として単純化順序がある．

定義 2.3.2 (単純化順序) 項の集合$\mathcal{T}(\mathcal{F})$ 上の半順序$\succ$(推移的かつ非反射的な二項関係) は，次の 3 つの性質をみたすならば単純化順序である．

(1) $s \succ t$ ならば $f(\ldots,s,\ldots) \succ f(\ldots,t,\ldots)$　置換

(2) $f(\ldots,t,\ldots) \succ t$　部分項

(3) $f(\ldots,t,\ldots) \succ f(\ldots,\ldots)$　削除

削除の性質は関数記号が固定個の引数をとる場合必要がない．

定理 2.3.2 (停止定理 [2, 8]) TRS:$\mathcal{R} = \{l_i \to r_i\}_{i=1}^{n}$ は，任意の代入θに対して，

$$l_i\theta \succ r_i\theta \quad i = 1, \ldots, n$$

となる単純化順序$\succ$が $\mathcal{T}(\mathcal{F}, \mathcal{V})$ 上に存在するならば停止する．

単純化順序として経路順序 (*path ordering*) がよく用いられる．経路順序は関数記号の集合上の半順序を項の集合上に構文的に拡張して定義される整礎な半順序である．経路順序として辞書式経路順序，再帰的経路順序，再帰的分割順序などがよく知られている [17].

本論文では，停止性検証に辞書式経路順序を用いるが，これを他の経路順序にしても本論文の手法は成り立つ．

定義 2.3.3 (辞書式経路順序 (*lexicographic path ordering*)) $\succ$を固定個の引数をとる関数記号の集合 $\mathcal{F}$上の半順序とする．このとき $\mathcal{F}$から構成された項の集合 $\mathcal{T}(\mathcal{F})$ 上の辞書式経路順序$\succ_{lpo}$は以下のように再帰的に定義される．項を $s \equiv f(s_1, \ldots, s_m)$，$t \equiv g(t_1, \ldots, t_n)$ とする．このとき，$s \succ_{lpo} t$ であることは以下のいずれかをみたすときに限る．

(1) $s_i \succeq_{lpo} t$ for *some* $i = 1, \ldots, m$

(2) $f \succ g$ かつ $s \succ_{lpo} t_j$ for *all* $j = 1, \ldots, n$

(3) $f = g$ かつ $(s_1, \ldots, s_m) \succ_{lpo}^{*} (t_1, \ldots, t_n)$ かつ $s \succ_{lpo} t_j$ for *all* $j = 1, \ldots, n$

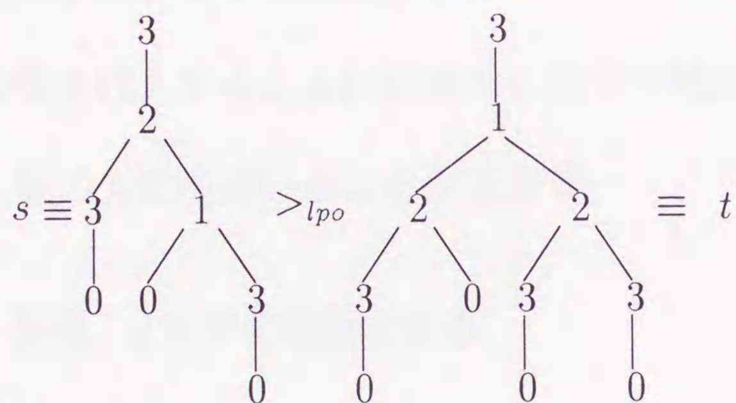
ここで$\succ_{lpo}^{*}$は$\succ_{lpo}$の辞書式順序への拡張である．

上記の定義の $\mathcal{F}$上の半順序$\succ$を先行順序 (*precedence*)，先行順序に属する順序対を先行順序対と呼ぶ．以後，先行順序対を (f, g) と表わす (直観的には $f \succ g$を表わす).

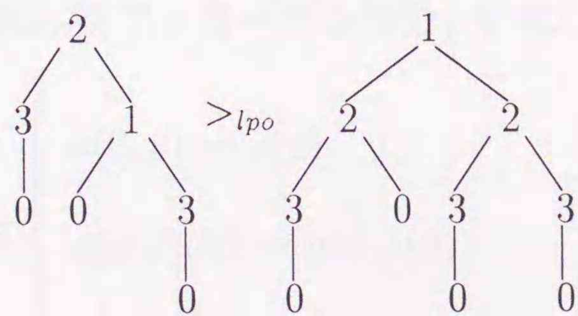
定理 2.3.3　$\mathcal{T}(\mathcal{F})$ 上の辞書式経路順序$\succ_{lpo}$は関数記号の集合$\mathcal{F}$中のすべての関数記号が固定個の項数を持つならば単純化順序である．

辞書式経路順序で項sが項tより大きいことを決定するため，はじめに 2 つの項の最も外側の関数記号を比較する．もし 2 つの関数記号が等しければ$(s_1,\ldots,s_m)$が$(t_1,\ldots,t_n)$より辞書式順に辞書式経路順序で大きく，さらにsが$t_i(i=1,\ldots,n)$よりも辞書式経路順序で大きくなければならない．もしsの最も外側の関数記号がtの最も外側の関数記号より大きければsはtの各部分項よりも辞書式経路順序で大きくなければならない．一方，sの最も外側の関数記号がtの最も外側の関数記号と等しくもなく，大きくもなければsのある部分項がtと等しいかまたは大きくなければならない．

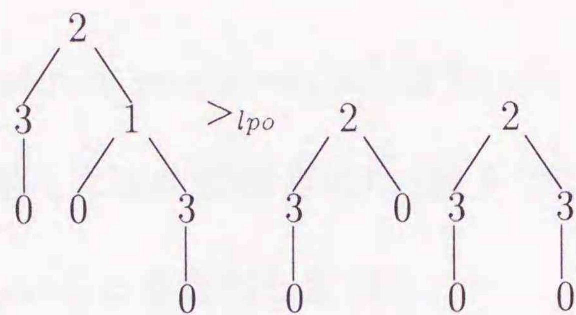
例えば，整数を関数記号とする項を木として表わす (項をそれぞれs，tとする)．辞書式



経路順序$>_{lpo}$の定義によって，最も外側の関数記号を比較する．したがって目標は，



に減ぜられる (実際には$(s_1) >^*_{lpo} (t_1)$)．ここで$2>1$より，さらに



に減ぜられる．以下同様の議論によって項 s が項 t より$>_{lpo}$によって大きいことがわかる．

定理 2.3.2と 2.3.3より以下の停止性検証法を得ることができる．

停止性検証法 (辞書式経路順序法) TRS:$\mathcal{R} = \{l_i \to r_i\}_{i=1}^{n}$は，任意の代入に対して，

$$l_i\theta \succ_{lpo} r_i\theta \quad i = 1, \ldots, n$$

となる先行順序$\succ$が $\mathcal{F}$上に存在するならば停止する．

ここで，変数へ任意の項を代入することは計算的には不可能である．しかし，次の命題を利用することにより上記の方法を用いることができる．

命題 2.3.1 $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ を項，$x \in \mathcal{V}$を変数とする．

(1) ある代入θに対して，$\neg(x\theta \succ_{lpo} t\theta)$

(2) x が t の真部分項であるとき，またそのときに限り，任意の代入θに対して，$t\theta \succ_{lpo} x\theta$

例題 2.3.3 例えば，次の TRS:$\mathcal{R}$(アッカーマン関数) を考える．

$$\mathcal{R}\begin{cases} a(0,y) \to s(y) \\ a(s(x),0) \to a(x,s(0)) \\ a(s(x),s(y)) \to a(x,a(s(x),y)) \end{cases}$$

$\mathcal{R}$ は辞書式経路順序法により先行順序 $\{(a,s)\}$ のもとで停止性を示すことができる．

2.4 完備化手続き

最初の完備化手続き (*completion procedure*)[10] は Knuth と Bendix によって普遍代数の語の問題 (*word problem*) を解くために提案された (以下では完備化手続きを単に KB と略記する). Huet は KB が正しいことを証明した [18].

完備化手続きについて述べる前に，準備を行う.

定義 2.4.1 (危険対 (*critical pair*)) $l \to r$, $s \to t$ を TRS:$\mathcal{R}$ 中の書換え規則とする. ある文脈 $c[\,]$ と非変数 u に対して $s \equiv c[u]$ であるとする. $u\theta \equiv l\theta$となる代入θ(但し，θは u と l の最汎単一化子) が存在するとき項 $s\theta \equiv c\theta[u\theta]$ は $t\theta$にも $c\theta[r\theta]$ にも書換えられる. この 2 つの項から作られる等式 $t\theta \leftrightarrow c\theta[r\theta]$ を危険対という. $\mathcal{R}$ 中の書換え規則から得られる危険対の集合を $cp(\mathcal{R})$ と表す.

補題 2.4.1 (危険対補題 [10, 19]) 任意のピーク $s \leftarrow_{\mathcal{R}} u \to_{\mathcal{R}} t$ に対して，書換え証明 $s \to^*_{\mathcal{R}} v \leftarrow^*_{\mathcal{R}} t$ または危険対証明 $s \equiv c[p\theta] \leftrightarrow c[q\theta] \equiv t$ が存在する. (但し，$p \leftrightarrow q \in cp(\mathcal{R})$ は危険対)

補題 2.2.1，2.4.1及び弱合流性の定義から，停止する TRS:$\mathcal{R}$ は，もしすべての危険対 $p \leftrightarrow q$が $p \to^*_{\mathcal{R}} \circ \leftrightarrow^*_{\mathcal{R}} q$をみたせば合流性をみたす. KB はこの事実に基づいている. KB のアルゴリズムを図 2.5 に示す.

このアルゴリズムが停止して成功ならば次のことが成立する.

1. 等式理論 Eと書換え規則の集合 Rは，Rを等式理論とみなしたとき同値である
2. Rは合流性，停止性をみたす

Backmair 等は KB を以下の抽象的な推論規則として定式化した．

Delete:	$(\mathcal{E} \cup \{s \leftrightarrow s\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R})$	
Compose:	$(\mathcal{E}; \mathcal{R} \cup \{s \to t\}) \vdash (\mathcal{E}; \mathcal{R} \cup \{s \to u\})$	if $t \to_{\mathcal{R}} u$
Simplify:	$(\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R}) \vdash (\mathcal{E} \cup \{s \leftrightarrow u\}; \mathcal{R})$	if $t \to_{\mathcal{R}} u$
Orient:	$(\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R} \cup \{s \to t\})$	if $s \succ t$
Collapse:	$(\mathcal{E}; \mathcal{R} \cup \{s \to t\}) \vdash (\mathcal{E} \cup \{u \leftrightarrow t\}; \mathcal{R})$	if $s \to_{\mathcal{R}} u$ by $l \to r \in \mathcal{R}$ with $s \rhd l$
Deduce:	$(\mathcal{E}; \mathcal{R}) \vdash (\mathcal{E} \cup \{s \leftrightarrow t\}; \mathcal{R})$	if $s \leftarrow_{\mathcal{R}} u \to_{\mathcal{R}} t$

$\succ$は簡約順序，$\mathcal{E}$は任意の等式集合，$\mathcal{R}$ はすべての書換え規則 $l \to r$に対し $l \succ r$が成り立つ任意の TRS，$\rhd$は包含順序 (*encompassment ordering*) である．$s \rhd t$ であるとは s の部分項が t の代入例であり，逆が成り立たないとき，かつそのときに限る．

Delete は自明な等式 $s \leftrightarrow s$ を $\mathcal{E}$から削除する．**Compose** は書換え規則 $s \to t$ の右辺 t を書換える．**Simplify** は等式 $s \leftrightarrow t$ の左辺，右辺どちらかを書換える．**Orient** は等式 $s \leftrightarrow t$ を簡約順序に従って書換え規則 $s \to t$ あるいは $t \to s$ に変換する．**Collapse** は書換え規則 $s \to t$ の左辺 s を書換え，等式 $u \leftrightarrow t$ とする．但し，s に適用しようとする書換え規則 $l \to r$は左辺 lが $s \rhd l$のものに限る．**Deduce** は新たな等式 $s \leftrightarrow t$ を $\mathcal{E}$に加える．但し，$s \leftarrow_{\mathcal{R}} u \to_{\mathcal{R}} t$.

KB は等式集合 $\mathcal{E}$，TRS:$\mathcal{R} = \{\}$ 及び簡約順序$\succ$を入力とし，上記の推論規則の適用を公平 (*fair*)[18] に行い，等式集合 $\mathcal{E}$が空になるならば完備な TRS:$\mathcal{R}$ を返す．KB は (1) 完備な TRS を生成し終了する，(2) ある等式を向き付けることができずに失敗する，(3) 発散する，のいずれかの結果を生じる．

```
1: function kb(E等式の集合) return 書換え規則の集合
2:    var R, R_new
3:    while E ≠ φ
4:       Eから等式 s ↔ t を選択する
5:       if s↓≡ t↓ then
6:          /*自明な等式を Eから削除する*/
7:          E := E\{s ↔ t}
8:       else
9:          /*等式を書換え規則に向き付ける*/
10:         case s↓≻ t↓ then R_new := {s↓→ t↓}
11:         case t↓≻ s↓ then R_new := {t↓→ s↓}
12:         otherwise fail を表示して停止する
13:         /*簡約可能な書換え規則を Rから取り除く*/
14:         while Rの中に R_newで書換え可能な書換え規則 l → rが存在
15:            R := R\{l → r}
16:            E := E ∪ {l ↔ r}
17:         end
18:         /*新しい書換え規則を Rへ，危険対を Eへ追加する*/
19:         R := R ∪ R_new
20:         E := E\{s ↔ t} ∪ {p ↔ q|p ↔ qは R_newと Rの間の危険対 }
21:      end
22:   end
23:   return R /*完備な Rを返す*/
24: end
```

図 2.5: 完備化アルゴリズム

第 3 章

真偽維持システム

本章では，真偽維持システム (*truth maintenance system*：TMS) [20] について述べる．TMS は問題解決器の推論をその根拠と共に保持し，その後の推論を効率よく行なうための機構である．

3.1　問題解決器と TMS

多くの問題解決器 (*problem solver*) は探索を行ない，さもなくば直接的なアルゴリズムによって問題は解かれる．問題解決器は常に等しくもっともらしい選択肢の中から選択を行なう必要性が生じる．これらは“次に進むために行なうゴールの選択”，“不完全なデータから導くためのもっともらしい推論の選択”，“次に探索する仮定世界の選択”など様々な選択がある．結局，これらの選択の多く，そしてほとんどすべてが間違いであることが示される．しかし，問題解決器は，はじめにこれらの等しく似通った選択肢に出会う．この観点において，問題解決器は各次元 (*dimension*) が問題解決器の出会う選択肢の集合に

よって定義された空間を探索するように見ることができる．

探索空間を探索する 1 つの方法は，生成検定法 (*generate and test*) である．この方法は単に探索空間の各要素を数え上げそれがゴールを満足するかどうかテストする方法である．この方法は非常に非効率的であるがもし解が 1 つあるならそれを見つけることを保証する．もし探索空間の各要素が似ていないならば問題を解くための一番良い方法である．しかし，多くの問題解決において探索空間の各要素の中には非常に多くの似たものがある．結果として効率は探索空間のある領域で得られた結果を他の領域へ持ち上げる (*lift*) ことによって得られる．この仕事は簡単なものではなく，問題解決器が注意深く設計されなければ，効率が向上するかわりに一貫性 (*coherency*) 及び徹底性 (*exhaustivity*) を犠牲にしてしまう．

これらの問題点を解決するために TMS が開発された．TMS は全体的な問題解決器 (*overall problem solver*) を “単に問題領域のルールに関する部分 (*problem solver*)” と “単に現在の探索の状態を記憶する部分 (つまり TMS)” に明確に分割する．また TMS は徹底性，一貫性を犠牲にすることなく問題解決器の効率を向上させることを可能にする．

3.2　探　索

本節ではバックトラック法の問題点について述べる．文献 [14] で定義されている制約言語 (*constraint language*) を用いる．

定義 3.2.1 (制約言語)

(1) 各変数は固定個の値の中の 1 つである．

(2) 制約は変数，整数，演算子 (=，$\neq$，+，−，×，!) を用いる．

(3) 論理和 (*inclusive or*) は$\vee$で表わす．

(4) *one of disjunction* は$\otimes$で表わされる．これは排他的論理和 (*exclusive or*) と同義であるが特にどちらか一方が成り立つことを強調するために用いる．

(4) は例えば，$(x+y=z)\otimes(z=3)$ のようにどちらか一方が成り立つ場合に用いる．特に，$x\in\{-1,0,1\}$ は $(x=-1)\otimes(x=0)\otimes(x=1)$ と同義である．

関数 $e_i(x)$ は負荷の大きい計算を必要とする ($e_i(x)=(x+100000)!$)．問題のゴールは制約を満足する変数への値の割り当てをすべて求めることである．値は常に有理数であり，通常は整数である．

問題

$(1) x\in\{0,1\}$	$(2) a=e_1(x)$
$(3) y\in\{0,1\}$	$(4) b=e_2(y)$
$(5) z\in\{0,1\}$	$(6) c=e_3(z)$

$(7) a\neq b\&b\neq c$　($a\neq b$ であっても $b\neq c$ を評価する)

この問題を生成検定法 で解くと以下のようになる (◯は正当な解，× は矛盾を示す)．

$$x=0,y=0,z=0,\cdots\times$$

$$x=0,y=0,z=1,\cdots\times$$

$$x=0,y=1,z=0,\cdots\bigcirc$$

$$x=0,y=1,z=1,\cdots\times$$

$$x=1,y=0,z=0,\cdots\times$$

$$x=1,y=0,z=1,\cdots\bigcirc$$

$$x=1,y=1,z=0,\cdots\times$$

$$x=1,y=1,z=1,\cdots\times$$

各解は 3 つの e_iの計算を必要とし，合計で 24 回の計算を必要とする．通常のバックトラック法 (*conventional backtracking*) は効率を改善する．この深さ優先 (*depth first*) の問題解決器は示された順に値を選択し，式を計算し，不等式を評価する．

$(1) x = 0$	$(8) x = 1$
$(2) x = 0, y = 0$	$(9) x = 1, y = 0$
$(3) x = 0, y = 0, z = 0, \cdots \times$	$(10) x = 1, y = 0, z = 0, \cdots \times$
$(4) x = 0, y = 0, z = 1, \cdots \times$	$(11) x = 1, y = 0, z = 1, \cdots \bigcirc$
$(5) x = 0, y = 1$	$(12) x = 1, y = 1$
$(6) x = 0, y = 1, z = 0, \cdots \bigcirc$	$(13) x = 1, y = 1, z = 0, \cdots \times$
$(7) x = 0, y = 1, z = 1, \cdots \times$	$(14) x = 1, y = 1, z = 1, \cdots \times$

この例では，14 回の e_iの計算だけが必要とされ，力ずく (*brute-force*) の方法では 24 回必要とする．このバックトラック法は Prolog で用いられている．問題解決器の動作を調べることで 8 回の e_iの計算が簡単に避けられることがわかる．

(1) 無駄なバックトラック (*futile backtracking*) ステップ (4) と (14) は無駄である．ステップ (3) の矛盾は $\{x = 0, y = 0\}$，$\{y = 0, z = 0\}$ が原因でありこの原因を同時に解決する選択点へバックトラックすればよく，ステップ (4) は安全に避けられる．矛盾が生じたときは矛盾を生じさせた選択点に戻るべきであり，最も近くの選択点に戻るべきではない．同様の議論によってステップ (14) も避けられる．

(2) 矛盾の再検知 (*rediscovering contradictions*) ステップ (10) とステップ (14) は無駄である．ステップ (3) が矛盾であるときその矛盾は $\{x = 0, y = 0\}$，$\{y = 0, z = 0\}$ に依存したものであり，y，zの値の等しいステップ (10) は決して試みられるべきではない．同様の議論によって，ステップ (14) は無駄であることが示される．

(3) **推論の再検知** (*rediscovering infereneces*) ステップ (6)，(7) 及び (9)–(14) の e_iの計算は必要がない．例えば，ステップ (7) はステップ (4) の $z = 1$ での階乗計算を繰り返している．バックトラックをすることで，既に計算された結果は取り消されてしまうが，結果を蓄えておくことができれば，各階乗計算は一度計算するだけでよい．

これらの問題点はTMSによって完全に避けられる．問題解決器は推論結果（依存関係）をTMSのデータベースに蓄えることによって，矛盾が生じたときはどこへバックトラックすればよいかを尋ねることができる．ステップ (3) を考える．依存関係の記録は $\{x = 0, y = 0\}$，$\{y = 0, z = 0\}$ が矛盾を生じさせるということを示している．このとき問題解決器は実際に矛盾を生じさせた最も近い選択点へバックトラックを行なう．このバックトラック法は関係依存型バックトラック (*dependency directed backtracking*)[14]，知的バックトラック (*intelligent backtracking*)[21] と呼ばれる．

3.3 ATMS

本節では，TMS として知られている ATMS(*Assumption-based* TMS[14, 15, 16]) について述べる．

3.3.1 問題解決器と ATMS

推論システム (*reasoning system, overall problem solver*) は 2 つの構成要素からなる．つまり問題解決器と ATMS である (図 3.1)．

問題解決器でなされた推論は ATMS に与えられ記録される．ATMS の仕事は “どのデータが信じることができ，信じることができないか” をこれまで記録された正当化 (*justification*)

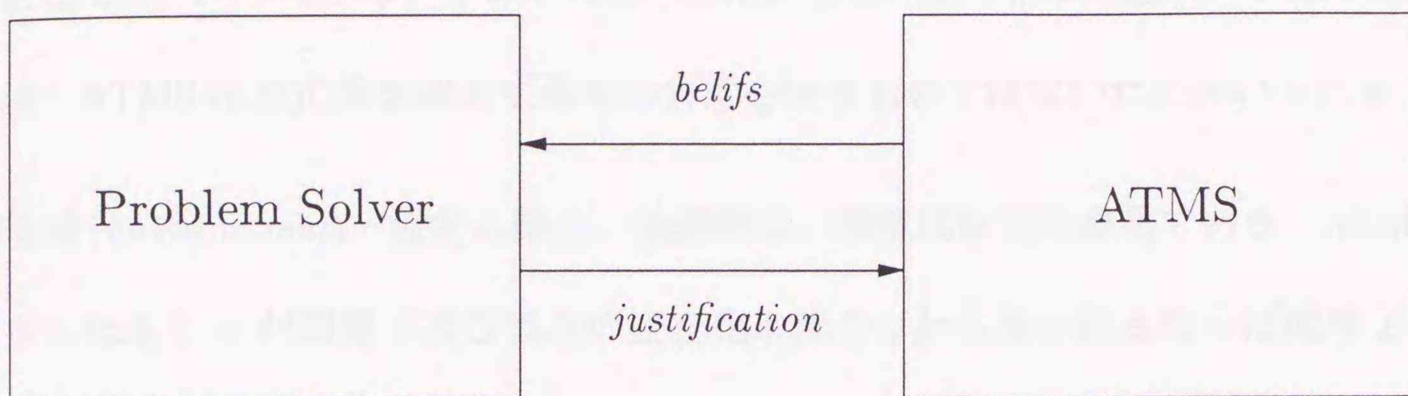


図 3.1: ATMS と問題解決器

から決定することである.

3.3.2 基本定義

ここでは，ATMS の特有な用語について述べる [14].

- 仮定 (*assumption*)：推論事実を導くプリミティブなデータ.
- ATMS ノード：問題解決器の事実を表わす．仮定は特別な種類のノードである.
- 正当化 (*justification*)：どのようにある ATMS ノードが他の ATMS ノードから導かれたかを示す．正当化は 3 つの部分からなる．つまり後件と呼ばれる正当化されている ATMS ノード，前件と呼ばれる ATMS ノードのリスト，*informant* と呼ばれる正当化の問題解決器の記述である．正当化は次のように書かれる.

$$x_1, x_2, \cdots \Rightarrow n$$

ここで $x_1, x_2, \cdots$ は前件の ATMS ノードであり，n は後件の ATMS ノードである．正当化は含意としてみることができる.

$$x_1, x_2, \cdots \rightarrow n$$

ここで $x_1, x_2, \ldots, n$ はアトムである．したがって，正当化は命題ホーン節である．$\Rightarrow$ は，ATMS は否定節を認めず通常の含意を扱うものではないため用いられる．

- 環境 (*environment*)：仮定の集合．論理的に，環境は仮定の連言である．ATMS ノード n はもし n が環境 E 及び現在の正当化の集合 J から導かれるならば環境 E において保持されるといわれる．導出は通常の命題演算によって定義される．

$$E, J \vdash n$$

ここで E はアトムの連言として，J は含意の集合として見ることができる．環境はもし偽 ($\perp$ で表わされる) が命題的に導くことができるならば矛盾である．

$$E, J \vdash \perp$$

- 文脈 (*context*)：無矛盾な環境から命題的に導くことのできるすべてのデータの集合．

3.3.3　ラベル

ラベル (*label*) は AMTS ノードを保持する環境の集合である．n のラベルのすべての環境は無矛盾であり $E, J \vdash n$ の性質を持つ．直感的には，$J \vdash E \rightarrow n$ である．ラベルはデータがどの仮定に依存するかを示し，正当化と異なり ATMS 自身によってつくられる．ATMS の仕事は各 ATMS ノードの各ラベルが無矛盾，健全，完全，極小であることを保証することである．

無矛盾性 ラベルは，もしその環境のすべてが無矛盾であるならば無矛盾である．

健全性 ATMS ノード n のラベルは，もしデータ n がそのラベルの各環境から導くことができるならば健全である．つまり，$J \vdash E \rightarrow n$.

完全性 ATMS ノード n のラベルは，もし $J \vdash E \rightarrow n$ となるすべての無矛盾な環境 E が n のラベルのある環境 E' の超集合ならば完全である．

極小性 ラベルは，もしそのラベルの環境が他の環境の超集合となるものが 1 つもなければ極小である．

上記の性質から問題解決器は ATMS ノードがある環境において保持されるかどうかを直接尋ねることができる．ATMS ノードはもし仮定の無矛盾な集合から導くことができなければかつそのときに限って空のラベルを持つ．ATMS ノードはもし仮定の無矛盾な集合から導くことができればかつそのときに限って非空のラベルを持つ．

3.3.4 基本データ構造

基本的なデータ構造は ATMS ノードである．ATMS ノードは問題解決器の事実と問題解決器のつくった正当化と ATMS によって計算されたラベルの 3 つのスロットを持つ．

$$\gamma_{datum} : \langle datum, label, justifications \rangle$$

ラベルは ATMS によって計算され問題解決器によって変えられてはならなず，正当化は問題解決器によって与えられ ATMS によって変えられてはならない．

ATMS ノードの種類は 4 つある．

前提ノード (*premises node*) 前提は前件を 1 つも持たない正当化を持つ．つまり，普遍的に成り立つ．ATMS ノード $\langle p, \{\{\}\}, \{()\} \rangle$ は前提 p を表わす．

仮定ノード (*assumption node*) 仮定は自分自身を言及する 1 つの環境を持つ ATMS ノード

である．ATMS ノード $\langle A, \{\{A\}\}, \{(A)\}\rangle$ は仮定 A を表わす．仮定は正当化されているかも知れない，例えば $\langle A, \{\{A\}, \{B, C\}\}, \{(A), (d)\}\rangle$.

仮定されたノード (*assumed node*) このATMS ノードは前提でもなく仮定でもなく，仮定を言及している正当化を持つ．例えば，仮定 A のもとで成り立つATMS ノード a は $\langle a, \{\{A\}\}, \{(A)\}\rangle$ と表わされる．

導出ノード (*derived node*) 上記の種類以外のATMS ノードはすべて導出ノードである．例えば，$\langle w=1, \{\{A, B\}, \{C\}, \{E\}\}, \{(b), (c, d)\}\rangle$ はATMS ノード b またはATMS ノード c，d から導かれることを示してる．加えて，このATMS ノードは環境 $\{A, B\}$，$\{C\}$，$\{E\}$ において保持されることを示している．

論理的に，ラベルは含意を表わしている．

$$\langle n, \{\{A_1, A_2, \ldots\}, \{B_1, B_2, \ldots\}, \ldots\}, \{(z_1, z_2, \ldots), (y_1, y_2, \ldots), \ldots\}\rangle$$

のラベルは，

$$(A_1 \wedge A_2 \wedge \cdots) \vee (B_1 \wedge B_2 \wedge \cdots) \wedge \cdots \rightarrow n$$

を表わしている．同様に，正当化は，

$$(z_1 \wedge z_2 \wedge \cdots) \vee (y_1 \wedge y_2 \wedge \cdots) \wedge \cdots \rightarrow n$$

を表わしている．

ATMS ノード

$$\gamma_\perp : \langle \perp, \{\ \}, \{\ldots\}\rangle$$

は矛盾を表わしている．矛盾環境は *nogood* と呼ばれる．論理的には，

$$A_1 \wedge A_2 \wedge \cdots \rightarrow \perp$$

を表わす．ここで $A_1, A_2, \ldots$ は仮定である．

3.3.5 環境束

すべての無矛盾な環境はある文脈を特徴付ける．図 3.2は仮定 $\{A, B, C, D, E\}$ についての環境束 (*environment lattice*) を表わしている．各ノードは環境を表わしている．環境 (つまりノード) からの上向きのエッジは大きさが 1 つ大きい環境への部分集合関係を表わしている．逆に環境からの下向きのエッジは大きさが 1 つ小さい環境への超集合関係を表わしている．ある環境のすべての超集合は束を上向きにたどることによって見つけることができ，すべての部分集合は束を下向きにたどることによって見つけることができる．したがって，すべての環境は $\{A, B, C, D, E\}$ の部分集合であり，また $\{\ \}$ の超集合である．$\perp$ を導出する環境は *nogood* であり束から取り除かれる．図 3.2では，線の引かれたノードが *nogood* に対応する．もしある環境が *nogood* ならばその環境の超集合のすべてが *nogood* である．図 3.2の *nogood* は *nogood*:$\{A, B, E\}$ によるものである．

ATMS はすべてのデータをその文脈で関連付ける．もしあるデータがある文脈にあるならば，その文脈のすべての超集合のなかにもそのデータはある (矛盾のある超集合は除く)．たとえば，図 3.2の楕円で囲まれたノードは$\gamma_{x+y=1}$のすべての文脈を表わしている．ATMS ノードのラベルは楕円で囲まれたノードの最大下界の集合である．

$$\gamma_{x+y=1} : \langle x + y = 1, \{\{A, B\}, \{B, C, D\}\}, \{\ldots\}\rangle$$

図 3.2の四角で囲まれたノードは$\gamma_{x=1}$の文脈に対応する．

$$\gamma_{x=1} : \langle x = 1, \{\{A, C\}, \{D, E\}\}, \{\ldots\}\rangle$$

ここで，yに関してはなにも知らず，問題解決器は $x + y = 1$ と $x = 1$ から $y = 0$ を導きだ

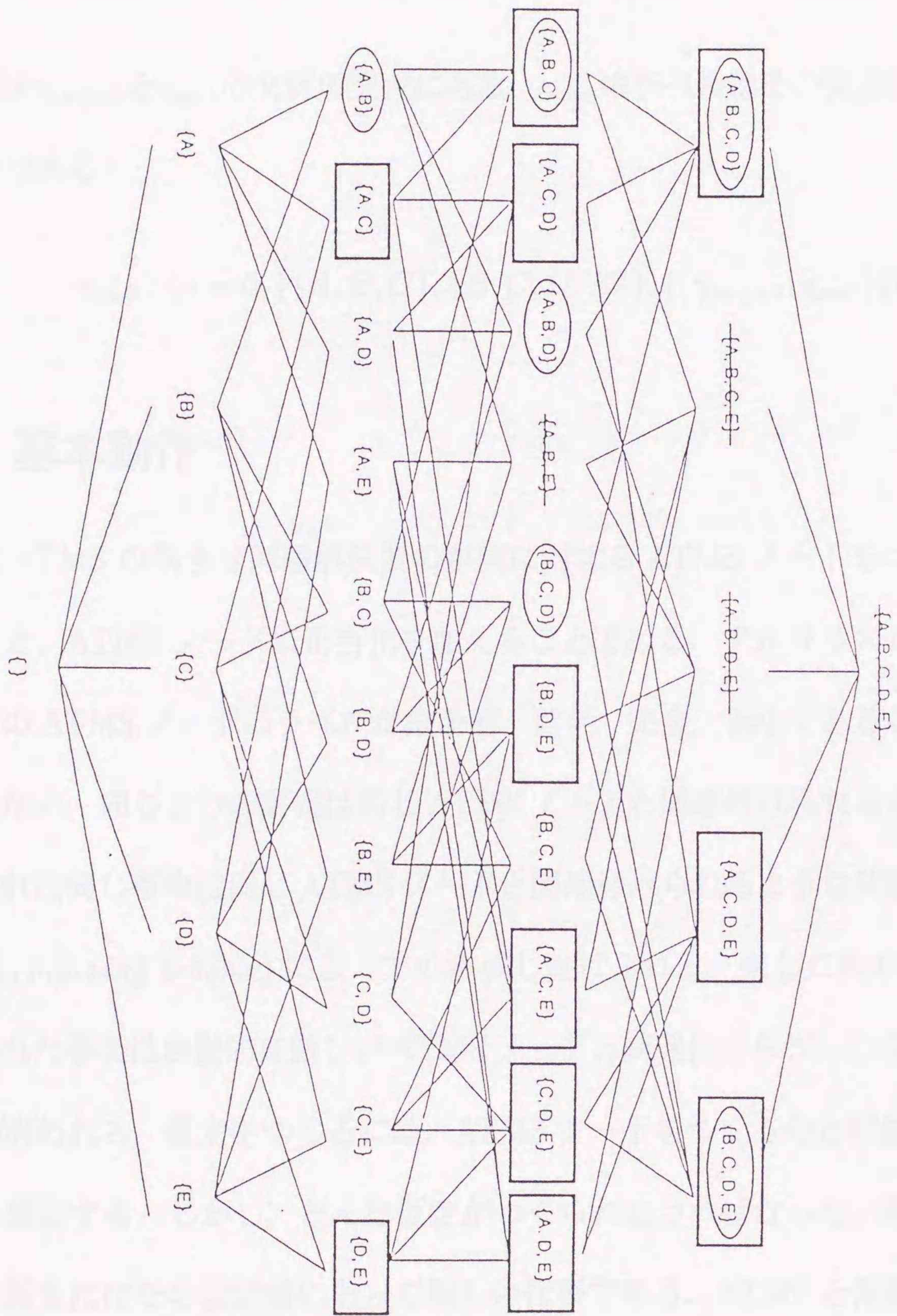


図 3.2: 環境束

したとする．問題解決器は次の正当化を ATMS に与える．

$$\gamma_{x+y=1}, \gamma_{x=1} \Rightarrow \gamma_{y=0}$$

$\gamma_{y=0}$の文脈は$\gamma_{x+y=1}$と$\gamma_{x=1}$の文脈の交点にある．$\gamma_{y=0}$のラベルはその交点にある文脈の最大下界の集合である．

$$\gamma_{y=0} : \langle y = 0, \{\{A, B, C\}, \{B, C, D, E\}\}, \{(\gamma_{x+y=1}, \gamma_{x=1})\}\rangle$$

3.3.6 基本動作

基本的な ATMS の働きは問題解決器の事実に対する ATMS ノードをつくること，仮定をつくること，ATMS ノードに正当化を加えることである．アルゴリズムは前記の動作の後，すべての ATMS ノードのラベルが無矛盾，健全，完全，極小であることを確かにしなければならない．同じ 2 つの事実は同じ ATMS ノードと関連付けられるということが重要である．これは同じ事実は同じ ATMS ノードと関連付けられるような問題解決器のある索引付け機構 (*indexing scheme*) によってのみ成し遂げられる．もしこれがなされなければ，新しく導かれた事実は自動的に新しい ATMS ノードと関連付けられ，このとき ATMS の多くの利点が失われる．仮定をつくるには，ATMS ノードをつくるのと同様少しだけ ATMS の仕事を必要とする．しかし，どんな仮定がつくられなければならないかを認識することは問題解決器またはその設計者にとって難しい仕事である．ATMS と問題解決器について重要な処理を生ずるプリミティブな操作のみが正当化を加える．

ATMS ノード n は仮定を含む正当化を与えることによって仮定される．例えば，

$$A \Rightarrow n$$

を与える．前提は1つも前件ノードを持たない正当化として表わされる．正当化，

$$\Rightarrow p$$

はATMSノード $\langle p, \{\{\ \}\}, \{(), \ldots\}\rangle$ を生ずる．$\gamma_\perp$に対する正当化はノードの矛盾のある連言を表わす (つまり $(n_1 \wedge n_2 \cdots \wedge n_k \rightarrow \gamma_\perp) \equiv \neg(n_1 \wedge n_2 \wedge \cdots \wedge n_k)$).

3.3.7 基本アルゴリズム

ATMSはすべての正当化に対して前件の文脈の和が後件の文脈に等しいことを保証しなければならない．ATMSは文脈の最大下界によって文脈を表わすため，この計算は複雑である．次の例を考える．

$$\gamma_{x+y=1} : \langle x+y=1, \{\{A,B\},\{B,C,D\}\}, \{\ldots\}\rangle$$

$$\gamma_{x=1} : \langle x=1, \{\{A,C\},\{D,E\}\}, \{\ldots\}\rangle$$

$$\gamma_{y=0} : \langle y=0, \{\ \}, \{\ \}\rangle$$

$$nogood{:}\{A,B,E\}$$

($nogood$ はATMSにとって中心的なものであるため，それらは別のデータベースに蓄えられる) 問題解決器が $x+y=1$ と $x=1$ から $y=0$ を導くとATMSに次の正当化をあたえる．

$$\gamma_{x+y=1}, \gamma_{x=1} \Rightarrow \gamma_{y=0}$$

この正当化の追加の結果，ATMSは$\gamma_{y=0}$のラベルを無矛盾，健全，完全，極小な形に更新する．

$$\gamma_{y=0} : \langle y=0, \{\{A,B,C\},\{B,C,D,E\}\}, \{(\gamma_{x+y=1}, \gamma_{x=1})\}\rangle$$

後件に対して健全で完全なラベルは前件ノードのラベルの環境の可能な組み合わせの和の集合である．したがって，この例の$\gamma_{y=0}$の健全で完全なラベルは，

$$\{\{A,B,C\},\{A,B,C,D\},\{A,B,D,E\},\{B,C,D,E\}\}$$

である．任意の健全で完全なラベルは，ある環境の超集合となっている環境 (この例の場合 $\{A,B,C,D\}$) と矛盾のある環境 (この例の場合 $\{A,B,D,E\}$) を取り除くことによって無矛盾で極小になる．

次に簡単なラベル更新アルゴリズムを示す．処理は問題解決器が新しい正当化を ATMS に与えることで始まる．

1. ラベル計算は集合演算または論理演算と見ることができる．後件の ATMS ノード n に対する k番目の正当化の i 番目の ATMS ノードのラベル j_{ik} が与えられると，ATMS ノード n に対する完全なラベルは，

$$\bigcup_k \{x \mid x = \bigcup_i x_i \quad \text{where} \quad x_i \in j_{ik}\}$$

で定義される集合演算によって計算することができる．もしラベルを連言標準形の命題論理式としてみるならば，完全なラベルは，

$$\bigvee_k \bigwedge_i j_{ik}$$

で計算される．矛盾のある環境，ある環境の超集合になっている環境を取り除いた後，ラベルは健全で極小になる．

2. もし新しく計算されたラベルが古いラベルと同じならば，その ATMS ノードについての処理を止める．

3. もし ATMS ノードが$\gamma_{\perp}$ならば，そのラベルの各環境が *nogood* データベースに追加され，すべての矛盾のある環境 (つまりその環境自身と超集合) がすべての ATMS ノードのラベルから取り除かれる．

4. もし ATMS ノードが$\gamma_{\perp}$でなければ，更新処理はすべての後件のラベル (つまりラベルが更新された ATMS ノードを言及している正当化を持つ他の AMTS ノード) の更新を再帰する．

この簡単な更新処理は 1 回以上同じ ATMS ノードを更新するかもしれないが，処理は有限個の仮定しか存在しないため停止しなければならない．ラベルはこのアルゴリズムが停止するまで無矛盾で完全であることは保証されない．

第 4 章

停止性検証の効率化

本章では，通常のバックトラック法を用いた停止性検証アルゴリズムの問題点を明確にし，その問題点を解決し効率を改善するための TMS の設計と，TMS を利用した停止性検証の有効性を計算機実験を通して述べる．

4.1 停止性検証アルゴリズム

本節では，通常のバックトラック法を用いた停止性検証アルゴリズムについて述べる．ここで述べるアルゴリズムは辞書式経路順序を用いるが，他の単純化順序を仮定しても同様である．

関数記号の集合 $\mathcal{F}$上の先行順序$\succ'$は$\succ \subseteq \succ'$(すなわち，$f \succ g$ならば $f \succ' g$) であるとき先行順序$\succ$の拡張 (*extension*) という．$l \succ_{lpo} r$のとき，$\succ_{lpo}$は書換え規則 $l \rightarrow r$を向き付けるという．

TRS : $\mathcal{R} = \{l_i \rightarrow r_i\}_{i=1}^{n}$及び関数記号の集合 $\mathcal{F}$上の先行順序$\succ$(初期値は空集合) が与え

られたとき，$\succ$のある拡張$\succ'$によって定義される辞書式経路順序$\succ'_{lpo}$によって$\mathcal{R}$のすべての書換え規則を向き付けることができるならば$\succ'$を返し，さもなくば，失敗を告げる (非決定) 手続きを tp$(\mathcal{R}, \succ)$ とする．また，項s，tと先行順序$\succ$が与えられたとき，$\succ$をある$\succ'$に拡張し ($\succ \subseteq \succ'$)，$s \succ'_{lpo} t$とできるならば$\succ'$を返し，さもなくば失敗を告げる (非決定) 手続きを tp1$(s, t, \succ)$ とする．

tp$(\mathcal{R}, \succ)$ は$n = 0$ならば$\succ$を返す．$n > 0$ならば手続き tp1$(l_1, r_1, \succ)$ によって，$\succ$を拡張し，最初の書換え規則の向き付けを試みる．もしこれに成功したら，ここで得られた拡張$\succ'$をさらに拡張し，残りの書換え規則$l_i \to r_i (i = 2, \ldots, n)$の向き付けを再帰的に試みる．すべての書換え規則を向き付けることができたならば，最終的に得られた先行順序$\succ'$を返す．途中である書換え規則の向き付けに失敗したならば，最も近くの選択点にバックトラックする．もし1つも選択点が存在しなければ失敗を告げる．

このアルゴリズムの正しさは，以下の補題から容易に確認できる．

補題 4.1.1 (辞書式経路順序の単調性) $\succ \subseteq \succ'$ならば$\succ_{lpo} \subseteq \succ'_{lpo}$である．

tp1$(s, t, \succ)$ のアルゴリズムを述べる．まず，sまたはtが変数のときは命題 2.3.1にしたがう．すなわち，

- sが変数ならば失敗を告げる．
- sが変数でなく，tが変数のとき，もしs中にtが生起していれば$\succ$を返す．さもなくば失敗を告げる．

sもtも変数でないとき$s \equiv f(s_1, \ldots, s_m)$，$t \equiv g(t_1, \ldots, t_n)$とする．この場合，辞書式経路順序の定義から，以下の場合がある．

(1) $f \succ g$ならば定義 2.3.3(2) にしたがって項 s と項 $t_1, \ldots, t_n$の各々を辞書式経路順序で比較する.

(2) $f = g$ならば定義 2.3.3(3) にしたがって $(s_1, \ldots, s_m)$ と $(t_1, \ldots, t_n)$ を辞書式順に辞書式経路順序で比較し，さらに項 s と項 $t_2, \ldots, t_n$の各々を辞書式経路順序で比較する.

(3) $g \succ f$ならば定義 2.3.3(1) にしたがって項 $s_1, \ldots, s_m$ のいずれかを非決定的に選択し，それと項 t を辞書式経路順序で比較する.

(4) (1) から (3) 以外，つまり関数記号 f，gが比較不能な場合，以下のいずれかを非決定的に選択し，実行する.

　(4-a) 先行順序を $[(f, g) \cup \succ]$ の推移的閉包に拡張し，上記 (1) と同様の比較を行なう.

　(4-b) 先行順序を $[(g, f) \cup \succ]$ の推移的閉包に拡張し，上記 (3) と同様の比較を行なう.

上記 (4) は補題 4.1.1によって正当化される．すなわち，f，gを比較不能とする先行順序$\succ$が解 (すべての書換え規則を向き付ける) ならば，f，gを比較可能とする半順序$\succ'$も解である．特に，$\succ$ の全順序への拡張$\succ^T$も解である．よって，全順序の解が存在しないならば，半順序の解も存在しない．したがって，(4) は$\succ$を全順序の範囲で探索することに対応する (ただし，この場合 $\mathtt{tp}(\mathcal{R}, \succ)$ の返す半順序の全順序への任意の拡張を解とする).

4.2　停止性検証における問題点

まず，停止性検証を行う際に推論手続きが生成する探索木に含まれる節点を定義する.

定義 4.2.1 (節点) 停止性検証手続きは以下の 4 種類の節点を生成する.

(1) P 選択点 (先行順序の拡張)：先行順序を $f \succ g$または $g \succ f$のいずれかに拡張する選択点である．この節点は楕円によって表わされ，楕円中に大小を決定したい関数記号 $f : g$を記入する．

(2) S 選択点 (部分項の選択)：辞書式経路順序の定義 2.3.3(1) における部分項 s_iの選択点である．この節点は長方形によって表わす．

(3) 境界点：書換え規則 $n+1$ の向き付け開始 (書換え規則 $1, 2, \ldots, n$ の向き付け成功) の時点に対応する節点であり，小円で表わす．

(4) 失敗点：ある書換え規則の向き付け失敗に対応する節点であり，$\times$で表わす．

通常のバックトラック法を用いたアルゴリズムの問題点を考察するため，次の 3 本の書換え規則からなる TRS を考える．

$$\begin{cases} i(h(g(h(x)))) \to f(g(j(x)), x) & r_1 \\ f(g(x), h(x)) \to i(i(x)) & r_2 \\ i(x) \to h(x) & r_3 \end{cases}$$

通常のバックトラック法を用いた停止性検証手続きでこの TRS の停止性を検証すると生成する探索木は図 4.1のようになる (変数との比較の部分木は含んでいない)．図 4.1の破線は各書換え規則の向き付けの過程に生成される節点を区切るために用いられている．各枝のラベルは，その枝をたどったときに選択し，拡張した関数記号の先行順序 (推移性から生じた先行順序対も含む) を表わす．S 選択点から出る枝のラベルは選択したサブゴールを表わす．

まず，問題解決器は書換え規則 r_1の向き付けを行なう．P 選択点 (1) から (3) までの左側の先行順序の拡張によって向き付けに成功する．さらに，問題解決器は r_2 の向き付けに進

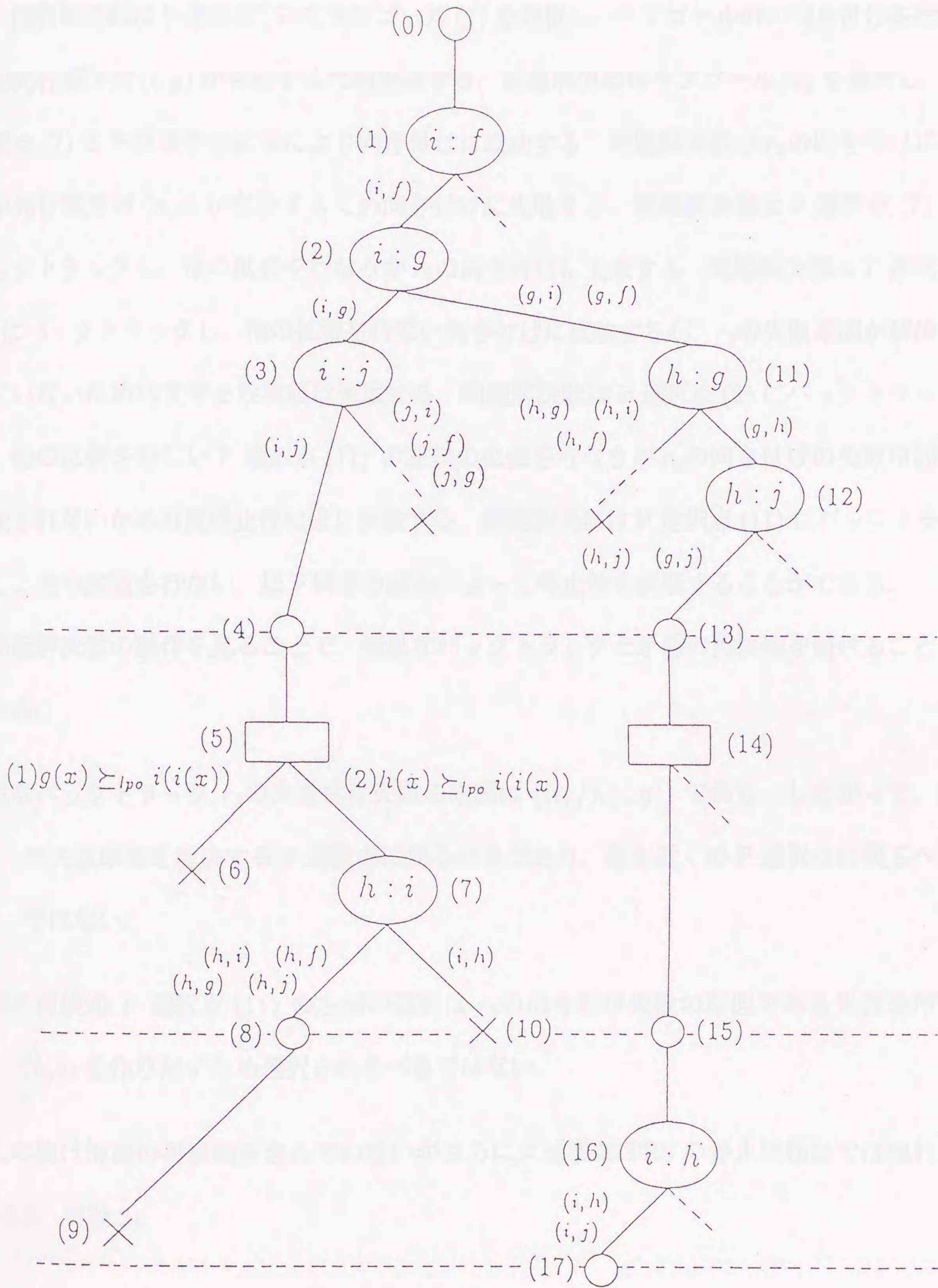


図 4.1: 通常のバックトラック法による探索木

む．問題解決器はS選択点(5)でサブゴール(1)を選択し，サブゴールのの向き付けを行なうが先行順序対(i,g)が存在するため失敗する．問題解決器はサブゴール(2)を選択し，P選択点(7)の先行順序の拡張により向き付けに成功する．問題解決器はr_3の向き付けに進むが先行順序対(h,i)が存在するため向き付けに失敗する．問題解決器はP選択点(7)にバックトラックし，他の拡張を行なうがr_2の向き付けに失敗する．問題解決器はP選択点(3)にバックトラックし，他の拡張を行ない向き付けに成功するが，r_2の失敗原因が解決されていないため再度停止性検証は失敗する．問題解決器はP選択点(2)にバックトラックし，他の拡張を行ないP選択点(11)の左側の拡張を行なうがr_3の向き付けの失敗原因が解決されないため再度停止性検証に失敗する．問題解決器はP選択点(11)にバックトラックし，他の拡張を行ない，以下同様の議論によって停止性を検証することができる．

問題解決器の動作を見ることで，無駄なバックトラックと矛盾の再検知を避けることができる．

無駄なバックトラック r_2の向き付け失敗の原因は$\{(i,f),(i,g)\}$である．したがって，この失敗原因を解決するP選択点に戻るべきであり，最も近くのP選択点に戻るべきではない．

矛盾の再検知 P選択点(11)の左側の選択はr_3の向き付け失敗の原因である先行順序対(h,i)を作りだすため選択されるべきではない．

この例は推論の再検知を含んでいないがさらに大規模なTRSの停止性検証では現れる(節**4.5**，実験2)．

4.3 停止性検証システムのための TMS の設計

本節では停止性検証システムのための TMS について述べる．TMS を用いて問題解決システムを設計する場合，“どのような推論を TMS に対して与えるか”，“TMS にどのような問い合わせを行なうか” という問題 (*encoding problem*) が生じる [14, 20]．この問題に対する一般的な解決法は存在しない．

本 TMS は前章で述べた ATMS に基づくが，停止性検証システム全体の効率を考え，ATMS ほど複雑な機能は持たせていない．ATMS はすべての解を見つける問題解決器向きであるが [14]，停止性検証の場合は 1 つの解しか必要としないからである．

4.3.1 TMS ノードと正当化

ATMS ノードは 3 つのスロットから構成されていたが，本 TMS では 2 つのスロットを持たせた．概念的には次のように表わす．

$$\gamma_{datum} : \langle datum, label \rangle$$

さらに以下のように表わすことができる．

$$\gamma_{l \to r} : \langle l \to r, \{\{(f_1, g_1), (f_2, g_2), \cdots\}, \ldots\} \rangle$$

ここで $l \to r (datum)$ は向き付けることのできた書換え規則，$\{(f_1, g_1), (f_2, g_2), \cdots\}$ すなわちラベルの各要素は，その書換え規則を向き付けることのできる環境 (関数記号の先行順序対の集合) である．ここで，(f_i, g_i) は $f_i \succ g_i$を意味する．以後，(f, g) を $f \succ g$と，また $\{(f_1, g_1), (f_2, g_2), \cdots\}$ は論理式 $(f_1 \succ g_1) \land (f_2 \succ g_2) \land \cdots$ と同一視する．

項 **3.3.4**の ATMS ノードと異なり，本 TMS では正当化のスロットを削除した．この理由は次のとおりである．

(1) 本 TMS を知的枝刈りの機能に限定した．

(2) 停止性検証手続きが直接書換え規則を向き付けることのできる環境を計算するようにした．

これらの理由によって，項 **3.3.7**で述べた TMS ノード間の依存関係の管理及びラベル更新アルゴリズムが不用になり，TMS のオーバーヘッドが減少し効率的になった．正当化は停止性検証手続きが生成した仮定 (関数記号の先行順序対) の集合 (つまり環境) を前件に持たせ，後件にその環境で向き付けることのできた書換え規則を持たせた．正当化は，

$$\{(f_1, g_1), (f_2, g_2), \ldots\} \Rightarrow l \to r$$

と表わす．論理的には，

$$f_1 \succ g_1 \wedge f_2 \succ g_2 \wedge \cdots \to l \succ_{lpo} r$$

を表わす．

nogood は，ある書換え規則 $l \to r$ の向き付け失敗に関与した環境である．

$$nogood{:}\{(f_1, g_1), (f_2, g_2), \ldots\}$$

は論理的には，

$$f_1 \succ g_1 \wedge f_2 \succ g_2 \wedge \cdots \to [\exists (l \to r) \in \mathcal{R}] \neg (l \succ_{lpo} r)$$

を表わす．*nogood* を TMS に与える正当化は，

$$\{(f_1, g_1), (f_2, g_2), \ldots\} \Rightarrow \bot$$

と表わす．ただし，以後は単に前件部を正当化という．

4.3.2　正当化の計算

本停止性検証システムの停止性検証手続きが TMS に対して与える正当化は，

(1) 書換え規則の向き付けに成功した場合の正当化

(2) 書換え規則の向き付けに失敗した場合の正当化

の 2 種類とした．

(2) の正当化は *nogood* であり，次項で述べる．(1) の場合，正当化の計算は境界点で行なわれる．

例えば，先行順序が $\{(f,g),(g,h),(f,h)\}$ であり，向き付けられた書換え規則が，

$$f(g(x)) \rightarrow g(h(x))$$

であるとき，向き付けに関与した先行順序対の集合 (環境) は $\{(f,g),(f,h)\}$ である．したがって，正当化は $\{(f,g),(f,h)\}$ となる．

他の例として，上記と同じ先行順序のもとで，向き付けられた書換え規則が，

$$h(f(x)) \rightarrow g(x)$$

とすると，正当化は $\{(f,g)\}$ となる．ここで先行順序対 (g,h) が正当化に含まれていないことに注意する．これは辞書式経路順序の定義 2.3.3 と正当化の論理的意味から明らかである．

4.3.3　*nogood* の計算

辞書式経路順序の定義 2.3.3(1)$s_i \succeq_{lpo} t$ を用いる場合，部分項 s_iの選択によっては向き付けに失敗する場合がある．しかし，この失敗に関与した環境が *nogood* であるとは限らな

い．サブゴール $s_i \succeq_{lpo} t$ に失敗しても別の部分項 s_j の選択によってサブゴール $s_j \succeq_{lpo} t$ に成功し，同じ環境で書換え規則の向き付けに成功する可能性が残されているからである．いずれの部分項を選択しても向き付けに失敗するときに限り，この環境が *nogood* となる．そこで部分的な失敗を表わす環境として *halfgood* という考えを導入し，*nogood* の計算に利用する．

halfgood は各失敗点で生成される．一般に，ノード n からノード m へバックトラックする際に，ノード n での *halfgood* がノード m に伝播される．ノード m では，すべての子ノードからの *halfgood* を適当な方法で合成し，新しい *halfgood* を伝播させる．境界点において *halfgood* は *nogood* となる．以下，各ノードでの *halfgood* の計算を具体的に述べる．

(1) 失敗点での計算

向き付け失敗に関与した環境を *halfgood* とする．例えば，先行順序が，

$$\{(f,g),(h,i),(h,j)\}$$

であり，向き付けに失敗した書換え規則が，

$$f(i(x)) \to g(h(x))$$

ならば，この書換え規則の向き付け失敗に関与した先行順序対の集合，つまり，環境は $\{(f,g)\ ,(h,i)\ \}$ である．この環境が失敗点での *halfgood* となる．

(2)P 選択点での計算

P 選択点では，$f \succ g$の選択に対応する部分木から *halfgood*:$\alpha_1 = \{(f,g)\} \cup \beta_1$と，$g \succ f$ の選択に対応する部分木から *halfgood*:$\alpha_2 = \{(g,f)\} \cup \beta_2$を受け取ったとき，$\alpha_1$と$\alpha_2$から超導出規則 (*hyperresolution rule*)[15] を用いて新しい *halfgood*:$\alpha_3 = \beta_1 \cup \beta_2$を生成し，親ノー

ドに伝播する．文献 [15] の表記法を用いると次のように表わすことができる．

$$\frac{\begin{array}{c} halfgood{:}\alpha_1 = \{(f,g)\} \cup \beta_1 \\ halfgood{:}\alpha_2 = \{(g,f)\} \cup \beta_2 \end{array}}{halfgood{:}\alpha_3 = \beta_1 \cup \beta_2}$$

例えば，図 4.2の P 選択点では一方の部分木から $halfgood{:}\{(h,i)\}$ と他方の部分木から $halfgood{:}\{(i,f),(i,h)\}$ を受け取り，新しい $halfgood{:}\{(i,f)\}$ を得る．

この計算の妥当性を確かめる．P 選択点では全順序の範囲で先行順序を拡張するため $f \succ g$あるいは $g \succ f$のどちらかが必ず成り立つ．

$$(f \succ g) \vee (g \succ f) \tag{4.1}$$

また $halfgood{:}\alpha_1$，α_2は論理的に以下のように表わすことができる (β_1,β_2を論理式と見なす．また，$s \succ_{lpo} t$ を示そうとしていたサブゴールとする)．

$$(f \succ g) \wedge \beta_1 \rightarrow \neg(s \succ_{lpo} t) \tag{4.2}$$

$$(g \succ f) \wedge \beta_2 \rightarrow \neg(s \succ_{lpo} t) \tag{4.3}$$

上記の式 4.1，4.2，4.3より，

$$\beta_1 \wedge \beta_2 \rightarrow \neg(s \succ_{lpo} t)$$

したがって，$\beta_1 \cup \beta_2$はサブゴール $s \succ_{lpo} t$ の失敗に関与する環境，すなわち $halfgood$ である．

上記は関数記号 fと gの先行順序を拡張しようとした P 選択点での選択が失敗原因ではなく，さらに前の P 選択点での選択に失敗原因があることを示している．

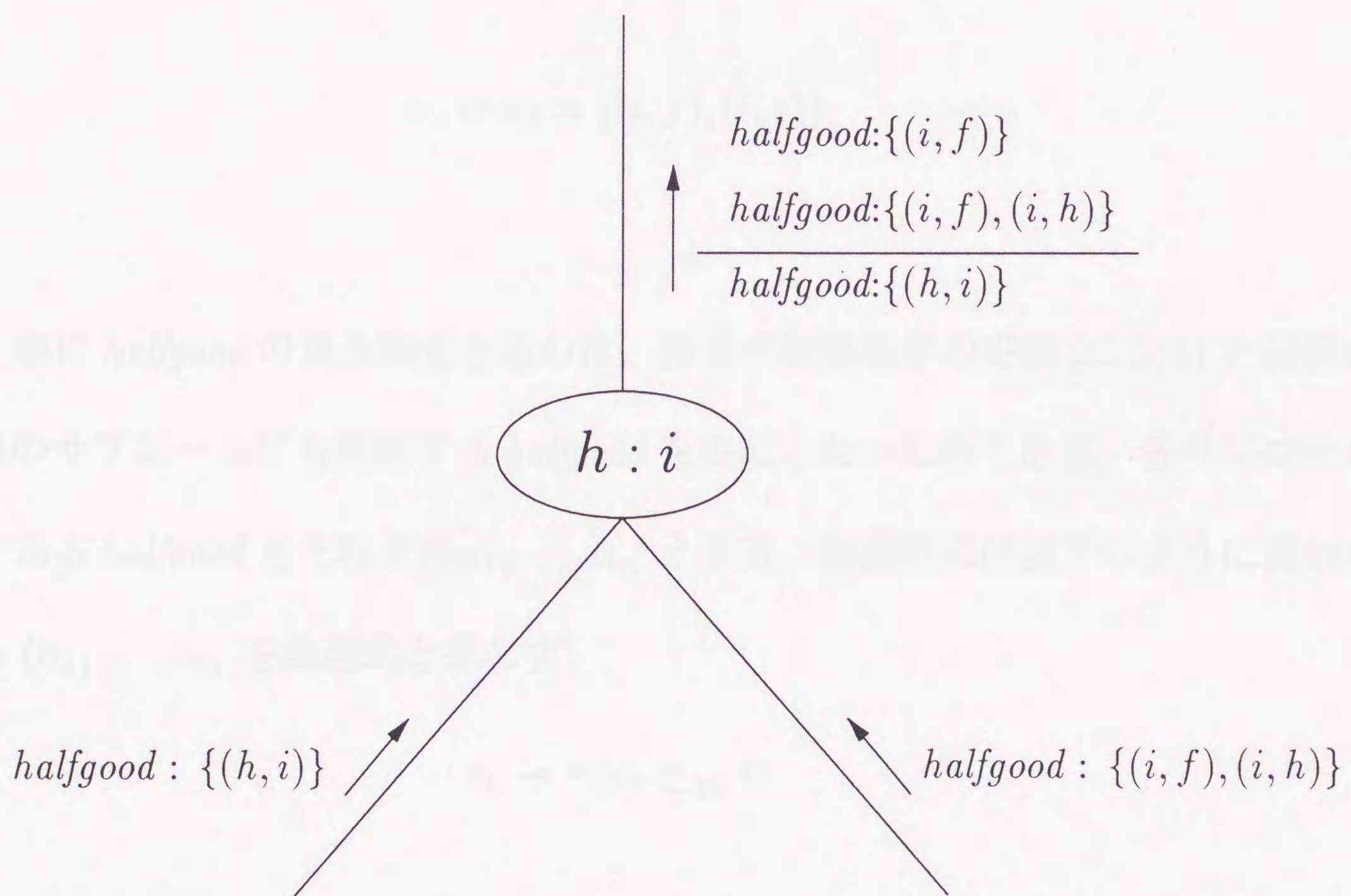


図 4.2: P 選択点における新 *halfgood* の計算

(3)S 選択点での計算

S 選択点では子ノードから伝播された *halfgood* の集合和をとり新 *halfgood* として親ノードに伝播する．例えば，図 4.3の S 選択点では，

$$halfgood{:}\alpha_1 = \{(i,f),(i,g)\}$$

と

$$halfgood{:}\alpha_2 = \{(i,f)\}$$

を受け取り，新しい *halfgood* は，

$$\alpha_1 \cup \alpha_2 = \{(i,f),(i,g)\}$$

となる．

ここで，単に *halfgood* の集合和をとるのは，辞書式経路順序の定義 2.3.3(1) と関係があり，いずれのサブゴールにも失敗する *halfgood* を生成したいためである．各サブゴールの失敗原因である *halfgood* をそれぞれ$\alpha_1,\ldots,\alpha_m$ とする．論理的には以下のように表わすことができる ($\alpha_1,\ldots,\alpha_m$ を論理式と見なす)．

$$\frac{\begin{array}{c}\alpha_1 \rightarrow \neg(s_1 \succeq_{lpo} t)\\ \vdots \\ \alpha_m \rightarrow \neg(s_m \succeq_{lpo} t)\end{array}}{\alpha_1 \wedge \cdots \wedge \alpha_m \rightarrow \neg(s_1 \succeq_{lpo} t) \wedge \cdots \wedge \neg(s_m \succeq_{lpo} t)}$$

したがって，$\alpha_1 \cup \cdots \cup \alpha_m$はすべてのサブゴールの向き付けを失敗させる環境である．つまり，$\alpha_1 \cup \cdots \cup \alpha_m$ が先行順序に包含される限り S 選択点からでるすべてのサブゴールの向き付けが失敗する．

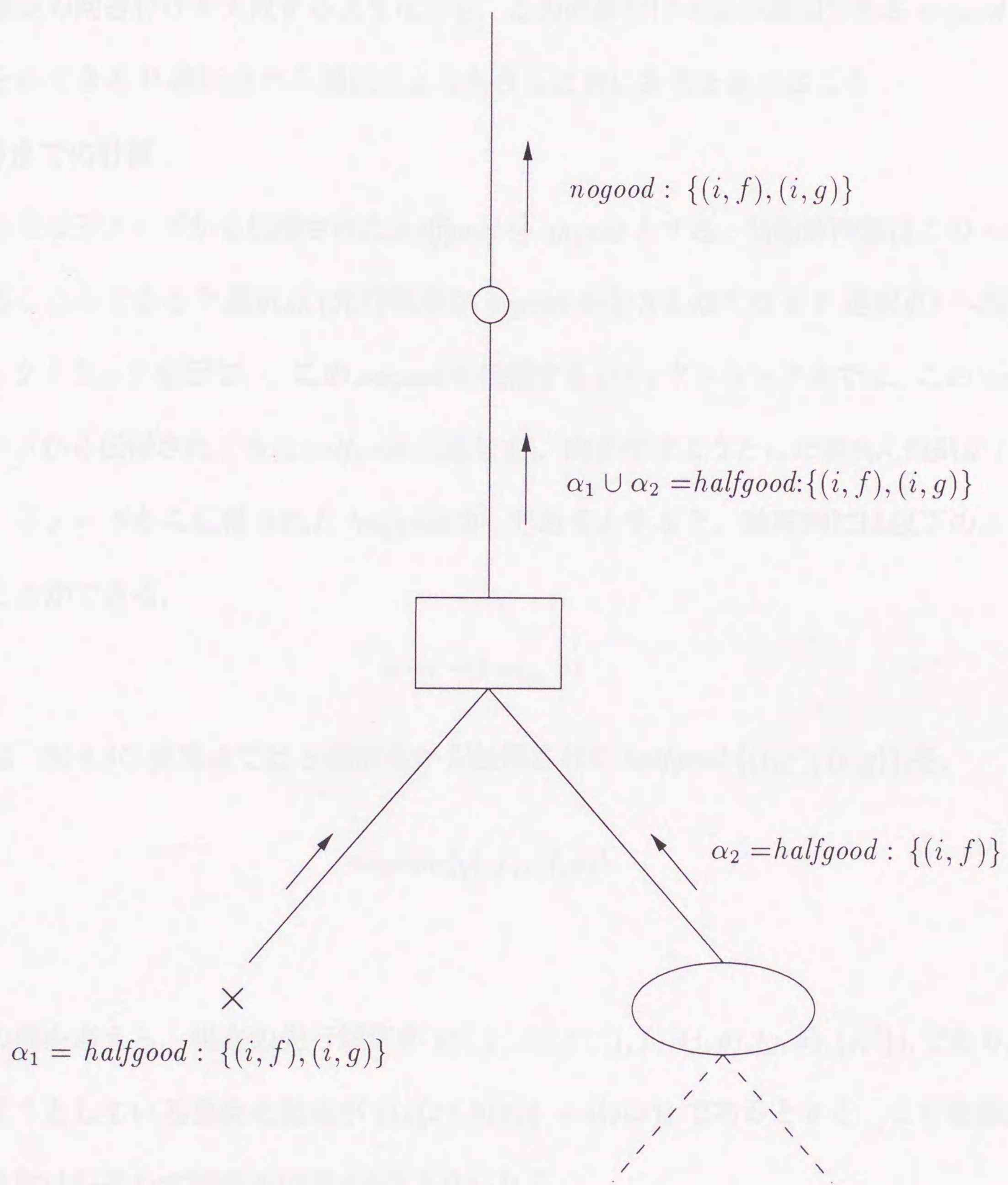


図 4.3: S 選択点における新 *halfgood* の計算

注意しなければならないことは S 選択点は関係依存型バックトラックにより飛びこされる場合があることである．これは S 選択点のあるサブゴールの向き付けには成功し，次の書換え規則の向き付けが失敗するようなとき，この向き付け失敗の原因である *nogood* を避けることのできる P 選択点が S 選択点よりもさらに前にあるときにおこる．

(4) 境界点での計算

境界点では子ノードから伝播された *halfgood* を *nogood* とする．問題解決器はこの *nogood* を避けることのできる P 選択点 (先行順序が *nogood* を包含しなくなる P 選択点) へ関係依存型バックトラックを行ない，この *nogood* を伝播する (バックトラック先では，この *nogood* を子ノードから伝播されてきた *halfgood* と見なす)．向き付けようとした書換え規則が $l \to r$ であり，子ノードから伝播された *halfgood* がαであるとすると，論理的には以下のように表わすことができる．

$$\alpha \to \neg(l \succ_{lpo} r)$$

例えば，図 4.3の境界点では S 選択点から伝播された *halfgood*:$\{(i,f),(i,g)\}$ を，

$$nogood{:}\{(i,f),(i,g)\}$$

とする．

実際の例を考える．現在の先行順序が $\{(i,f),(i,g),(j,f),(j,g),(j,h),(j,i)\}$ であり，向き付けようとしている書換え規則が $f(g(x),h(x)) \to i(j(x))$ であるとする．この書換え規則の向き付けを表わす探索木は図 4.4のようになる．

境界点 (1) からこの書換え規則の向き付けが始まる．(i,f) であるため，S 選択点 (2) が生成され，サブゴールは，

$$(a) g(x) \succeq_{lpo} i(j(x))$$

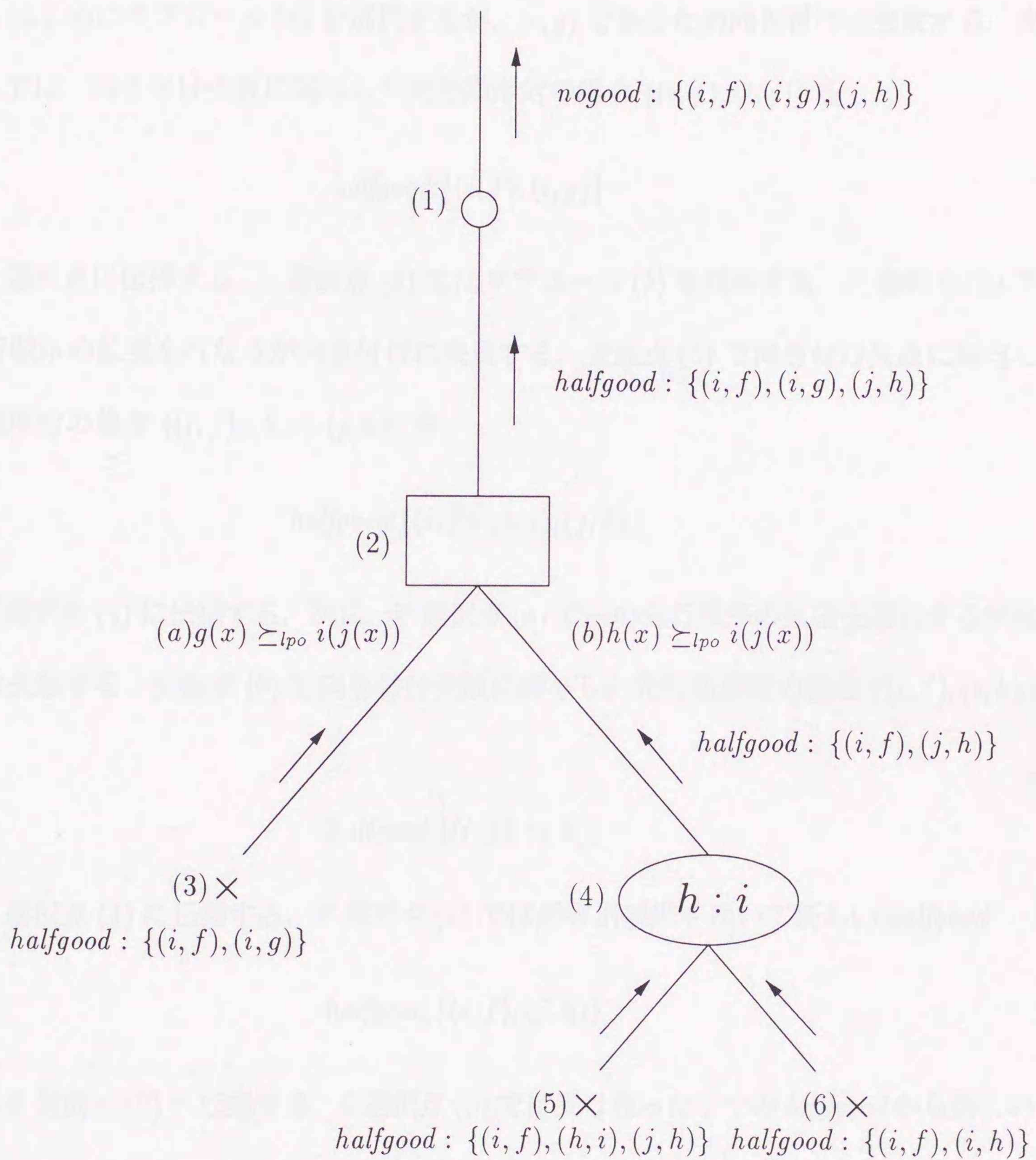


図 4.4: *nogood* の計算例

と

$$(b) h(x) \succeq_{lpo} i(j(x))$$

となる．はじめにサブゴール (a) を選択するが，(i,g) であるため向き付けに失敗する．失敗点 (3) では，向き付け失敗に関与した先行順序対の集合 $\{(i,f),(i,g)\}$ を

$$halfgood{:}\{(i,f),(i,g)\}$$

とし，S 選択点に伝播する．S 選択点 (2) ではサブゴール (b) を選択する．P 選択点 (4) で左の先行順序の拡張を行なうが向き付けに失敗する．失敗点 (5) で向き付け失敗に関与した先行順序対の集合 $\{(i,f),(h,i),(j,h)\}$ を

$$halfgood{:}\{(i,f),(h,i),(j,h)\}$$

とし，P 選択点 (4) に伝播する．次に，P 選択点 (4) で右の先行順序の拡張を選択するが向き付けに失敗する．失敗点 (6) で向き付け失敗に関与した先行順序対の集合 $\{(i,f),(i,h)\}$ を

$$halfgood{:}\{(i,f),(i,h)\}$$

とし，P 選択点 (4) に伝播する．P 選択点 (4) では超導出規則を用いて新しい *halfgood*

$$halfgood{:}\{(i,f),(j,h)\}$$

を生成し S 選択点 (2) へ伝播する．S 選択点 (2) では受け取った 2 つの *halfgood* から新しい *halfgood*

$$halfgood{:}\{(i,f),(i,g),(j,h)\}$$

を生成し，境界点 (1) へ伝播する．境界点 (1) では受け取った *halfgood* を *nogood* とし，この *nogood* を避けることのできる P 選択点へ伝播する．

4.3.4 TMS への登録

本停止性検証システムの問題解決器が TMS に対して推論結果を通知するのは以下の 2 つの場合とした．

(1) 各書換え規則の向き付けに成功したとき (境界点).

(2) 各書換え規則の向き付けに失敗してバックトラックする時点 (境界点).

(1) の場合，向き付けることのできた書換え規則 r と正当化 j の対を TMS に対して与える．TMS は，この正当化から TMS ノードの登録あるいは更新を行なう．この登録手続きを record-inference(r, j) とする．このとき，既に TMS データベースに書換え規則 r の TMS ノード γ_r が存在するならば，この TMS ノードのラベルに正当化 j を追加する．さもなくば，書換え規則 r の TMS ノードを作成し，登録する．

例えば，先行順序が $\{(f, g), (h, i)\}$ で，向き付けられた書換え規則が

$$f(x) \to g(x)$$

であるとすると，問題解決器は正当化，

$$\{(f, g)\} \Rightarrow f(x) \to g(x)$$

を登録手続き record-inference を用いて登録する．このとき TMS ノード，

$$\gamma_{f(x)\to g(x)} : \langle f(x) \to g(x), \{\{(f, g)\}\}\rangle$$

がつくられ登録される．

(2) の場合も (1) と同様 record-inference(r, j) を用いる．このとき r は向き付け失敗を表わす $\perp$，j は正当化である．このとき j は *nogood* データベースに登録される．登録の際に，も

し *nogood* データベースに j を包含する *nogood* が存在するならば，それらは冗長なので捨てられる．

さらに，もし j と既に登録されている *nogood* から超導出規則を用いて新しい *nogood* が得られるならば，その新しい *nogood* も登録する (このときも上記と同様に包含の検査を行なう)．

例えば，*nogood* データベースに，$\{\{(h,i),(f,g)\},\{(g,h)\}\}$ が登録されていたとする．このとき新しい *nogood*:$\{(i,h),(i,j)\}$ が登録されたとする．TMS は新しい *nogood* と既に登録されている *nogood* から超導出規則を用いて新しい *nogood* を生成する．論理的には次のように表わすことができる (*nogood* を論理式と見なす)．また停止性検証アルゴリズムでは$\succ$を全順序と仮定しているから $h \succ i$ か $i \succ h$ のいずれかが成り立たなければならない．

$$\frac{\begin{array}{c}(h \succ i) \vee (i \succ h) \\ \neg(h \succ i) \vee \neg(f \succ g) \\ \neg(i \succ h) \vee \neg(i \succ j)\end{array}}{\neg(f \succ g) \vee \neg(i \succ j)}$$

したがって，$\{(f,g),(i,j)\}$ も *nogood* である．よって *nogood* データベースは，

$$\{\{(h,i),(f,g)\},\{(g,h)\},\{(i,h),(i,j)\},\{(f,g),(i,j)\}\}$$

に更新される．

問題解決器は登録されているすべての *nogood* を避けるように関係依存型バックトラックを行なうことによって，通常のバックトラック法の問題点である無駄なバックトラックを避けることができる．

4.3.5　TMS への問い合わせ

本停止性検証システムの問題解決器が TMS に対して問い合わせを行なうのは以下の 2 つの場合とした．

(1) 先行順序の拡張を行なう各選択点 (P 選択点).

(2) 次の書換え規則の向き付けへ行く直前 (境界点).

(1) の場合，先行順序の拡張を行なうことによって，拡張後の先行順序が，ある *nogood* を包含することになるかどうかを検査する．

この問い合わせ手続きを nogoodp$(f, g, \succ)$ とする．ただし，f，gは大小を決定しようとしている関数記号，$\succ$は現在の先行順序とする．このとき，$[\{(f,g)\} \cup \succ]$ の推移的閉包が，ある *nogood* を包含するならば，真を返し，さもなくば，偽を返す．真ならば，問題解決器は (f,g) を選択しない．これによって通常のバックトラック法の問題点である，矛盾の再検知を避けることができる．

例えば，現在の先行順序が $\{(f,g),(h,i)\}$ であり，*nogood*:$\{(g,i)\}$ が登録されていたとする．このとき，ある P 選択点で問題解決器が (g,h) を選択し，先行順序を拡張すると $\{(f,g),(g,h),(h,i),(f,h),(f,i),(g,i)\}$ となる．拡張後の先行順序には$\succ$の推移性より *nogood*:$\{(g,i)\}$ が包含される．したがって，nogoodp を用いることによって，問題解決器は先行順序対 (g,h) を選択すべきではないことがわかる．

(2) の場合，向き付けようとしている書換え規則が既に，ある環境のもとで向き付けに成功している場合があるため，その書換え規則の TMS ノードのラベル中に現在の先行順序に包含される環境が存在するかどうかを検査する．

この問い合わせ手続きを orientp$(r, \succ)$ とする．ただし，rは向き付けようとしている書換え規則，$\succ$は現在の先行順序である．もし orientp が偽を返すならば，問題解決器はその書換え規則の向き付けを行ない，真ならばその書換え規則の向き付けを行なわず (補題 4.1.1 より，その書換え規則の向き付けは現在の先行順序のもとで保たれるため)，次の書換え規則の向き付けに進む．これによって通常のバックトラック法の問題点である，推論の再検知を避けることができる．

例えば，現在の先行順序を $\{(f,g),(h,i),(h,j)\}$ とし，向き付けようとしている書換え規則が，

$$h(x) \to i(x)$$

であり，既に登録されているこの書換え規則の TMS ノードが，

$$\gamma_{h(x)\to i(x)} : \langle h(x) \to i(x), \{\{(h,i)\}\}\rangle$$

であったならば，この TMS ノードを保持する環境が現在の先行順序に包含されるため，orientp を用いることによって，問題解決器はこの書換え規則の再向き付けを行なわないでよいことがわかる．

4.4 TMS 利用停止性検証システムの動作

本節では，前節で述べた TMS を用いた場合の停止性検証システムがいかにして通常のバックトラック法の問題点を避けうるかにについて例を用いて述べる．

例として用いる TRS は節 **4.2**で用いたものである．再度，以下に示す．

$$\begin{cases} i(h(g(h(x)))) \rightarrow f(g(j(x)), x) & r_1 \\ f(g(x), h(x)) \rightarrow i(i(x)) & r_2 \\ i(x) \rightarrow h(x) & r_3 \end{cases}$$

図 4.5は TMS を利用した場合の問題解決器が生成する探索木を示している (変数の比較の部分木は含んでいない)．各枝のラベルは，その枝をたどったときに選択し，拡張した関数記号の先行順序 (推移性からの先行順序対も含む) を表わす．S 選択点から出る枝のラベルは選択したサブゴールを表わす．

問題解決器は，まず境界点 (0) で書換え規則 r_1が既に，ある環境で向き付けに成功しているかどうかを orientp を用いて TMS に問い合わせる．その結果，orientp は偽を返し，問題解決器は書換え規則 r_1の向き付けを行う (以後，orientp を用いたこの問い合わせの記述は省略する)．書換え規則 r_1は P 選択点 (1) から P 選択点 (3) までの左側の先行順序の拡張によって向き付けに成功する．境界点 (4) で問題解決器は書換え規則 r_1の向き付けに用いた環境を正当化，

$$\{(i, f), (i, g), (i, j)\} \Rightarrow r_1$$

とし，record-inference を用いて TMS に与える．TMS は正当化を受け取り，TMS ノード，

$$\gamma_{r_1} : \langle r_1, \{\{(i, f), (i, g), (i, j)\}\}\rangle$$

を生成し，TMS データベースに登録する．

問題解決器は書換え規則 r_2の向き付けに進む．書換え規則 r_2 の左右の項の最も外側の関数記号は f，i であり，現在の先行順序では $i \succ f$であるため辞書式経路順序の定義 2.3.3(1) により，2 つのサブゴール $g(x) \succeq_{lpo} i(i(x))$ と $h(x) \succeq_{lpo} i(i(x))$ を生じる．S 選択点 (5) で

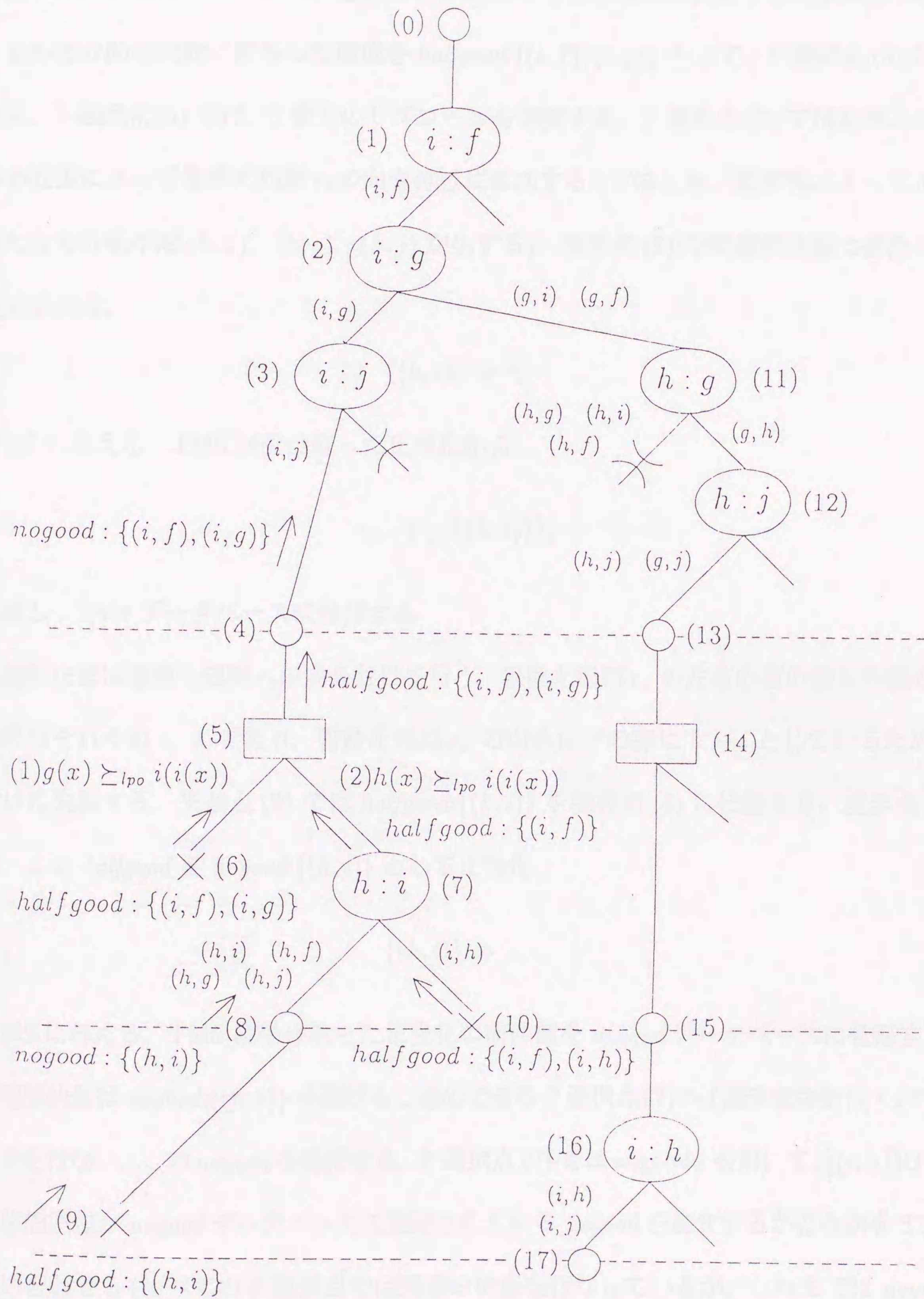


図 4.5: TMS 利用の場合の探索木

は，はじめに 1 番目のサブゴールを選択するが，$i \succ g$であるため失敗する．失敗点 (6) では，この部分的な失敗に関与した環境を *halfgood*:$\{(i,f),(i,g)\}$ として，S 選択点 (5) に伝播する．S 選択点 (5) では，2 番目のサブゴールを選択する．P 選択点 (7) では左側の先行順序の拡張によって書換え規則 r_2の向き付けに成功する (このとき，推移性によって 3 つの新たな先行順序対 (h,f)，(h,g)，(h,j) が生ずる)．境界点 (8) で問題解決器は書換え規則 r_2の正当化，

$$\{(h,i)\} \Rightarrow r_2$$

を TMS に与える．TMS は受け取った正当化から，

$$\gamma_{r_2} : \langle r_2, \{\{h,i)\}\}\rangle$$

を生成し，TMS データベースに登録する．

問題解決器は書換え規則 r_3の向き付けを行う．書換え規則 r_3 の左右の項の最も外側の関数記号はそれぞれ i，h であり，書換え規則 r_2 の向き付けの際に $h \succ i$ としているため向き付けに失敗する．失敗点 (9) では *halfgood*:$\{(h,i)\}$ を境界点 (8) に伝播する．境界点 (8) では，この *halfgood* を *nogood*:$\{(h,i)\}$ として正当化，

$$\{(h,i)\} \Rightarrow \bot$$

を TMS に与える．TMS は受け取った正当化の前件部を *nogood* データベースに登録する．

問題解決器は *nogood*:$\{(h,i)\}$ を避けることのできる P 選択点 (7) へ (関係依存型) バックトラックを行ない，この *nogood* を伝播する．P 選択点 (7) では nogoodp を用いて $[\{(i,h)\} \cup \succ]$ の推移的閉包が *nogood* データベースに登録されている *nogood* を包含するかどうかを TMS に問い合わせる (すべての P 選択点では同様の検査を行なっているが，これまでは *nogood* が登録されていなかったため省略している)．このとき nogoodp は偽を返し，この拡張を

選択することができる．しかし，この拡張では書換え規則 r_2 の向き付けに失敗する．失敗点 (10) では書換え規則 r_2 の向き付け失敗に関与した環境を $halfgood$:$\{(i,f),(i,h)\}$ としP選択点 (7) に伝播する．P 選択点 (7) では 2 つの $halfgood$:$\{(h,i)\}$ と $halfgood$:$\{(i,f),(i,h)\}$ から超導出規則を用いて新 $halfgood$:$\{(i,f)\}$ を生成し，S 選択点 (5) に伝播する．S 選択点 (5) では，すべてのサブゴールを選択しつくしたため 2 つの $halfgood$:$\{(i,f),(i,g)\}$ と $halfgood$:$\{(i,f)\}$ の集合和をとり，新 $halfgood$:$\{(i,f),(i,g)\}$ を境界点 (4) に伝播する．境界点 (4) では，この $halfgood$ を $nogood$ とし，正当化,

$$\{(i,f),(i,g)\} \Rightarrow \perp$$

をTMS に与える．TMS は受け取った正当化の前件部を $nogood$ データベースに登録する．

問題解決器は $nogood$:$\{(i,f),(i,g)\}$ を避けることのできるP 選択点 (2) へ関係依存型バックトラックを行なう．つまり，$nogood$ は同時には成り立たない先行順序対の集合であるから，現在の先行順序が $nogood$ を包含しなくなる最も近くのP 選択点にバックトラックすることになる．アルゴリズム的には $nogood$ を伝播した境界点の最も近い親P 選択点から順に，そのP 選択点で選択した先行順序対及び推移性からの先行順序対が $nogood$ の要素であるかどうかを検査する．この検査に成功したP 選択点が $nogood$ を避けることのできるP 選択点である．この結果，P 選択点 (3) の右の選択は刈り込まれる．これによって，通常のバックトラック法の問題点である，無駄なバックトラックを避けることができる．

P 選択点 (2) では，nogoodp を用いて先行順序対 (g,i) が選択できるかどうかをTMS に問い合わせる．このとき nogoodp は偽を返し，この選択を行なうことができる．P 選択点 (11) でも同様に先行順序対 (h,g) を選択できるかどうかを問い合わせる．この結果，先行順序の推移性から，(h,g) の選択後の先行順序が $nogood$:$\{(h,i)\}$ を包含するため nogoodp は真を返し，問題解決器は先行順序対 (h,g) の選択を行なわない．これによって，通常の

バックトラック法の問題点である，矛盾の再検知を避けることができる．

以後同様に P 選択点 (12)，(16) で先行順序の拡張を行ない，境界点 (13)，(15)，(17) で TMS に向き付け成功の正当化を与えることによってすべての書換え規則の向き付けに成功し，この TRS の停止性を検証することができる．問題解決器はこのときの先行順序，

$$\{(i,f),(g,i),(g,f),(g,h),(h,j),(g,j),(i,h),(i,j)\}$$

を返す．このときの TMS データベース，*nogood* データベースの内容を以下に示す．

- TMS データベース

$$\gamma_{r_1} : \langle r_1, \{\{(i,f),(h,j)\}, \{(i,f),(i,g),(i,j)\}\}\rangle$$
$$\gamma_{r_2} : \langle r_2, \{\{(g,i)\}, \{(h,i)\}\}\rangle$$
$$\gamma_{r_3} : \langle r_3, \{\{(i,h)\}\}\rangle$$

- *nogood* データベース

$$nogood{:}\{\{(i,f),(i,g)\},\{(h,i)\}\}$$

4.5 実験結果と TMS 利用の問題点

本節では，通常のバックトラック法によるインプリメントと TMS 利用のインプリメントを実験比較し，TMS 利用の有効性と問題点について述べる．

4.5.1 実験結果

本研究では，多くの実験を行なっているが典型的な結果を示す 2 つの実験結果について示す．ここで用いたインプリメントは $Symbolics$3620 上の LISP を用いて実現した．

(1) 実験 1

図 4.6は x に関する微分を行なう TRS である．α，βは変数である．この例では，表 4.1 からわかるように，通常のバックトラック法を用いたインプリメントも TMS を用いたインプリメントも共に，生成した P 選択点数 (10 個) は同じであり，一度もバックトラックすることなく停止性を検証することが可能である．したがって，通常のバックトラック法を用いたインプリメントの方が TMS を利用した場合より，実行速度は優れている．これは，TMS の利点が生かされず，TMS への登録，問い合わせによるオーバーヘッドが原因であると考えられる．ただし，バックトラックを行なうことなく停止性を示すことのできる TRS は構文的に非常に簡単であり，実行時間が非常に短いため，TMS 利用のオーバーヘッドは実用上容認することのできる時間であると考えられる．

(2) 実験 2

図 4.7は定理自動証明の応用として，ある論理回路の故障診断を行なう際に用いられた TRS である [22]．(1) から (10) の書換え規則は論理回路の結線情報，(11) から (21) の書換え規則はブール代数公理，(22) から (24) の書換え規則は論理ゲートの機能を表わしている．

この例の場合，通常のバックトラック法を用いたインプリメントでは生成された P 選択点数が 1490 個であるのに対し，TMS を用いたインプリメントでは 56 個に減じられている．これは TMS の利点である無駄なバックトラックを避けることにより 29 本の枝刈りを行なったこと (刈り込んだ枝の下の部分探索木の枝は本数に含めていない)，また，推論の再検知を避けることにより 13 本の書換え規則の再向き付けを行なわなかったことによる．特に無駄なバックトラックを避ける関係依存型バックトラックによる枝刈りが非常に効率を改善している．

表 4.1: 停止性検証システムの実験結果

TRS	実験 1		実験 2	
インプリメント	BT 法	TMS 利用	BT 法	TMS 利用
関係依存型バックトラックによる枝刈り	–	0	–	29
nogoodp による枝刈り	–	0	–	0
orientp による枝刈り	–	0	–	13
P 選択点数	10	10	1490	56
実行時間 (sec)	0.039	0.063	138.03	1.608

(1)$d(x) \rightarrow 1$

(2)$d(c) \rightarrow 0$

(3)$d(+(\alpha,\beta)) \rightarrow +(d(\alpha),d(\beta))$

(4)$d(*(\alpha,\beta)) \rightarrow +(*(d(\alpha),\beta),*(\alpha,d(\beta)))$

(5)$d(-(\alpha,\beta)) \rightarrow -(d(\alpha),d(\beta))$

(6)$d(minus(\alpha)) \rightarrow minus(d(\alpha))$

(7)$d(/(\alpha,\beta)) \rightarrow -(/(d(\alpha),\beta),*(\alpha,/(d(\beta),\wedge(\beta,2))))$

(8)$d(ln(\alpha)) \rightarrow /(d(\alpha),\alpha)$

(9)$d(\wedge(\alpha,\beta)) \rightarrow +(*(*(\beta,\wedge(\alpha,-(\beta,1))),d(\alpha)),*(*(\wedge(\alpha,\beta),ln(\alpha)),d(\beta))$

図 4.6: 実験 1(x に関する微分)

$(1) in1(e1) \rightarrow out(s1)$	$(2) in1(a1) \rightarrow out(s1)$
$(3) in2(e1) \rightarrow out(s2)$	$(4) in2(a1) \rightarrow out(s2)$
$(5) in1(a2) \rightarrow out(s3)$	$(6) in2(e2) \rightarrow out(s3)$
$(7) in2(a2) \rightarrow out(e1)$	$(8) in1(e2) \rightarrow out(e1)$
$(9) in2(o1) \rightarrow out(a1)$	$(10) in1(o1) \rightarrow out(a2)$
$(11) not(0) \rightarrow 1$	$(12) not(1) \rightarrow 0$
$(13) and(0, x) \rightarrow 0$	$(14) and(1, x) \rightarrow x$
$(15) and(x, x) \rightarrow x$	$(16) or(0, x) \rightarrow x$
$(17) or(1, x) \rightarrow 1$	$(18) or(x, x) \rightarrow x$
$(19) xor(0, x) \rightarrow x$	$(20) xor(1, x) \rightarrow not(x)$
$(21) xor(x, x) \rightarrow 0$	$(22) out(x) \rightarrow and(in1(x), in2(x))$
$(23) out(x) \rightarrow xor(in1(x), in2(x))$	$(24) out(x) \rightarrow or(in1(x), in2(x))$

図 4.7: 実験 2(論理回路の結線情報とブール代数公理)

4.5.2 TMS 利用の問題点

TMS を利用する際の問題点として，前項の実験 **(1)** で述べたように，

(1) TMS への登録のオーバーヘッド.

(2) TMS への問い合わせのオーバーヘッド.

が挙げられる.

(1) のオーバーヘッドは各境界点で生じる．バックトラックを一度も生じることなく停止性を検証できる TRS の場合，各書換え規則の TMS ノードを作成するだけになり後の推論には必要の無いものとなる.

(2) の問い合わせによるオーバーヘッドは各境界点，P 選択点で生じる．これらの選択点では常に，orientp，nogoodp の問い合わせ手続きを用いるため，はじめて向き付けようとしている書換え規則に対しても orientp による問い合わせが行なわれる．同様に *nogood* が登録されていないにもかかわらず先行順序の拡張を行なう前に問い合わせを行なってしまう.

これらのオーバーヘッドは TMS の利点とのトレードオフであり，特に (1) のオーバーヘッドはバックトラックが生じるかどうかということは問題解決器にはわからないことであり避けようがない.

4.6 議　論

停止性は一般に決定不能であるから，本停止性検証システムの論理的基礎となっている辞書式経路順序による検証法にも限界がある．本節では，本方式に関したいくつかの検証法についてその関係を紹介し，本方式の位置付けを明らかにする.

図4.8はいくつかの方法(クラス)をその包含関係により図式化したものである．単純停止[5]とは，単純化順序によって停止性を検証できるクラスである．経路順序[2]は関数記号の集合上の半順序または擬順序(*quasi-ordering*)(先行順序と呼ばれる)を項の集合上に構文的に拡張して定義される単純化順序によって停止性を検証できるクラスである．擬順序$\succsim$とは反射的かつ推移的な二項関係である．$f \succsim g$ならばかつそのときに限って$f \succ g \wedge \neg(g \succ f)$，$f \approx g$ならばかつそのときに限って$f \succsim g \wedge g \succsim f$と定義する．$\succ$は半順序，$\approx$は同値関係となる．任意の$f$，$g$に対して$f \succsim g \vee g \succsim f$，すなわち$f \succ g \vee g \succ f \vee f \approx g$が成り立つとき擬順序$\succsim$は全順序的(*total*)であるという．

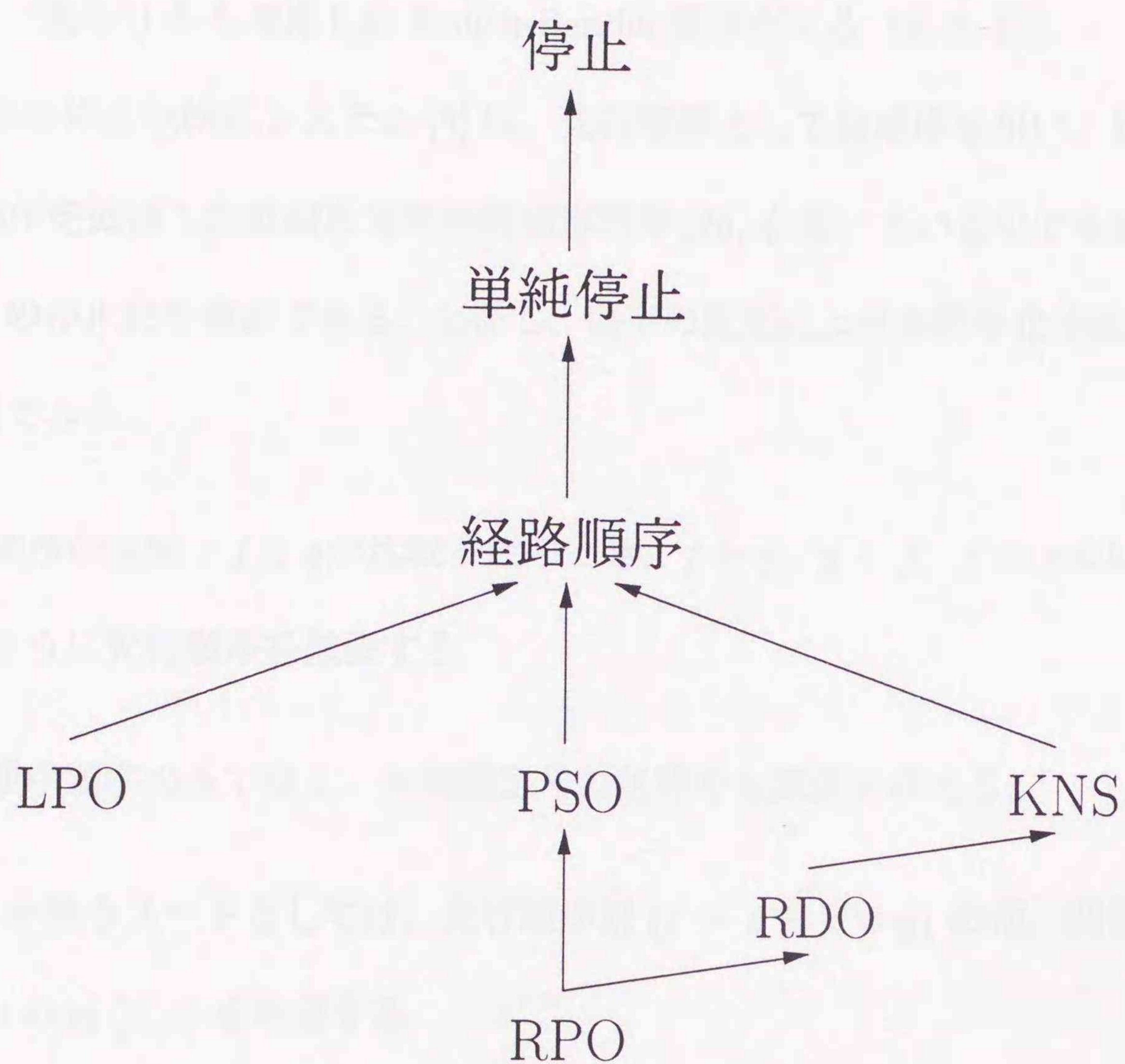
図4.8中の英字3文字からなる記号は特定の経路順序を略記したものである[23]． 実用上は，定義の簡潔さや効率の面から辞書式経路順序と再帰的経路順序がよく使われる[1]．

図4.8中で明示されていない包含関係が成立しないことを示すいくつかの例を挙げる．

- $\mathcal{R} = \{f(0,1,x) \to f(x,x,x)\}$ は停止するが単純停止しない[5]．
- $\mathcal{R} = \{f(a) \to f(b), g(b) \to g(a)\}$ は単純停止する[25]が，経路順序でそれを示せない[17]．
- $\mathcal{R} = \{f(f(x,y),z) \to f(x,f(y,z))\}$ は辞書式経路順序で停止性を示せるが再帰的経路順序では示せない．
- $\mathcal{R} = \{f(g(x),y) \to f(y,x)\}$ は再帰的経路順序で停止性を示せるが辞書式経路順序では示せない．

その他にも種々の経路順序がある．Steinbach は各関数記号に left，right，mult のいず

[1] 2つの基底項が辞書式経路順序によって順序付け可能かどうかの判定は，$O(n \log n)$であることが示されている[24]．ここで，nは基底項に含まれる関数記号の数である．



LPO : Lexicographic Path Ordering of Kamin & Lévy

PSO : Path of Subterms Ordering of Plaisted

RPO : Recursive Path Ordering of Dershowitz

KNS : Path Ordering of Kapur, Narendran and Sivakumar

RDO : Recursive Decomposition Ordering of Jouannaud et al.

図 4.8: 停止性検証法の包含関係

れかの状態 (*status*) を割り当てることにより，より強力な経路順序を定義し，それらの関係を論じている [26]．また，順序を構文のみで定めるのではなく，意味的な情報 (各関数記号に対する “重み”) をも考慮した Knuth-Bendix 順序がある [10, 2, 17]．

Detlefs 等の停止性検証システム [9] は，先行順序として擬順序を用い，状態を用いて再帰的経路順序を拡張した拡張再帰的経路順序順序 [26] を用いているので本論文の方式より広いクラスの停止性を検証できる．しかし，以下の変更により本効率化手法を Detlefs 等の方法に応用できる．

(1) 先行順序の拡張：f と g が比較不能のとき，$f \succ g$，$g \succ f$，$f \approx g$ のいずれかが成り立つように先行順序を拡張する．

(2) 先行順序順序のみでなく，各関数記号の状態をも探索で求める．

(3) TMS が扱うノードとしては，先行順序対 ($f \succ g$ や $f \approx g$) の他，関数記号 f とその状態 s の対 $\langle f, s \rangle$ を考慮する．

4.7　まとめ

本章では，従来の停止性検証アルゴリズムにおける通常のバックトラック法の計算効率上の問題点を明確にし，その問題点を解決するための TMS の設計・開発について述べ，実験結果を通してその有効性を示した．

第 5 章

完備化手続きの効率化

一般に，完備化手続きは等式集合と (生成される項書換えシステムの停止性を保証する) 簡約順序を入力として，完備な項書換えシステムを生成する．本論文で対象としている手続きは，入力として複数個の簡約順序を受け取り，その中から適切なものを自動的に選択するものとしている．この手続きは，論理的には，入力として簡約順序を 1 つだけ受けとる手続きを 1 つのプロセスとし，これらを並行 (*concurrent*) に動作させたものと等価である．しかし，その場合には，各プロセス間で多くの推論の重複が生じ，無駄が多い．

本章では，完備化手続きの問題点を明確にし，その問題点を避けるために完備化手続きを並行に実行する並行完備化手続きについて考察し，さらに並行完備化手続きの問題点である推論の重複を避けるため ATMS のデータ構造に基づき設計・開発した複数簡約順序完備化手続きについて述べる．

5.1 完備化手続きの問題点

完備化手続き [10] は等式集合と簡約順序が与えられたとき，

(i) 完備な項書換えシステムを生成し成功する，

(ii) ある等式をその簡約順序で向き付けられずに失敗する，

(iii) 無限に手続きが停止せずに発散する，

のいずれかの結果を生ずる．完備化の成功は与えられた簡約順序に大きく依存する．したがって，完備化手続きの問題点として，少なくとも以下の 3 点が挙げられる．

(1) 生成される TRS の停止性を保証するために必要な簡約順序を利用者に要求する．

(2) 簡約順序が不適当な場合，手続きが無限に継続する場合がある．

(3) 簡約順序が適切な場合でも (完備な TRS が存在するにもかかわらず) 失敗する場合がある．

(1) の問題点は利用者に停止性に関する知識を要求する．また，多少その知識があったとしても，利用者が適切な簡約順序を与えることは難しい．(2) の問題点に関しては，Hermann による 5 つの発散解決法が提案されている [27]．(3) の問題点に関しては，Bachmair 等による基礎項上に限定した無失敗完備化 (*unfailing completion*)[28] や順序付完備化 (*ordered completion*)[2] と呼ばれる手続きが提案されている．ただし，通常の完備化手続きが発散する場合，これらの手続きでも発散する．以下では，特に問題点 (2) について例を用いて述べる．

KB を用いるためには，推論規則 **Orient** で用いる簡約順序$\succ$を必要とする．この簡約順序に依存して，完備化が発散する場合がある．次の簡単な例を考える．

例題 5.1.1

$$\mathcal{E} = \begin{cases} (x+y)+z \leftrightarrow x+(y+z) & (1) \\ f(x)+f(y) \leftrightarrow f(x+y) & (2) \end{cases}$$

簡約順序として先行順序$\succ^1 = \{(+,f)\}$のもとでの辞書式経路順序$\succ^1_{lpo}$とし，$\mathcal{E}$を完備化する．

等式 (1) は **Orient** の適用によって書換え規則

$$r_1 : (x+y)+z \to x+(y+z)$$

となり，等式 (2) は

$$r_2 : f(x)+f(y) \to f(x+y)$$

となる．書換え規則 r_1，r_2から **Deduce** の適用によって新たな等式，

$$e_3 : f(x+y)+z \leftrightarrow f(x)+(f(y)+z)$$

が得られ，**Orient** の適用により，書換え規則

$$r_3 : f(x+y)+z \to f(x)+(f(y)+z)$$

が得られる．さらに，書換え規則 r_2，r_3から同様に **Deduce**，**Orient** の適用によって新たな書換え規則

$$r_4 : f^2(x+y)+z \to f^2(x)+(f^2(x)+(f^2(y)+z)$$

が得られる．この **Orient**，**Deduce** の推論規則の適用は無限に続き，書換え規則

$$f^n(x+y)+z \to f^n(x)+(f^n(x)+(f^n(y)+z)$$

の無限の族を生成する．

先行順序を$\succ^2 = \{(f,+)\}$とし，辞書式経路順序のクラスを変更することによって，等式(2) は書換え規則 r_2の逆に向き付けられ，

$$r_2' : f(x+y) \to f(x) + f(y)$$

となり，書換え規則 r_1，r_2'を生成し完備化に成功する．

この例からわかるように，与える簡約順序によって完備化の発散を避けることができる．この例では簡約順序を辞書式経路順序として，完備化に成功したが，簡約順序のクラスを変更をしなければならない場合もある [27].

このような手続きの発散を避ける明らかな方法は簡約順序の変更である．この方法はある書換え規則の向きを逆向きにすることによって発散の原因を取り除く．ただし，候補となる簡約順序は複数存在するため適切な簡約順序を予め決定することは難しい．したがって，適切な簡約順序の候補を複数個用意し，各簡約順序のもとで完備化を試みるしかない場合がある．その場合，手続きは発散する可能性があるため，複数の簡約順序を逐次的に試していく方法は不適切である．

発散を避けるようなアプローチとして以下の 2 通りが考えられる．

- 既存のシステム (REVE[12] や RRL[11] 等) のように，簡約順序を適当な経路順序に定め，利用者と対話しながら先行順序を順次拡張し，完備化を行う方法：利用者が誤りに気がついたときは，割り込みによって適当な選択点にバックトラックして間違った先行順序の取り消しや新たな選択を行う．しかし，利用者は停止性に関する知識が必要であり，バックトラック (深さ優先探索) を利用しているため，利用者が適切な割り込みをしなければ解があっても得られない場合がある．

- 手続きを各簡約順序毎に並行に動作させる方法：この方法では各プロセス間に重複した推論が生じ，効率が良くない．

次節では 2 つめの方法の問題点を例を挙げて述べる．

5.2 並行完備化手続き

KB を並行に動作させることは非常に簡単である．簡約順序の集合 $O = \{\succ_1, \ldots, \succ_n\}$ と等式集合 Eが与えられたとき，各簡約順序$\succ_i$ごとに KB をプロセスとして発生し，ある簡約順序$\succ_i$で完備化成功ならばそのときの簡約順序$\succ_i$ と得られた TRS を返せばよい．この手続きを CKB とする．CKB の問題点を例を挙げて述べる．

例題 5.2.1 次の等式集合 $\mathcal{E}$と簡約順序として先行順序$\succ^1 = \{(-,+),(+,0),(-,0)\}$ $\succ^2 = \{(+,-),(-,0),(+,0)\}$ とする辞書式経路順序$\succ^1_{lpo}$，$\succ^2_{lpo}$が CKB に与えられたとする．

$$\mathcal{E} = \begin{cases} 0 + x \leftrightarrow x & (e1) \\ -x + x \leftrightarrow 0 & (e2) \\ (x+y)+z \leftrightarrow x+(y+z) & (e3) \end{cases}$$

CKB は受けとった簡約順序数分だけのプロセスを発生するため，この例では 2 つのプロセス KB_1 と KB_2を発生したことになる．KB_1の動作を追ってみる．まず，KB_1では **Orient** の適用により，$(e1)$，$(e2)$，$(e3)$ の等式を向き付け，書換え規則とする．

$$\begin{array}{ll} 0 + x \rightarrow x & (r1) \\ -x + x \rightarrow 0 & (r2) \\ (x+y)+z \rightarrow x+(y+z) & (r3) \end{array}$$

さらに，**Deduce** の適用により，$r2$ と $r3$ から危険対 $(e4): -x + (x + y) \leftrightarrow 0 + y$ を得る．$(e4)$ は **Simplify** の適用によって $(e4'): -x + (x + y) \leftrightarrow y$ となり，**Orient** の適用によって書換え規則 $(r4): -x + (x + y) \rightarrow y$ となる．KB_2でもこれまでの動作は同じである．以下同様に KB_1と KB_2 間に重複した推論が数多く現れながら完備化が進む．完備化の途中で等式$-(x+y) \leftrightarrow -x + -y$が生じた以降に，$KB_1$と KB_2の動作が異なるようになる．KB_2 ではこの等式は **Orient** の適用により書換え規則$-(x + y) \rightarrow -x + -y$になるが，$KB_2$ では逆向きの書換え規則$-x + -y \rightarrow -(x + y)$ となる．KB_2はこの書換え規則が原因となり発散してしまう．

この例からわかるように，与えられた簡約順序数分のプロセスを発生して完備化の発散を避けることは推論の重複が非常に多く現れ非効率的である．

5.3 複数簡約順序完備化手続き

本節では，複数の簡約順序を同時に扱う完備化手続きにおける重複した推論を避けるための TMS のアイデアに基づく完備化手続きについて述べる．以下，この手続きを MKB と記す．

5.3.1 ノード

任意の簡約順序の集合を $O = \{\succ_1, \ldots, \succ_n\}$，そのインデックスの集合を $I = \{1, \ldots, n\}$ とする．MKB は並行プロセス $\{P_1, \ldots, P_n\}$ をシミュレートする (以後，プロセス P_iを単にプロセス i と呼ぶ)．ここで，プロセス i は簡約順序$\succ_i$のもとでの KB を実行している．

KB は等式集合 $\mathcal{E}$と書換え規則の集合 $\mathcal{R}$ に対する推論規則であった．それに対し，MKB

は推論の重複を避けるため ATMS ノードに基づくノードと呼ばれるデータ構造の集合 $\mathcal{N}$ に対する推論規則として形式化される．この推論規則は TMS の推論の再検知を避ける機能を直接完備化手続きに持たせたことに対応する．以下，MKB が用いるノードについて述べる．

前章では，停止性検証のための TMS について述べたが，そこで用いた TMS ノードは，

$$\gamma_{l\to r} : \langle l \to r, \{\{(f_1, g_1), (f_2, g_2), \cdots\}, \ldots\}\rangle$$

であった．停止性検証では，与えられた書換え規則 $l \to r$ が $l \succ_{lpo} r$ とできるかどうかを確かめ，そのとき得られた環境を保持しておくだけでよかったため 1 つのラベルしか持たせていなかった．完備化では，等式 $l \leftrightarrow r$ は与えられた簡約順序により，どちらに向き付けられるかは予め知ることができない．また，例えば簡約順序として $\succ_i$ $(i = 1, \ldots, 5)$ の 5 つが与えられ，等式 $l \leftrightarrow r$ を向き付けるには次の 3 つの場合が考えられる．

(1) $l \to r$ に向き付けられる場合 ($\succ_1, \succ_4$ で向き付けられるとする)

(2) $r \to l$ に向き付けられる場合 ($\succ_2, \succ_5$ で向き付けられるとする)

(3) (1)，(2) のどちらにも向き付けられない場合 ($\succ_3$ のときとする)

簡約順序を環境とみなし，停止性検証で用いたノードを MKB に採用したとすると，上記の場合に合わせて，

$$\gamma_{l\to r} : \langle l \to r, \{\succ_1, \succ_4\}\rangle$$

$$\gamma_{r\to l} : \langle r \to l, \{\succ_2, \succ_5\}\rangle$$

$$\gamma_{l\leftrightarrow r} : \langle l \leftrightarrow r, \{\succ_3\}\rangle$$

の3つのノードをつくらなければならない．これでは非常に多くのノードを生成することになり有効ではない．

そこで，MKBのためのノードは3つのラベルを持たせ上記の3つのノードを1つのノードで表すことにした．

定義 5.3.1 (ノード) ノードは次の4つ組である．

$$\langle s : t, L_1, L_2, L_3 \rangle$$

ここで，$s : t$ は項の順序対，L_1, L_2, L_3はラベルである．ラベルはインデックス集合 Iの部分集合である．

ノードの各ラベルは，プロセス$i \in L_1$においては$s \to t \in \mathcal{R}$，プロセス$i \in L_2$においては$t \to s \in \mathcal{R}$，プロセス$i \in L_3$においては$s \leftrightarrow t \in \mathcal{E}$ であることを意味する．以後，ノード$\langle s : t, L_1, L_2, L_3 \rangle$ と $\langle t : s, L_2, L_1, L_3 \rangle$ を同一視する．

5.3.2 MKBの推論規則

Backmair が定式化した推論規則 KB に基づきノードの集合 $\mathcal{N}$上の推論規則 MKB を定式化した．MKB の推論規則を図 5.1に示す．$\mathcal{N}$はノードの集合である．各ノード $\langle s : t, L_1, L_2, L_3 \rangle \in \mathcal{N}$ に対し，$i \in L_1$ならば$s \succ_i t$，$i \in L_2$ならば$t \succ_i s$ が成立する．演算子 $\setminus$，$\cup$は集合の差と和を表し，$\cap$は共通集合を表す．この共通集合を求めることによって CKB で生じる推論の重複を避けることが可能となる．

以下，各推論規則について述べる．

- **Delete-1** は **Delete** に対応し，すべての適当なプロセスにおいて自明な等式を削除する．

Delete-1: $\mathcal{N} \cup \{\langle s : s, \ldots, \ldots, \ldots \rangle\} \vdash \mathcal{N}$

Delete-2: $\mathcal{N} \cup \{\langle s : t, \{\}, \{\}, \{\} \rangle\} \vdash \mathcal{N}$

Merge: $\mathcal{N} \cup \left\{ \begin{array}{c} \langle s : t, L_1, L_2, L_3 \rangle \\ \langle s : t, L_1', L_2', L_3' \rangle \end{array} \right\} \vdash \mathcal{N} \cup \{\langle s : t, L_1 \cup L_1', L_2 \cup L_2', L_3 \cup L_3' \rangle\}$

Rewrite-0: $\mathcal{N} \cup \{\langle s : t, L_1 \cup \{i\}, \ldots, L_3 \cup \{i\} \rangle\} \vdash \mathcal{N} \cup \{\langle s : t, L_1 \cup \{i\}, \ldots, L_3 \rangle\}$

Rewrite-1: $\mathcal{N} \cup \{\langle s : t, L_1, L_2, L_3 \rangle\} \vdash \mathcal{N} \cup \left\{ \begin{array}{l} \langle s : t, L_1 \backslash L, L_2, L_3 \backslash L \rangle \\ \langle s : u, L \cap L_1, \{\}, L \cap L_3 \rangle \end{array} \right\}$

if $\langle l : r, L, \ldots, \ldots \rangle \in \mathcal{N}, t \rightarrow_{\{l \rightarrow r\}} u$ and either $t \not\triangleright l$ or $L \cap L_2 = \{\}$

Rewrite-2: $\mathcal{N} \cup \{\langle s : t, L_1, L_2, L_3 \rangle\} \vdash \mathcal{N} \cup \left\{ \begin{array}{l} \langle s : t, L_1 \backslash L, L_2 \backslash L, L_3 \backslash L \rangle \\ \langle s : u, L \cap L_1, \{\}, L \cap (L_2 \cup L_3) \rangle \end{array} \right\}$

if $\langle l : r, L, \ldots, \ldots \rangle \in \mathcal{N},\ t \rightarrow_{\{l \rightarrow r\}} u,\ t \triangleright l,\ L \cap L_2 \neq \{\}$

Orient: $\mathcal{N} \cup \{\langle s : t, L_1, L_2, L_3 \rangle\} \vdash \mathcal{N} \cup \{\langle s : t, L_1 \cup \{i\}, L_2, L_3 \setminus \{i\} \rangle\}$

if $s \succ_i t,\ i \in L_3$

Deduce: $\mathcal{N} \vdash \mathcal{N} \cup \{\langle s : t, \{\}, \{\}, L \cap L' \rangle\}$

if $\langle l : r, L, \ldots, \ldots \rangle, \langle l' : r', L', \ldots, \ldots \rangle \in \mathcal{N},\ s \leftarrow_{\{l \rightarrow r\}} u \rightarrow_{\{l' \rightarrow r'\}} t,$

$L \cap L' \neq \{\}$

図 5.1: 複数簡約順序完備化 (MKB) の推論規則

- **Delete-2** はすべてのプロセスが書換え規則 $s \to t$ あるいは $t \to s$ あるいは等式 $s \leftrightarrow t$ のいずれも持たないことを表すノードを削除する．これはノードのラベルがすべて空であることから，そのノードを保持するプロセスがもはや存在しないことを意味する．
- **Merge** は第 1 スロットが等価なノードを 1 つにまとめノード数を減少させる．この推論規則は ATMS に基づき同じデータを 1 つのノードで表すために用いられる．
- **Rewrite-0** は書換え規則 $s \to t(t \to s)$ がプロセス i における $\mathcal{R}$ 中に存在し，同時に $\mathcal{E}$ 中に等式 $s \leftrightarrow t$ が存在するならば，$\mathcal{E}$からこの等式を削除する．等式 $s \leftrightarrow t$ はすでに簡約順序$\succ_i$のもとで向き付けられているため必要がない．
- **Rewrite-1** は適当な書換え規則と等式が存在するすべてのプロセスにおいて **Simplify** と **Compose** を同時に行う推論規則である．次の 2 つのノードを考える．ただし，書換え規則 $l \to r$により t は u に書換え可能であるとする．また $t \not\triangleright l$または $L \cap L_2 = \{\}$ であるとする．

$$\langle l : r, L, \ldots, \ldots \rangle \quad (1)$$

$$\langle s : t, L_1, L_2, L_3 \rangle \quad (2)$$

$L \cap L_3$に属するプロセス中には書換え規則 $l \to r$と等式 $s \leftrightarrow t$ が存在するので **Simplify** を適用できる．その結果，等式 $s \leftrightarrow t$ が削除され，等式 $s \leftrightarrow u$ が生成される．これは，ノード (2) を

$$\langle s : t, L_1, L_2, L_3 \backslash L \rangle$$

に変更し，ノード

$$\langle s : u, \{\}, \{\}, L \cap L_3 \rangle$$

を生成することに対応する．

同様に，$L \cap L_1$に属するプロセス中には書換え規則 $l \to r$ と $s \to t$ が存在するので

Compose を適用できる．その結果，書換え規則 $s \to t$ が削除され，書換え規則 $s \to u$ が生成される．これはノード (2) を，

$$\langle s : t, L_1 \backslash L, L_2, L_3 \rangle$$

に変更し，ノード

$$\langle s : u, L \cap L_1, \{\}, \{\} \rangle$$

を生成することに対応する．

したがって，**Simplify** と **Compose** を同時に行うことは，ノード (2) を，

$$\langle s : t, L_1 \backslash L, L_2, L_3 \backslash L \rangle$$

に変更し，ノード

$$\langle s : u, L \cap L_1, \{\}, L \cap L_3 \rangle$$

を生成することに対応する．これが **Rewrite-1** である．

- **Rewrite-2** は **Simplify**，**Compose** に加え，**Collapse** を行う．上記と同じ 2 つのノードを用いて説明する．ただし，$t \rhd l$とする．

$L \cap L_2$に属するプロセス中には書換え規則 $l \to r$と $t \to s$ が存在するので **Collapse** を適用できる．その結果，書換え規則 $t \to s$ が削除され，等式 $u \leftrightarrow s$ が生成される．これはノード (2) を，

$$\langle s : t, L_1, L_2 \backslash L, L_3 \rangle$$

に変更し，ノード

$$\langle s : u, \{\}, \{\}, L \cap L_2 \rangle$$

を生成することに対応する．したがって，**Simplify**，**Compose**，**Collapse** を同時に適用することは推論規則 **Rewrite-2** で表現できる．

• **Orient** はあるプロセス i に与えられた簡約順序$\succ_i$のもとで等式 $s \leftrightarrow t$ が $s \succ_i t$ ならば書換え規則 $s \rightarrow t$ に向き付ける．例えば，ノード

$$\langle s:t, L_1, L_2, \{1,2\}\rangle \qquad (3)$$

が与えられたとき，**Orient** は $s \succ_1 t$ ならば，ノード (3) を

$$\langle s:t, L_1 \cup \{1\}, L_2, \{1,2\} \backslash \{1\}\rangle$$

に変更する．

• **Deduce** は適切なプロセスにおいて存在する 2 つのノードから危険対を求め，新たなノードを生成する．例えば，2 つのノード

$$\langle l:r, \{1,2,3\}, \ldots, \ldots\rangle \quad (4)$$

$$\langle l':r', \{1,3\}, \ldots, \ldots\rangle \quad (5)$$

が与えられ，$s \leftarrow_{\{l \rightarrow r\}} u \rightarrow_{\{l' \rightarrow r'\}} t$ であるならば，**Deduce** はノード (4) と (5) 間の重複するプロセス $\{P_1, P_3\}$ で同時に危険対を求めたことになり，ノード

$$\langle s:t, \{\}, \{\}, \{1,3\}\rangle$$

を生成する．

5.4　基本アルゴリズム

NEをノードの集合，NRを相互の推論が済んでいるノードの集合，Iを簡約順序のインデックスの集合とする．文献 [18] に基づいた複数簡約順序完備化アルゴリズムを図 5.2に示す (手続きの記述は基本的に Modula-2 の構文による)．

```
1: procedure mkb(NE, NR, I)
2: begin
3:   while not success-p(NE, NR, I) do
4:     if NE = {} then return(fail)
5:     else
6:       n := choose(NE);
7:       NE := NE\{n} ∪ rewrite({n}, NR);
8:       (* n のラベルが副作用的に変更される *)
9:       if n ≠ ⟨..., {}, {}, {}⟩ then
10:        n := orient(n);
11:        (* n のラベルが副作用的に変更される *)
12:        if n ≠ ⟨..., {}, {}, ...⟩ then
13:          NE := NE ∪ rewrite(NR, {n});
14:          (* NR中のノードのラベルが副作用的に変更される *)
15:          NE := NE ∪ cp(n, NR)
16:        end;
17:        NR := NR ∪ {n}
18:      end
19:    end
20:  end
21:  return(NR)
22: end mkb
```

図 5.2: 複数簡約順序完備化アルゴリズム

関数手続き $mkb(NE, NR, I)$ は複数簡約順序のもとで完備化を行い，完備化に成功した時点のノードの集合 NRを返す．初期値は NEが第 1 ラベルと第 2 ラベルが空で第 3 ラベルに全簡約順序のインデックスを持つノードの集合，NRが空集合，Iが簡約順序のインデックスの集合である．アルゴリズムの基本的な流れを明確に記述するため，便宜上，**Delete-1**，**Delete-2**，**Rewrite-0**，**Merge** の操作を無視して記述していない．これらは適当な時点で適宜実行される．

このアルゴリズムの実行中 NRは常に以下の条件をみたしている．

- すべてのノード $n \in NR$は **Orient** 済み.
- すべての異なる 2 つのノード $m, n \in NR$は互いに **Rewrite-(1,2)** 済み.
- すべての 2 つのノード $m, n \in NR$間で **Deduce** 済み.

アルゴリズム中で用いている述語 *success-p* と関数手続き *choose*，*rewrite*，*orient*，*cp* について述べる．

述語 *success-p*(NE, NR, I) は完備化成功かどうかをチェックするために用いる．*success-p* は I中のあるインデックス i が NE中のすべてのノードのどのラベルにも存在せず，かつ NR中のすべてのノードの第 3 ラベルに存在しないならば真を，さもなくば偽を返す．*success-p* が真を返すならば，簡約順序$\succ_i$のもとで完備化成功を表す．

関数手続き *choose*(*nodes*) はノード集合 *nodes* から 1 つのノードを非決定的に選択する．この選択は公平 (*fair*) であるものとする．つまり，どのノードも有限ステップの間に必ず選択されるということである．

関数手続き *rewrite*(*nodes*1, *nodes*2) は **Rewrite-1**，**Rewrite-2** を繰り返し適用することによってノード集合 *nodes*1 中のノードをノード集合 *nodes*2 により書換える．そして，

書換えによって新たに生成されたノードの集合を返す．このとき，一般に $nodes1$ 中のノードのラベルが副作用的に変更される ($nodes1$ 中のノードは図 5.1の各推論規則のノード $\langle s : t, \ldots, \ldots, \ldots \rangle$ に対応させる).

関数手続き $orient(node)$ は **Orient** を繰り返し適用し，ノード $node$ の第 3 ラベル中の簡約順序によって等式を向き付け，副作用的にノードのラベルを更新する．

関数手続き $cp(node, nodes)$ は **Deduce** の適用によってノード $node$ とノード集合 $nodes$ 中のノード間から得られるすべての危険対を求め，新たなノードを生成し，得られたノードの集合を返す ($node$ と $node$ 間の危険対も含む).

5.5　複数簡約順序完備化手続きの動作

節 5.1.1で用いた例を再度用いて述べる．

$$(x + y) + z \leftrightarrow x + (y + z) \quad (1)$$

$$f(x) + f(y) \leftrightarrow f(x + y) \quad (2)$$

簡約順序として辞書式経路順序を用いる．先行順序を$\succ^1 = \{(+, f)\}$，$\succ^2 = \{(f, +)\}$ とし，対応する辞書式経路順序をそれぞれ$\succ^1_{lpo}$，$\succ^2_{lpo}$とする．初期値は

$$NE = \begin{cases} \langle (x + y) + z : x + (y + z), \{\,\}, \{\,\}, \{1, 2\} \rangle & (3) \\ \langle f(x) + f(y) : f(x + y), \{\,\}, \{\,\}, \{1, 2\} \rangle & (4) \end{cases}$$

$NR = \{\,\}$，$O = \{\succ^1_{lpo}, \succ^2_{lpo}\}$，$I = \{1, 2\}$ である．

ノード (3) を選択し，$orient$ により向き付けると

$$\langle (x + y) + z : x + (y + z), \{1, 2\}, \{\,\}, \{\,\} \rangle \quad (3')$$

となる．次にノード (4) を選択し，*orient* により向き付けると

$$\langle f(x)+f(y):f(x+y),\{1\},\{2\},\{\ \}\rangle \quad (4')$$

となる．ノード (3′)，(4′) 間から *cp* により新たなノード (5),(6) を得る．

$$NE=\begin{cases}\langle f(x+y)+z:f(x)+f(y+z),\{\ \},\{\ \},\{1\}\rangle & (5)\\ \langle f(x+(y+z)):f(x+y)+f(z),\{\ \},\{\ \},\{2\}\rangle & (6)\end{cases}$$

ここで，ノード (6) を選択したとすると，*rewrite* によりノード (6) は自明な等式を持つノードとなり削除される．この時点で，

$$NR=\begin{cases}\langle (x+y)+z:x+(y+z),\{1,2\},\{\ \},\{\ \}\rangle & (3')\\ \langle f(x)+f(y):f(x+y),\{1\},\{2\},\{\ \}\rangle & (4')\end{cases}$$

$$NE=\Big\{\ \langle f(x+y)+z:f(x)+f(y+z),\{\ \},\{\ \},\{1\}\rangle \quad (5)$$

このとき *success-p* は真を返し，簡約順序 $\succ^2_{lpo}$ のもとで完備化は成功する．

5.6 複数簡約順序完備化手続きの実験結果

実験環境として Sun SPARC Station IPX(主記憶 32Mbyte,28.5MIPS) 上に Lucid Common Lisp を用いて，通常の完備化手続き (*kb*) と複数簡約順序完備化手続き (*mkb*) 及び Lucid Common Lisp のマルチタスク機能を用いて通常の完備化手続きを並行実行する並行完備化手続き (*ckb*) を実現し，表 5.1，5.2 に示す実験結果を得た．以下，実験結果について述べる．

行った実験は以下の6本の等式の完備化である.

$$1 * x \leftrightarrow x$$

$$f(x) * x \leftrightarrow 1$$

$$x * g(x) \leftrightarrow 1$$

$$(x * y) * z \leftrightarrow x * (y * z)$$

$$1 + x \leftrightarrow s(x)$$

$$(x + y) + z \leftrightarrow x + (y + z)$$

関数記号は$\mathcal{F} = \{1, *, f, g, s, +\}$の6つあり, $\mathcal{F}$上の先行順序を全順序$\succ_i$ $(i = 1, \ldots, 720)$とし(全順序と考えた場合$6! = 720$通り), それぞれに対応する辞書式経路順序$\succ^i_{lpo}$を簡約順序として用いた.

表5.1中の$P_n(n = 1, \ldots, 6)$は与えた簡約順序の集合を表している. この集合は, 先行順序の単調性を用いて定めた. P_1は$f \succ g \succ * \succ s \succ + \succ 1$から得られる1つの簡約順序である. P_2は$f \succ g \succ * \succ s \succ +$を拡張して得られる6通り, P_3は$f \succ g \succ * \succ s$の拡張による30通り, P_4は$f \succ g \succ *$の拡張による120通り, P_5は$f \succ g$の拡張による360通りの先行順序に対応する. P_6は720通りのすべての先行順序に対応する簡約順序の集合である. 表5.1中の実行時間の単位は秒である. また丸カッコで囲まれた数値は生成された危険対の数, 角カッコで囲まれた数値は生成されたノードの数である(危険対の数も含む).

図5.3は表5.1の*mkb*と*ckb*の実行時間と簡約順序数の関係をプロットしたグラフである. 図5.4は環境あたり*mkb*と*ckb*が生成したノード数と危険対数の関係をプロットしたグラフである(□は*ckb*,◁は*mkb*). 図5.3, 図5.4からわかるように*ckb*では立ち上がりがはやく, 重複した推論を行っていることから非常に多くの危険対を生成している. また, *ckb*はP_5以降では本実行環境では実行することができなかった. それに対し, *mkb*では重複した

推論を避けることによって非常に効率を改善し，ckb で実行できなかった P_5以降に対しても解を得ることができた.

表 5.2は 720 通りの簡約順序の中からランダムに簡約順序を選びだし (少なくとも 1 つ完備化に成功する簡約順序を含む)，mkb と ckb の実行時間と 1 簡約順序あたりの実行時間を比較したものであり,．図 5.5は表 5.2の簡約順序数と実行時間をプロットしたグラフである．表 5.2からわかることは ckb では与えられた簡約順序数が増加するにしたがって 1 簡約順序あたりの実行時間が大きく変化しないが，mkb では重複した推論を避けることによって，1 簡約順序あたりの実行時間が減少する傾向にある．図 5.5から mkb は ckb の 2 倍程度効率を改善していることがわかる.

表 5.1: 複数簡約順序完備化の実験結果 1

インプリメント	P_1	P_2	P_3	P_4	P_5	P_6
kb	4.10 (178)					
ckb	4.21 (178)	21.02 (793)	105.05 (2494)	488.31 (9257)		
mkb	4.81 [311]	8.63 [610]	45.95 [1573]	65.89 [1245]	353.60 [1893]	2444.53 [3955]

表 5.2: 複数簡約順序完備化手続きの実験結果 2

	mkb		*ckb*	
簡約順序数 O	time(sec)	CPU/O	time(sec)	CPU/O
1	4.81	4.81	4.21	4.21
10	38.67	3.87	35.48	3.55
20	49.25	2.46	79.79	3.89
30	51.92	1.73	116.05	3.87
40	98.38	2.45	167.98	4.12
50	79.56	1.59	190.42	3.81
60	146.41	2.44	254.34	4.24
70	165.42	2.36	300.55	4.29
80	164.75	2.06	394.24	4.93
180	318.60	1.71		
320	545.69	1.91		
500	952.81	1.97		

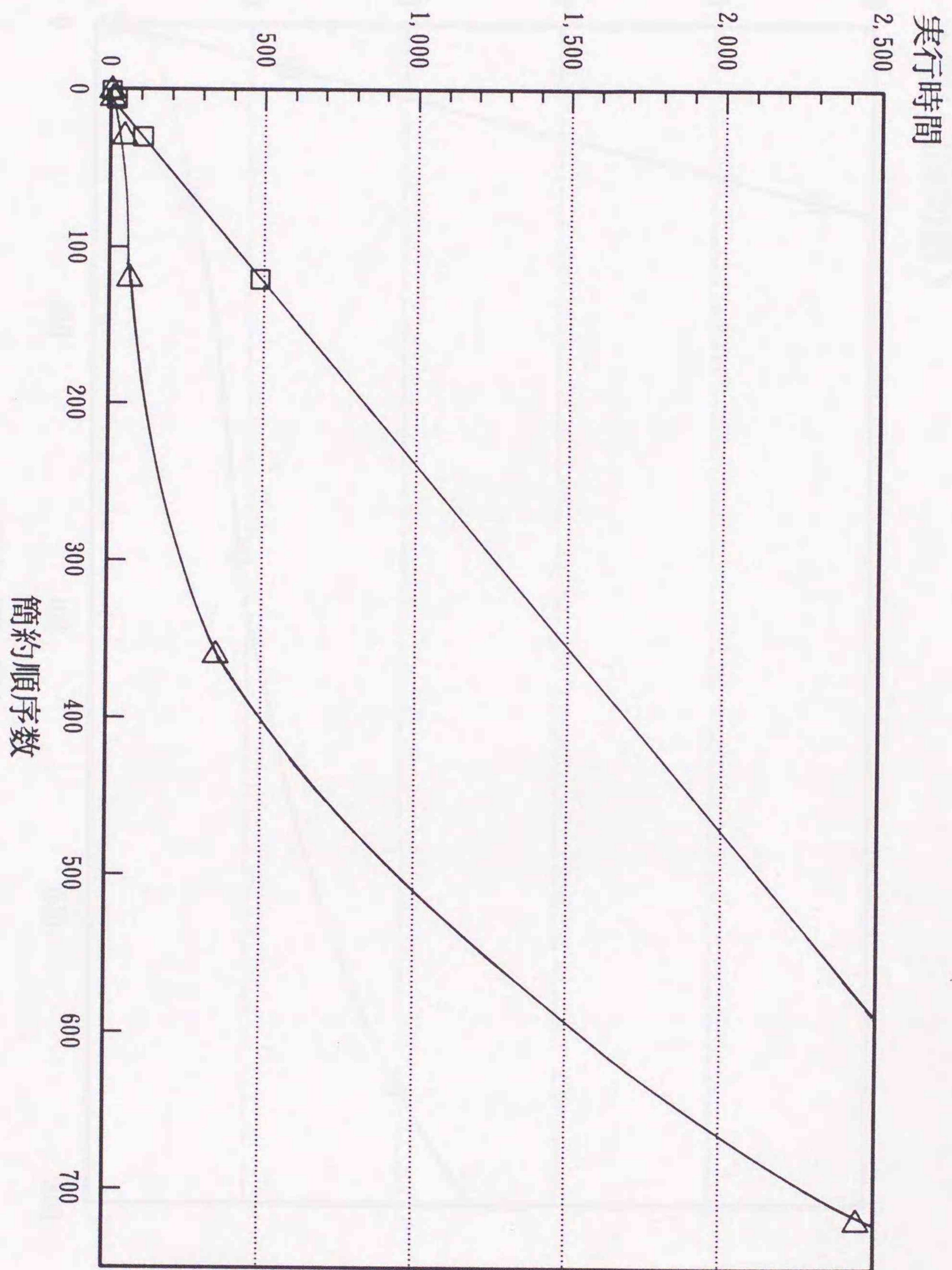


図 5.3: 実行時間と簡約順序数 (表 5.1)

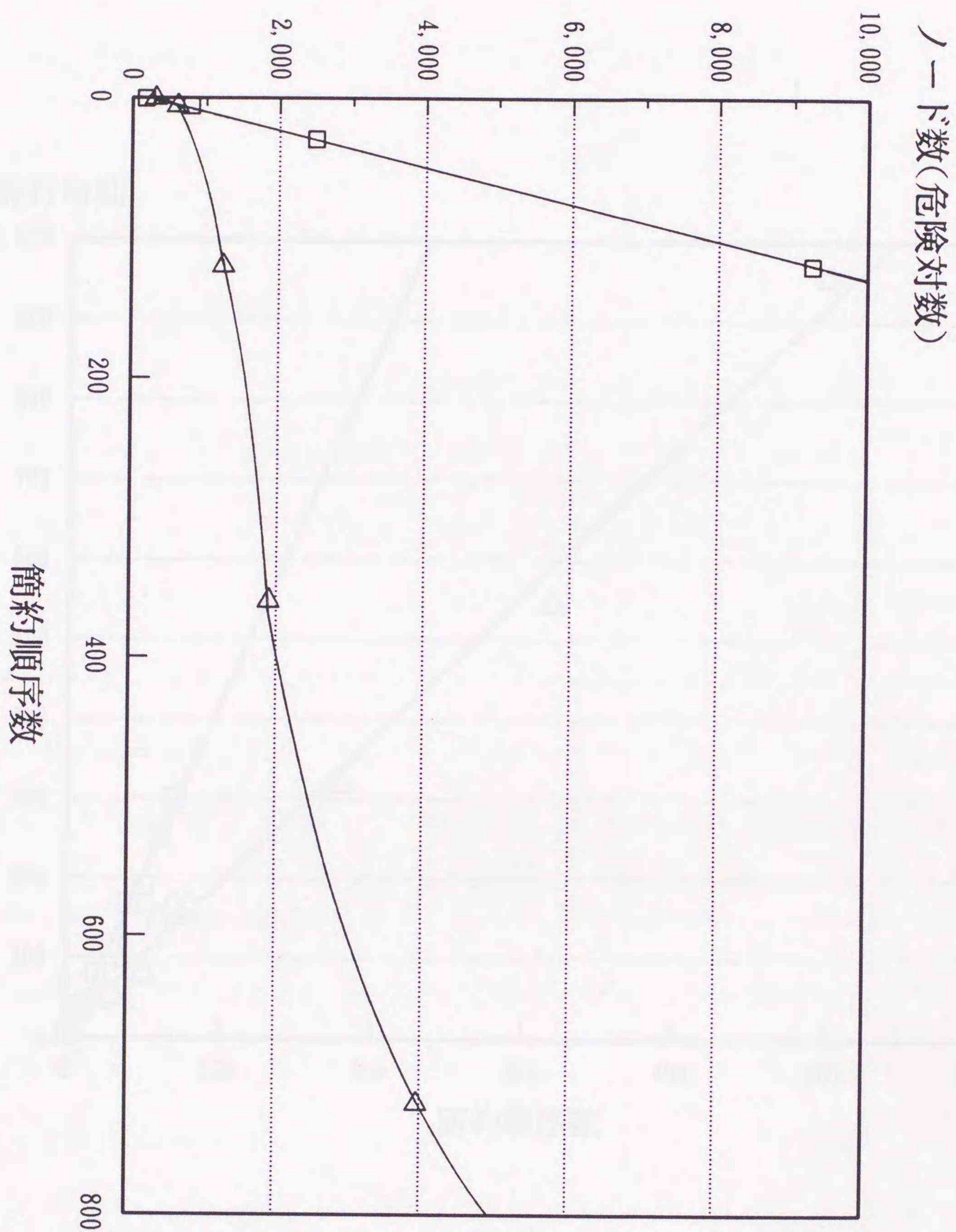


図 5.4: 簡約順序数とノード数 (危険対数)

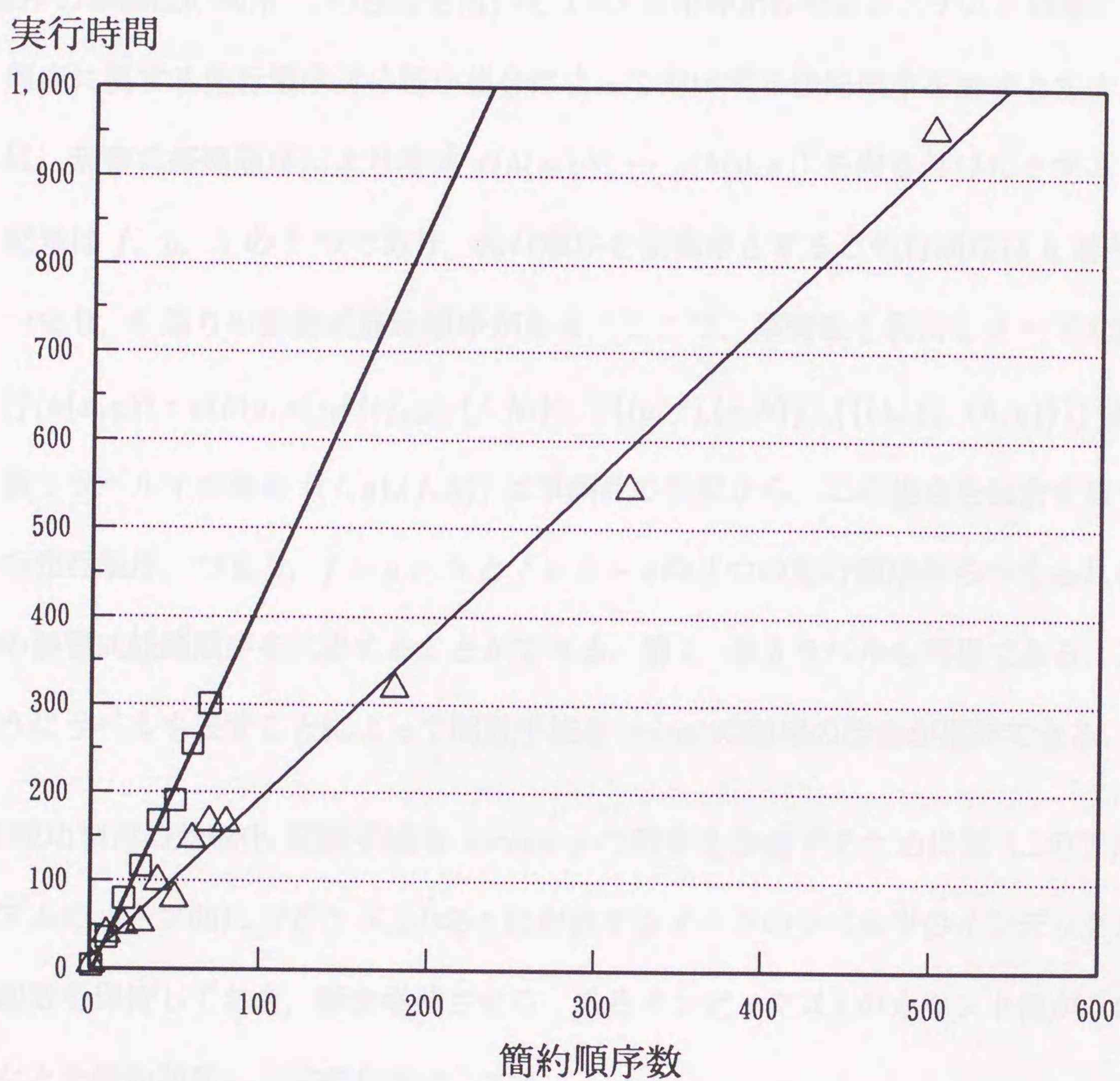


図 5.5: 実行時間と簡約順序数 (表 5.2)

5.7 議　論

本節では，mkb の効率をさらに向上させ得る手法について考察する．

経路順序の単調性の利用 この性質を用いて TMS 利用停止性検証システムと同様に，先行順序に属する先行順序対の極小集合によって対応する経路順序を表すとする．例えば，辞書式経路順序により等式 $f(h(x,y)) \leftrightarrow g(h(y,x))$ を向き付けたとする．関数記号は f，g，h の 3 つであり，先行順序を全順序とすると先行順序は 6 通りある．つまり，6 通りの辞書式経路順序がある．ここで，単調性を利用しノードを表すと $\langle f(h(x,y)) : g(h(y,x)),\{\{(f,g),(f,h)\}\}, \{\{(g,f),(g,h)\}\},\{\{(h,f),(h,g)\}\}\rangle$ となり，第 1 ラベル中の集合 $\{(f,g),(f,h)\}$ は単調性の性質から，この集合を包含するすべての先行順序，つまり，$f \succ g \succ h$ と $f \succ h \succ g$ の 2 つの先行順序からつくられる 2 つの辞書式経路順序を代表することができる．第 2，第 3 ラベルも同様である．このようにラベルを表すことによって関数手続き $orient$ の効率の改善が期待できる．

完備化成功判定の効率化 関数手続き $success\text{-}p$ の効率を改善するために図 5.2のアルゴリズムのループ間に NE と NR の各々に存在するノードのラベル中のインデックスの生起数を保持しておき，漸次増減させる．あるインデックス i のカウント値が 0 になったとき簡約順序 $\succ_i$ で完備化成功とする．

ヒューリスティクスの利用 上記のカウント値を利用し，図 5.2のステップ 6:の $choose$ によるノードの選択においてカウント値の小さい (完備化成功に有望な) インデックスを持つノードを選択する．この選択の仕方によって生成される危険対を少なくすることができる．

5.8　まとめ

本章では，完備化手続きの問題点を明確にし，その問題点から生じる並行完備化手続きの計算効率上の問題点である重複した推論を避ける ATMS のデータ構造のアイデアに基づく複数簡約順序完備化について述べた．さらに，計算機実験を通してその有効性を示した．

第 6 章

結　論

6.1　各章のまとめ

6.1.1　第 1 章

第 1 章では，本研究の背景と本論文が扱う問題領域である項書換えシステムの停止性と合流性，完備化手続き及び推論の高速化技法の 1 つである真偽維持システムについて簡単に紹介し，本研究の目的を述べた．

6.1.2　第 2 章

第 2 章では，本論の準備として，抽象書換えシステムを通して項書換えシステムの諸性質を述べた．また，項書換えシステムの停止性検証法と完備化手続きについて紹介した．

6.1.3　第 3 章

第 3 章では，真偽維持システムについて述べた．特に真偽維持システムとしてよく知られている ATMS について紹介した．

6.1.4　第 4 章

第 4 章では，まず，従来の停止性検証手続きの計算効率上の問題点を明らかにした．その問題点はバックトラック法を利用することによる，

(1) 無駄なバックトラック

(2) 推論の再検知

(3) 矛盾の再検知

であった．次に，これらの問題点を解決するために設計・開発した真偽維持システム及びそれを用いた検証手続きについて述べた．その内容は，

(1) TMS ノード

(2) 正当化の計算方法

(3) *nogood* の計算方法

(4) TMS と推論手続き間の通信プロトコル

である．また例題を通して本停止性検証手続きがいかにして，従来の実現手法の問題点を避けているかを示した．さらに，計算機実験を行いその有効性を論じた．

6.1.5　第 5 章

第 5 章では，完備化手続きの問題点を明らかにした．その問題点は，

(1) 利用者に対する簡約順序の要求

(2) 不適当な簡約順序による手続きの発散

(3) 解が存在するにもかかわらず手続きが失敗

であった．さらに問題点 (2) に対する解決のアプローチの 1 つである並行完備化手続きの問題点を述べ，その問題点を解決するために，真偽維持システムのデータ構造に基づくノードと呼ばれるデータを扱い推論の重複を避ける複数簡約順序完備化の設計について述べた．その内容は

(1) ノードの設計

(2) 重複した推論を避けるノードの集合上の推論規則 MKB

(3) 複数簡約順序完備化アルゴリズム

である．また計算機実験を通してその有効性について述べた．

6.2　まとめ

本論文では，真偽維持システムを用いた項書換えシステムの推論手続きに関する研究として，以下のことを行った．

(1) 従来の停止性検証手続きの問題点を明確にし，その問題点を解決するための真偽維持システムの設計と開発を行い，それを用いた停止性検証システムを開発した．

(2) 完備化の発散を避ける 1 つのアプローチである並行完備化手続きの問題点を明確にし，その問題点を解決するための真偽維持システムの設計と開発を行い，それを用いた複数簡約順序完備化システムを開発した.

(3) 計算機実験を行い本論文で提案した TMS 利用停止性検証システムと複数簡約順序完備化システムの有効性を確認した.

本研究により開発した 2 つのシステムは従来の実現手法と比較して，非常に効率的であるが以下の検討課題が残されていると考えられる.

- 4.6で述べたより強力な経路順序への拡張.
- より幅広い完備化を行うために，AC(*associative-commutative*) 完備化の導入.
- 完備化手続きの帰納的定理証明への応用.

謝　辞

本研究を進めるにあたり，北海道大学工学部情報工学科 大内　東教授には深い議論を通して御指導いただくとともに多くの御援助，励ましをいただいた．深甚の謝意を表します．本研究に対して有意義な御示唆と詳細な議論をいただいた，北海道大学工学部情報工学科 新保　勝教授，宮本　衛市教授，北海道大学工学部精密工学科 嘉数　侑昇教授に深く感謝申し上げます．北海道大学工学部情報工学科 栗原 正仁助教授には著者が北海道大学工学部情報工学科研究生として所属して以来，終始，本研究の理論的基礎を懇切，丁寧に御教授していただいた．心から感謝申し上げます．北海道工業大学経営工学科 北守　一隆助教授には研究の機会を与えていただいた．感謝申し上げます．先輩，友人そして先生として御世話になった北海道大学工学部情報工学科 遠藤 聡志助手に深く感謝いたします．また，毎日の生活を楽しいものとしていただいたシステム工学講座の皆様に感謝します．

最後に，長い間学生生活を送らせてくれた両親に感謝します．

参考文献

[1] 二木，外山. 項書き換え型計算モデルとその応用. 情報処理, 24(2):133–146, 1983.

[2] Dershowitz, N. and Jouannaud, J.-P. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol.B*, pages 243–320, Elsevier Science Publishers B. V., 1990.

[3] 富樫. 関数型プログラミングにおける計算モデル. 情報処理, 29(8):817–828, 1988.

[4] Toyama, Y. How to prove equivalence of term rewriting systems without induction. *Theo. Comp. Sci.*, 90:396–390, 1991.

[5] Kurihara, M. and Ohuchi, A. Modularity of simple termination of term rewriting systems. 情報処理, 31(5):562–571, 1990.

[6] 稲垣，坂部. 抽象データタイプの代数的仕様記述の基礎 (1). 情報処理, 25(1):47–53, 1984.

[7] Hsiang, J. Refutational theorem proving using term-rewriting systems. *Artif. Intell.*, 25:255–300, 1985.

[8] Dershowitz, N. Completion and its applications. In H. Aït-Kaci and M. Nivar, editors, *Resolution of equations in algebraic structures, Vol.II*, pages 31–85, Academic Press, 1989.

[9] Detlefs, D. and Forgaad, R. A procedure for automatically proving the termiation of a set of rewrite rules. In *Proc.of 1st. Int. Conf. on Rewriting Techniques and Applications,* LNCS 202, pages 255–270, Springer Verlag, 1985.

[10] Knuth, D. E. and Bendix, P. B. Simple word problems in universal algebras. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297, Pergamon Press, 1970.

[11] Kapur, D. and Sivakumar, G. Architecture of and experiments with RRL, a rewrite rule laboratory. In *Proc. NSF Workshop on the Rewrite Rule Laboratory, RenselIaervill, NY*, pages 33–56, 1984.

[12] Lescanne, P. Computer experiments with the REVE term rewriting system generator. In *Proc. 10th. ACM Sympo. on Principles of Programming Languages, Austin, TX*, pages 99–108, 1983.

[13] Doyle, J. A truth maintenance system. *Artif. Intell.*, 12:231–272, 1979.

[14] de Kleer, J. An Assumption based TMS. *Artif. Intell.*, 28:127–163, 1986.

[15] de Kleer, J. Extending the ATMS. *Artif. Intell.*, 28:163–196, 1986.

[16] de Kleer, J. Problem solving with the ATMS. *Artif. Intell.*, 28:197–224, 1986.

[17] Dershowitz, N. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[18] Huet, G. A completion proof of correctness of the Knuth–Bendix completion algorithm. *J. Comput. Syst. Sci.*, 23(1):11–21, 1981.

[19] Huet, G. Confluent reductions:abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[20] Kelleher, G. and Smith, B. M. A Brief introduction to reason maintenance systems. In B. M. Smith and G. Kelleher, editors, *Reason Maintenance Systems and Their Application*, pages 4–20, Ellis Horwood, 1988.

[21] Wolfram, D. A. Forward checking and intelligent backtracking. *Inf. Process. Lett.*, 32:85–87, 1989.

[22] 中川，北谷，栗原，加地. 論理回路の故障診断における自動推論システムの実験とヒューリスティクス. 電学論 *C*, 109-C(10):731–738, 1989.

[23] Rusinowitch, M. Path of subterms ordering and recursive decomposition ordering revisited. *J.Symbolic Computation*, 3:117–131, 1987.

[24] Snyder, W. *Computing the lexicographic path ordering.* Technical Report, Boston University, Boston, MA, 1990.

[25] Kurihara, M. and Ohuchi, A. Modularity of simple termination of term rewriting systems with shared constructors. 情報処理学会研究会報告, 90(76, SF-36–1):1–10, 1990.

[26] Steinbach, J. Comparison of simplification orderings. In *Proc. of 3rd. Conf. on Rewriting Techniques and Applications,* LNCS 355, pages 434–448, Springer Verlag, 1989.

[27] Hermann, M. *Vademecum of Divergent Term Rewriting Systems.* Technical Report 88–R–022, Centre de Recherche en Informatique de Nancy, 1988.

[28] Bachmair, L., Dershowitz, N., and Plaisted, D. A. Completion without failure. In H. Aït-Kaci and M. Nivar, editors, *Resolution of equations in algebraic structures, Vol.II,* pages 1–30, Academic Press, 1989.

付　録A

プログラムリスト

```
;;;-*- Mode: Lisp; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;;
;;;This file contains the termination verifier which verifies termination of
;;;term rewriting systems.
;;;This termination verifier find a partial order on a set of function symbols,
;;;which constructs term. The partial order can be used by lexicographic
;;;path ordering.

;;;
;;;Define count-variables.
;;;
(defvar *count-nodes* 0)
(defvar *intell-cut* 0)
(defvar *nogood-cut* 0)
(defvar *orient-cut* 0)

;;;
;;;Lexicographic path ordering.
;;;LPO> has a term rewriting system as its argument and returns pathial order
;;;on the set of function symbols which contstructs rewriting rules.
;;;
(defun tms-lpo> (lhs-term rhs-term and-problems rules rule temp-prec
                temp-nogood prec)
  (cond ((equal lhs-term rhs-term) (values temp-nogood 'fail))
        ((and (var-p lhs-term) (var-p rhs-term)) (values temp-nogood 'fail))
        ((and (not (var-p lhs-term)) (not (var-p rhs-term)))
         (let ((f (operator lhs-term))
               (g (operator rhs-term)))
           (cond ((eq f g) (tms-solve-lex-and-lpo>
                            lhs-term (args lhs-term) (args rhs-term)
                            and-problems rules rule temp-prec temp-nogood
                            prec))
                 ((operator> f g prec) (tms-solve-and-and-lpo>
                                        lhs-term (args rhs-term) and-problems
                                        rules rule
                                        (adjoin (list f g) temp-prec
                                                :test #'equal)
                                        (adjoin (list f g) temp-nogood
                                                :test #'equal)
                                        prec))
                 ((operator> g f prec) (tms-solve-or-and-lpo>=
                                        (args lhs-term) rhs-term and-problems
                                        rules rule temp-prec
                                        (adjoin (list g f) temp-nogood
                                                :test #'equal)
                                        prec))
                 (t (let ((extended-prec1 (add-prec f g prec)))
                      (if (not (and (nogoodp extended-prec1)
                                    (incf *nogood-cut*)))
                          (and (incf *count-nodes*)
                               (multiple-value-bind (newprec result)
                                (tms-solve-and-and-lpo> lhs-term
                                 (args rhs-term) and-problems rules rule
                                 (adjoin (list f g) temp-prec :test #'equal)
                                 (adjoin (list f g) temp-nogood :test #'equal)
                                 extended-prec1)
                                (case result
                                 (fail
                                  (if (intersection (generated-label
                                                     extended-prec1 prec)
                                                    newprec :test #'equal)
                                      (let ((extended-prec2
                                             (add-prec g f prec)))
                                        (if (not (and (nogoodp extended-prec2)
                                                      (incf *nogood-cut*)))
                                            (multiple-value-bind
                                             (newprec1 result1)
                                             (tms-solve-or-and-lpo>=
                                              (args lhs-term) rhs-term
                                              and-problems rules rule temp-prec
                                              (adjoin (list g f) temp-nogood
                                                      :test #'equal)
                                              extended-prec2)
                                             (case result1
                                              (fail
                                               (if (intersection
```

```
                                                  (generated-label
                                                   extended-prec2 prec)
                                                  newprec1 :test #'equal)
                                                 (values (hyperresolve
                                                          newprec newprec1)
                                                         'fail)
                                               (and (incf *intell-cut*)
                                                (values newprec1 'fail))))
                                            (success
                                             (values newprec1 'success))))
                                         (values newprec 'fail)))
                                       (and (incf *intell-cut*)
                                           (values newprec 'fail))))
                                  (success
                                   (values newprec 'success)))))
                         (let ((extended-prec3 (add-prec g f prec)))
                           (if (not (and (nogoodp extended-prec3)
                                         (incf *nogood-cut*)))
                               (and (incf *count-nodes*)
                                (tms-solve-or-and-lpo>=
                                 (args lhs-term) rhs-term and-problems rules
                                 rule temp-prec (adjoin (list g f) temp-nogood
                                                        :test #'equal)
                                 extended-prec3))
                             (values temp-nogood 'fail))))))))
        ((and (not (var-p lhs-term)) (var-p rhs-term))
         (tms-solve-or-and-lpo>=
          (args lhs-term) rhs-term and-problems rules rule temp-prec
          temp-nogood prec))
        (t (values temp-nogood 'fail))))

;;;
;;;The following definition is auxilialy of ATMS-LPO>.
;;;
(defun tms-solve-and-and-lpo> (lhs-term rhs-term-args and-problems
                               rules rule temp-prec temp-nogood prec)
  (if (endp rhs-term-args)
      (cond ((endp and-problems) (record-inference rule temp-prec)
             (termination-aux rules prec))
            (t (tms-lpo> (first (first and-problems))
                         (second (first and-problems)) (rest and-problems)
                         rules rule temp-prec temp-nogood prec)))
    (tms-lpo> lhs-term (first rhs-term-args) (extend-and-problems
                                              lhs-term
                                              (rest rhs-term-args)
                                              and-problems)
              rules rule temp-prec temp-nogood prec)))

(defun tms-solve-or-and-lpo>= (lhs-term-args rhs-term and-problems rules
                               rule temp-prec temp-nogood prec &optional
                               (halfgood))
  (cond ((and (endp lhs-term-args) (not (endp halfgood))) (values halfgood
                                                                  'fail))
        ((endp lhs-term-args) (values temp-nogood 'fail))
        ((equal (first lhs-term-args) rhs-term)
         (cond ((endp and-problems) (record-inference rule temp-prec)
                (termination-aux rules prec))
               (t (tms-lpo> (first (first and-problems))
                            (second (first and-problems)) (rest and-problems)
                            rules rule temp-prec temp-nogood prec))))
        (t (multiple-value-bind (newprec result)
            (tms-lpo> (first lhs-term-args) rhs-term and-problems rules rule
                      temp-prec temp-nogood prec)
            (case result
              (success (values newprec 'success))
              (fail (tms-solve-or-and-lpo>= (rest lhs-term-args) rhs-term
                                            and-problems rules rule temp-prec
                                            temp-nogood prec
                                            (union newprec halfgood :test
                                                   #'equal)))))))))

(defun tms-solve-lex-and-lpo> (lhs-term lhs-term-args rhs-term-args
                               and-problems rules rule temp-prec temp-nogood
                               prec)
  (cond ((endp lhs-term-args) (values temp-nogood 'fail))
        ((equal (first lhs-term-args) (first rhs-term-args))
```

```
          (tms-solve-lex-and-lpo> lhs-term (rest lhs-term-args)
                                  (rest rhs-term-args) and-problems rules
                                  rule temp-prec temp-nogood prec))
         (t (tms-lpo> (first lhs-term-args) (first rhs-term-args)
                      (extend-and-problems lhs-term (rest rhs-term-args)
                                           and-problems)
                      rules rule temp-prec temp-nogood prec))))

(defun generated-label (extended-prec prec)
  (set-difference extended-prec prec :test #'equal))
;;;
;;;Top level function.
;;;
(defun tms-termination (rules &optional (prec nil))
  (init-tms-database)
  (init-count-variables)
  (termination-aux rules prec))

(defun termination-aux (rules prec)
  (cond ((null rules)
         (format t "~%call record-inference = ~D ~%" *record-inference*)
         (format t "call nogoodp = ~D ~%" *nogoodp*)
         (format t "call orientp = ~D ~%" *orientp*)
         (format t "cut1 (by intelligente backtracking) = ~D ~%" *intell-cut*)
         (format t "cut2 (by nogoodp) = ~D ~%" *nogood-cut*)
         (format t "cut3 (by orientp) = ~D ~%" *orient-cut*)
         (format t "number of nodes = ~D ~%" *count-nodes*)
         (values prec 'success))
        ((orientp (first rules) prec) (incf *orient-cut*)
         (termination-aux (rest rules) prec))
        (t (multiple-value-bind (newprec result)
               (tms-lpo> (lhs-term (first rules))
                         (rhs-term (first rules))
                         nil (rest rules) (first rules) nil nil prec)
             (case result
               (success (values newprec 'success))
               (fail (record-inference 'BOTTOM newprec)
                     (values newprec 'fail)))))))

(defun init-count-variables ()
  (setf *count-nodes* 0)
  (setf *intell-cut* 0)
  (setf *nogood-cut* 0)
  (setf *orient-cut* 0)
  (setf *record-inference* 0)
  (setf *nogoodp* 0)
  (setf *orientp* 0))
```

```
;;;-*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;;
;;;This file contains procedure TMS for termination verifier of term rewriting
;;;systems. This TMS has extended for termination verifier.
;;;

;;;
;;;Define database.
;;;
(defvar *TMS-DATABASE* nil)
(defvar *NOGOOD-DATABASE* nil)

;;;
;;;Define count-variables.
;;;
(defvar *record-inference* 0)
(defvar *orientp* 0)
(defvar *nogoodp* 0)

;;;
;;;Key of *TMS-DATABASE* is a rule of TRS,and datum is tms-label of the rule.
;;;
(defun init-tms-database ()
  (setf *TMS-DATABASE* (make-hash-table :test #'eq))
  (setf *NOGOOD-DATABASE* nil))

;;;
;;;RECORD-INFERENCE records problem-solver's inferences ever made.
;;;If first argument is not BOTTOM,second argument is recorded for
;;;*TMS-DATABASE* otherwise it is recorded for *NOGOOD-DATABASE* as nogood.
;;;
(defun record-inference (rule justification)
  (incf *record-inference*)
  (if (eq rule 'BOTTOM) (record-nogood justification)
    (let ((tms-label (gethash rule *TMS-DATABASE*)))
      (if (endp justification) (setf (gethash rule *TMS-DATABASE*)
                                     (list justification))
        (setf (gethash rule *TMS-DATABASE*)
              (cons justification tms-label))))))

;;;
;;;ORIENTP queries whether first argument is hold in *TMS-DATABASE*.
;;;If some environment of first argument's label is subset of current
;;;precedence (ie,current environment),ORIENTP returns NON-NIL otherwise NIL.
;;;
(defun orientp (rule current-precedence)
  (incf *orientp*)
  (let ((rule-label (gethash rule *TMS-DATABASE*)))
    (some #'(lambda (env)
              (subsetp env current-precedence :test #'equal))
          rule-label)))

;;;
;;;NOGOODP has one argument,which is extend order.
;;;If transitive closure of union of current precedence and added order
;;;is a superset of any nogood, NOGOODP returns T otherwise NIL.
;;;
(defun nogoodp (extended-precedence)
  (incf *nogoodp*)
  (some #'(lambda (env)
            (subsetp env extended-precedence :test #'equal))
        *NOGOOD-DATABASE*))

;;;
;;;RECORD-NOGOOD has two arguments. Computing of nogood uses negative
;;;hyperresolution. These nogoods are recorded for *NOGOOD-DATABASE*.
;;;For example,
;;;    choose {f>g,g>f}        choose {f>g,g>f}
;;;    nogood {f>g,h>i}        nogood {f>g,h>i}
;;;    nogood {g>f,h>i}        nogood {g>f,j>k}
;;;new nogood {h>i}        new nogood {h>i,j>k},{f>g,h>i},{g>f,j>k}
;;;
(defun record-nogood (justification)
  (if (endp *NOGOOD-DATABASE*) (setf *NOGOOD-DATABASE* (list justification))
    (setf *NOGOOD-DATABASE* (record-nogood-aux1 justification
                                                *NOGOOD-DATABASE*))))
```

```
(defun record-nogood-aux1 (justification nogoods)
  (let ((new-nogoods (get-all-new-nogood justification nogoods)))
    (minimal-union (list justification) (minimal-union new-nogoods nogoods))))

(defun reverse-member (nogood)
  (cond ((null nogood) nil)
        ((find (reverse (first nogood)) (rest nogood) :test #'equal) t)
        (t (reverse-member (rest nogood)))))

(defun hyperresolve (old-nogood new-nogood)
  (dolist (nogood-element old-nogood)
          (when (find (reverse nogood-element) new-nogood :test #'equal)
                (let ((nogood (union (remove nogood-element old-nogood
                                                         :test #'equal)
                                     (remove (reverse nogood-element)
                                                     new-nogood :test #'equal)
                                     :test #'equal)))
                  (if (reverse-member nogood) (return nil)
                      (return nogood))))))

(defun get-all-new-nogood (justification nogoods)
  (let ((new-nogoods nil))
    (dolist (nogood nogoods new-nogoods)
            (let ((new-nogood (hyperresolve nogood justification)))
              (when (not (endp new-nogood))
                    (push new-nogood new-nogoods))))))

(defun minimal-union (new-nogoods nogoods)
  (if (endp new-nogoods) nogoods
    (minimal-union (rest new-nogoods) (minimal-cons (first new-nogoods)
                                                    nogoods))))

(defun minimal-cons (nogood nogoods)
  (cond ((null nogoods) (list nogood))
        ((subsetp nogood (first nogoods) :test #'equal)
         (minimal-cons nogood (rest nogoods)))
        ((subsetp (first nogoods) nogood :test #'equal) nogoods)
        (t (cons (first nogoods) (minimal-cons nogood (rest nogoods))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-
;;;
;;; This file contains the function match-term.
;;;
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))

(defun match-term (pat dat &optional (subst-so-far))
  (cond ((var-p pat)
         (var-match pat dat subst-so-far))
        ((atom pat)
         (and (eq pat dat) (list subst-so-far)) )
        ((atom dat) nil)
        (t (let ((subst-list (match-term (first pat)
                                         (first dat) subst-so-far) ))
             (if subst-list
                 (match-term (rest pat) (rest dat) (first subst-list))
                 nil)) )) )

(defun var-match (var dat subst &aux (binding (assoc var subst)))
  (if binding
      (const-match (second binding) dat subst)
      (list (cons (list var dat) subst))) )

(defun const-match (dat1 dat2 subst)
  (if (equal dat1 dat2)
      (list subst)
      nil))

(defun matching-position-and-subst (term1 term2 &optional position)
  (cond ((null term1) nil)
        ((var-p term1) nil)
        (t (let ((substitution (match-term term2 term1)))
             (cond ((and (atom term1) substitution)
                    (list (cons (reverse position) substitution)))
                   ((atom term1) nil)
                   (t (if substitution
                          (cons (cons (reverse position) substitution)
                                (matching-position-and-subst-aux
                                  (rest term1)
                                  term2 (cons '1 position)))
                          (matching-position-and-subst-aux
                            (rest term1) term2 (cons '1 position))))))))

(defun matching-position-and-subst-aux (term1 term2 position)
  (when (not (null term1))
    (nconc (matching-position-and-subst (first term1) term2 position)
           (matching-position-and-subst-aux (rest term1) term2
                                            (cons (1+ (first position))
                                                  (rest position))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: USER -*-
;;;
;;; This file contains the function rewrite-term.
;;;
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))

(defun one-step-rewriting (term rule &optional flag)
  (let ((subst (if (eq 'collapse flag)
                   (let ((subst-and-position
                          (matching-position-and-subst term (lhs-of-equation rule))))
                     (if (eq (first (first subst-and-position)) nil)
                         (rest subst-and-position)
                         subst-and-position))
                   (matching-position-and-subst term (lhs-of-equation rule)))))
    (if subst
        (instantiation
         (replace-term term (rhs-of-equation rule)
                       (subterm-position (first subst)))
         (mgu (first subst)))
        nil)))

(defun once-rewriting (term equation-set &optional flag)
  (let ((rewrited-term nil))
    (loop
     (when (endp equation-set)
       (return nil))

     (setf rewrited-term
           (one-step-rewriting term
                               (first equation-set) flag))
     (if (null rewrited-term)
         (setf equation-set (rest equation-set))

         (return rewrited-term)))))

(defun rewrite-term (term equation-set &optional flag)
  (if (null equation-set) term
      (let ((rewrited-term term))
        (loop
          (setf rewrited-term (once-rewriting rewrited-term equation-set flag))

          (if (null rewrited-term) (return term)
              (setf term rewrited-term))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;;
;;; This file contains the function unify.
;;;
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))

(defun unify (term1 term2 &optional (subst-so-far nil))
  (cond ((var-p term1)
         (var-unify term1 term2 subst-so-far))
        ((var-p term2)
         (var-unify term2 term1 subst-so-far))
        ((atom term1)
         (and (eq term1 term2) (list subst-so-far)))
        ((atom term2) nil)
        (t (let ((subst-list (unify (first term1) (first term2) subst-so-far)))
             (if subst-list
                 (unify (rest term1) (rest term2) (first subst-list))
                 nil)))))

(defun var-unify (var pat subst &aux (binding (assoc var subst)))
  (cond (binding (unify (second binding) pat subst))
        ((equal-var var pat subst) (list subst))
        ((not (occurs-in var pat subst))
         (list (cons (list var pat) subst)) )) )


(defun equal-var (var pat subst)
  (and (var-p pat)
       (or (equal var pat)
           (let ((binding (assoc pat subst) ))
             (and binding (equal-var var (second binding) subst)) )) ))


(defun occurs-in (var pat subst)
  (cond ((var-p pat)
         (or (equal var pat)
             (let ((binding (assoc pat subst) ))
               (and binding (occurs-in var (second binding) subst)) )))
        ((atom pat) nil)
        ((or (occurs-in var (first pat) subst)
             (occurs-in var (rest pat) subst) )) ))

;;;
;;;
;;;
(defun unifiable-position-and-subst (term1 term2 &optional position)
  (cond ((null term1) nil)
        ((var-p term1) nil)
        (t (let ((substitution (unify term1 term2)))
             (cond ((and (atom term1) substitution)
                    (list (cons (reverse position) substitution)))
                   ((atom term1) nil)
                   (t (if substitution
                          (cons (cons (reverse position) substitution)
                                (unifiable-position-and-subst-aux
                                  (rest term1)
                                  term2 (cons '1 position)))
                          (unifiable-position-and-subst-aux
                            (rest term1) term2 (cons '1 position))))))))))

(defun unifiable-position-and-subst-aux (term1 term2 position)
  (when (not (null term1))
    (nconc (unifiable-position-and-subst (first term1) term2 position)
           (unifiable-position-and-subst-aux (rest term1) term2
                                             (cons (1+ (first position))
                                                   (rest position))))))

;;;
;;;
;;;
(defun instantiation (term unifier)
  (cond ((null unifier) term)
        ((null term) nil)
        ((var-p term)
```

```
          (let ((subst (assoc term unifier)))
            (if subst (instantiation (instantiation (second subst) unifier) unifier)
                term)))
         ((atom term) term)
         ((var-p (first term))
          (let ((subst (assoc (first term) unifier)))
            (if subst (cons (instantiation (second subst) unifier)
                            (instantiation (rest term) unifier))
                (cons (first term) (instantiation (rest term) unifier)))))
         ((listp (first term))
          (cons (instantiation (first term) unifier)
                (instantiation (rest term) unifier)))
         (t (cons (first term) (instantiation (rest term) unifier)))))


(defun replace-term (term rterm position)
  (cond ((null term) nil)
        ((null position) rterm)
        (t (subst rterm (get-replaced-term term position) term))))

(defun get-replaced-term (term position)
  (if (null position) term
      (get-replaced-term (nth (first position) term) (rest position))))


(defun subterm-position (ps)
  (first ps))

(defun mgu (ps)
  (second ps))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-
;;;
;;;This file contains the function orinet.
;;;
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))

(defun lhs-of-equation (equation)
  (ctypecase equation
    (cons (first equation))))

(defun rhs-of-equation (equation)
  (ctypecase equation
    (cons (third equation))))


(defun orient (equation &optional (prec nil))
  (let ((lhs (lhs-of-equation equation))
        (rhs (rhs-of-equation equation)))
    (if (lpo> lhs rhs nil prec) (subst '-> '= equation)
        (if (lpo> rhs lhs nil prec) (list rhs '-> lhs)
            'fail))))



(defun lpo> (lhs-term rhs-term and-problems prec)
  (cond ((equal lhs-term rhs-term) nil)
        ((and (not (var-p lhs-term))
              (not (var-p rhs-term)))
         (let ((f (operator lhs-term))
               (g (operator rhs-term)))
           (cond ((eq f g)
                  (solve-lex-and-lpo> lhs-term (args lhs-term) (args rhs-term)
                                      and-problems prec))
                 ((operator> f g prec)
                  (solve-and-and-lpo> lhs-term (args rhs-term) and-problems prec))
                 ((operator> g f prec)
                  (solve-or-and-lpo>= (args lhs-term) rhs-term and-problems prec))
                 (t (solve-or-and-lpo>= (args lhs-term) rhs-term and-problems prec)))))
        ((and (not (var-p lhs-term)) (var-p rhs-term))
         (solve-or-and-lpo>= (args lhs-term) rhs-term and-problems prec))
        (t nil)))

(defun solve-and-and-lpo> (lhs-term rhs-term-args and-problems prec)
 (if (endp rhs-term-args)
     (if (endp and-problems) t
         (lpo> (first (first and-problems)) (second (first and-problems))
               (rest and-problems) prec))
     (lpo> lhs-term (first rhs-term-args)
           (extend-and-problems lhs-term (rest rhs-term-args) and-problems) prec)))

(defun solve-or-and-lpo>= (lhs-term-args rhs-term and-problems prec)
  (cond ((endp lhs-term-args) nil)
        ((equal (first lhs-term-args) rhs-term)
         (if (endp and-problems) t
             (lpo> (first (first and-problems)) (second (first and-problems))
                   (rest and-problems) prec)))
        (t (if (lpo> (first lhs-term-args) rhs-term and-problems prec) t
               (solve-or-and-lpo>= (rest lhs-term-args) rhs-term and-problems prec)))))


(defun solve-lex-and-lpo> (lhs-term lhs-term-args rhs-term-args and-problems prec)
  (cond ((endp lhs-term-args) nil)
        ((equal (first lhs-term-args) (first rhs-term-args))
         (solve-lex-and-lpo> lhs-term (rest lhs-term-args) (rest rhs-term-args)
                             and-problems prec))
        (t (lpo> (first lhs-term-args) (first rhs-term-args)
                 (extend-and-problems lhs-term (rest rhs-term-args) and-problems)
                 prec))))

;;;
;;;Operator> has 3 arguments which are function symbols and current precedence.
;;;If current precedence contains pair of two function symbols, OP> returns T
;;;othewise NIL.
;;;
(defun operator> (op1 op2 prec)
  (member op2 (member op1 prec :test #'eql) :test #'eql))
```

```
(defun args (term)
  (ctypecase term
    (list (rest term))
    (atom ())))

(defun operator (term)
  (ctypecase term
    (list (first term))
    (atom term)))

(defun extend-and-problems (lhs rhss and-problems)
  (if (endp rhss) and-problems
      (cons (list lhs (first rhss))
            (extend-and-problems lhs (rest rhss) and-problems))))

(defun var-p (term)
  (typecase term
    (symbol (char= #\? (char (symbol-name term) 0)))
    (t nil)))
```

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: user; Base: 10 -*-
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))
;;;
;;; This file contains the fucntion critical-pair
;;;
(defun critical-pair (rule1 rule2)
  (let* ((rename-rule (rename-variables rule1))
         (lhs1 (lhs-of-equation rename-rule))
         (lhs2 (lhs-of-equation rule2))
         (position-and-subst (unifiable-position-and-subst lhs1 lhs2)))
    (if (null position-and-subst) nil
        (mapcar #'(lambda (ps)
                    (list (instantiation
                            (replace-term
                              lhs1
                              (rhs-of-equation rule2)
                              (subterm-position ps))
                            (mgu ps))
                          '=
                          (instantiation (rhs-of-equation rename-rule) (mgu ps))))
                position-and-subst))))

(defun subterm-position (ps)
  (first ps))

(defun mgu (ps)
  (second ps))

(defun all-critical-pair (rules)
  (if (null rules) nil
      (mapcan #'(lambda (rule1)
                  (mapcan #'(lambda (rule2)
                              (critical-pair rule1 rule2))
                          rules))
              rules)))

(defun rename-variables (rule)
  (sublis (mapcar #'(lambda (var-name) (cons var-name (new-var)))
                  (variables-in rule))
          rule))

(defun new-var () (gensym "?"))

(defun variables-in (term)
  (cond ((var-p term) (list term))
        ((atom term) nil)
        (t (union (variables-in (first term))
                  (variables-in (rest term)) )) ))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-
;;;
;;; This file contains the function kb.
;;;
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))

(defmacro queue (item que)
  `(progn (if ,que
              (setf (rest (last ,que)) (list ,item))
              (setf ,que (list ,item)))
          ,que))

(defmacro dequeue (que)
  `(let ((result (first ,que)))
     (setf ,que (rest ,que))
     result))

(defun delete-equation (equations &optional (new-equations nil))
  (cond ((null equations) new-equations)
        ((equal (lhs-of-equation (first equations))
                (rhs-of-equation (first equations)))
         (delete-equation (rest equations) new-equations))
        (t (delete-equation (rest equations) (cons (first equations) new-equations)))))

(defun collapse (rule1 rule2)
  (let* ((rename-rule1 (rename-variables rule1))
         (lhs-1 (lhs-of-equation rename-rule1))
         (rewrited-term (rewrite-term lhs-1 (list rule2))))
    (if (eql lhs-1 rewrited-term) (values 'rule rule1)
        (values 'equation (list rewrited-term '= (rhs-of-equation rename-rule1))))))

(defun all-collapse (rules rule &optional new-rules new-equations)
  (if (null rules) (values new-rules new-equations)
      (multiple-value-bind
        (flag result)
          (collapse (first rules) rule)
        (if (eq 'rule flag)
            (all-collapse (rest rules) rule (cons (first rules) new-rules) new-equations)
            (all-collapse (rest rules) rule new-rules (cons result new-equations))))))


(defun compose (rule1 rule2)
  (let* ((rename-rule1 (rename-variables rule1))
         (rhs-1 (rhs-of-equation rename-rule1))
         (rewrited-term (rewrite-term rhs-1 (list rule2))))
    (if (eql rhs-1 rewrited-term) rule1
        (list (lhs-of-equation rename-rule1) '-> rewrited-term))))

(defun all-compose (rules rule)
  (mapcar #'(lambda (rule1)
              (compose rule1 rule))
          rules))

(defun simplify (equation rules)
  (let ((rename-eq (rename-variables equation)))
    (if (null rules) equation
        (list (rewrite-term (lhs-of-equation rename-eq) rules) '=
              (rewrite-term (rhs-of-equation rename-eq) rules)))))


(defun all-simplify (equations rules &optional new-equations)
  (if (null equations) new-equations
      (all-simplify (rest equations) rules
                    (adjoin (simplify (first equations) rules) new-equations
                            :test #'equal-equation))))

(defun kb (equations prec result &optional close open simplifier)
  (cond ((and (null equations) (null open) (null simplifier))
         (print close) (print prec) (set result :end))
        (simplifier
         (multiple-value-bind (new-open-1 new-equations-1)
             (all-collapse open (first simplifier))
           (let ((new-open-2 (all-compose new-open-1 (first simplifier))))
             (multiple-value-bind (new-close-1 new-equations-2)
                 (all-collapse close (first simplifier))
               (let ((new-close-2 (all-compose new-close-1 (first simplifier)))
```

```
                (new-equations-3 (nconc new-equations-1 new-equations-2 equations)))
             (kb new-equations-3 prec result new-close-2
                (add-rule (first simplifier) new-open-2) (rest simplifier)))))))
        ((and (null equations) (null simplifier))
         (kb (all-critical-pair-2 (first open) (cons (first open) close))
            prec result (cons (first open) close)(rest open) simplifier))
        ((null simplifier)
         (let ((new-equations
                (delete-equation (all-simplify equations (append open close)))))
           (if (null new-equations) (kb new-equations prec result close open simplifier)
               (multiple-value-bind (new-rule equation)
                   (orientable-equation new-equations prec)
                 (cond ((and (eq 'fail new-rule) (null open)) 'fail)
                       ((eq 'fail new-rule)
                        (kb (nconc (all-critical-pair-2 (first open)
                                                        (cons (first open) close))
                                   new-equations)
                            prec result (cons (first open) close) (rest open) simplifier))
                       (t (kb (remove equation new-equations :test #'eq)
                              prec result close open (cons new-rule simplifier))))))))))

(defun orientable-equation (equations prec)
  (if (null equations) (values 'fail nil)
      (let ((rule (orient (first equations) prec)))
        (if (eq 'fail rule) (orientable-equation (rest equations) prec)
            (values rule (first equations))))))

(defun all-critical-pair-2 (rule rules)
  (union (mapcan #'(lambda (rule1)
                     (critical-pair rule rule1))
                 rules)
         (mapcan #'(lambda (rule2)
                     (critical-pair rule2 rule))
                 rules)
         :test #'equal-equation))

(defun add-rule (rule rules)
  (if (member rule rules :test #'equal-equation)
      rules
      (queue rule rules)))

(defun equal-equation (eq1 eq2)
  (or (equal eq1 eq2)
      (equal eq1 (reverse eq2))
      (and (match-term (lhs-of-equation eq1) (lhs-of-equation eq2))
           (match-term (rhs-of-equation eq1) (rhs-of-equation eq2))
           (match-term (lhs-of-equation eq2) (lhs-of-equation eq1))
           (match-term (rhs-of-equation eq2) (rhs-of-equation eq1)))
      (and (match-term (lhs-of-equation eq1) (rhs-of-equation eq2))
           (match-term (rhs-of-equation eq1) (lhs-of-equation eq2))
           (match-term (rhs-of-equation eq2) (lhs-of-equation eq1))
           (match-term (lhs-of-equation eq2) (rhs-of-equation eq1)))))
```

```
;;; -*- Syntax: Common-Lisp; Mode: LISP; Package: USER; Base: 10 -*-
(proclaim '(optimize (compilation-speed 0) (safety 0) (speed 3)))
;;;
;;; This file contains the multiple-comp.
;;;
(defvar *position* 0)

(defvar *label-hash-table* nil)

(defvar *precedence-hash-table* nil)

(defun init-hash-table ()
  (setf *position* 0)
  (setf *label-hash-table* (make-hash-table :test #'eql :size 10000))
  (setf (gethash 0 *label-hash-table*) nil)
  (setf *precedence-hash-table* (make-hash-table :test #'eql :size 10000))
  'initialized)

(defun record-prec (precs)
  (cond ((null precs) 'recorded)
        (t (setf (gethash *position* *precedence-hash-table*) (first precs))
           (record-label (expt 2 *position*) (list *position*))
           (incf *position*)
           (record-prec (rest precs)))))

(defun get-prec (number)
  (gethash number *precedence-hash-table*))

(defun record-label (bitvector numbers-list)
  (setf (gethash bitvector *label-hash-table*) numbers-list))

(defun get-label (bitvector)
  (gethash bitvector *label-hash-table*))

;;;
;;;
;;;

(defun bitvector-to-label (bitvector)
  (if (= bitvector 0) nil
      (let ((log-ans  (floor (log bitvector 2))))
        (bitvector-to-label-aux bitvector 0 log-ans))))

(defun bitvector-to-label-aux (bitvector &optional n index result)
  (cond ((> n index) (nreverse result))
        ((logbitp n bitvector)
         (bitvector-to-label-aux bitvector (+ n 1) index (cons n result)))
        (t (bitvector-to-label-aux bitvector (+ n 1) index result))))


(defun label-to-bitvector (numbers-list)
  (if (null numbers-list) 0
      (apply #'+ (mapcar #'(lambda (n)
                             (expt 2 n))
                         numbers-list))))

(deftype label ()
  'integer)

(deftype var ()
  '(and symbol (satisfies var-p)))

(defun var-p (term)
  (typecase term
    (symbol (char= #\? (char (symbol-name term) 0)))
    (t nil)))

(deftype constant ()
  '(or symbol number))

(deftype comp-ex ()
  'cons)

(deftype term ()
```

```
  '(or var constant comp-ex))

(deftype node ()
  '(and cons (satisfies node-p)))

(defun node-p (lst)
  (and (= 3 (length lst))
       (every #'(lambda (elm)
                  (typep elm 'label))
              lst)))

(deftype equation ()
  '(and cons (satisfies equation-p)))


(defun equation-p (lst)
  (and (typep (first lst) 'term)
       (eq (second lst) '=)
       (typep (third lst) 'term)
       (= 3 (length lst))))


(deftype equation-node ()
  '(and cons (satisfies equation-node-p)))

(defun equation-node-p (lst)
  (and (typep (first lst) 'equation)
       (typep (second lst) 'node)
       (= 2 (length lst))))
;;;
;;;
;;;
(defun lhs-of-equation (equation)
  (etypecase equation
    (equation-node (first (first equation)))
    (equation (first equation))))

(defun rhs-of-equation (equation)
  (ctypecase equation
    (equation-node (third (first equation)))
    (equation (third equation))))

;;;
;;;
;;;
(defun get-node (equation-node)
  (second equation-node))

(defun get-equation (equation-node)
  (first equation-node))
;;;
;;;
;;;
(defun write-label1 (label equation-node)
  (setf (first (get-node equation-node)) label))

(defun write-label2 (label equation-node)
  (setf (second (get-node equation-node)) label))

(defun write-label3 (label equation-node)
  (setf (third (get-node equation-node)) label))

;;;
;;;
;;;
(defun get-label1 (equation-node)
  (first (get-node equation-node)))

(defun get-label2 (equation-node)
  (second (get-node equation-node)))

(defun get-label3 (equation-node)
  (third (get-node equation-node)))

(defun get-weight (equation-node)
  (fourth (get-node equation-node)))
```

```
;;;
;;;
;;;
(defun make-equation-node (equation label1 label2 label3)
  (list equation
        (list label1
              label2
              label3
              (+ (count-atom (lhs-of-equation equation))
                 (count-atom (rhs-of-equation equation))))))

;;;
;;;
;;;
(defun equal-equation (eq1 eq2)
  (or (and (equal eq1 eq2) 'l-to-r)
      (and (equal eq1 (reverse eq2)) 'r-to-l)
      (and (match-term (lhs-of-equation eq1) (lhs-of-equation eq2))
           (match-term (rhs-of-equation eq1) (rhs-of-equation eq2))
           (match-term (lhs-of-equation eq2) (lhs-of-equation eq1))
           (match-term (rhs-of-equation eq2) (rhs-of-equation eq1)) 'l-to-r)
      (and (match-term (lhs-of-equation eq1) (rhs-of-equation eq2))
           (match-term (rhs-of-equation eq1) (lhs-of-equation eq2))
           (match-term (rhs-of-equation eq2) (lhs-of-equation eq1))
           (match-term (lhs-of-equation eq2) (rhs-of-equation eq1)) 'r-to-l)))

;;;
;;;
;;;
(defun orient-prec (term1 term2 bitvector)
  (let* ((label (bitvector-to-label bitvector))
         (orientable-precs (remove-if-not
                            #'(lambda (position)
                                (lpo> term1 term2 nil
                                      (get-prec position)))
                            label))
         (bitvector1 (label-to-bitvector orientable-precs)))
    bitvector1))

(defun l-orient (equation-node)
  (let* ((equation (get-equation equation-node))
         (label3 (get-label3 equation-node))
         (add-bitvector1
          (orient-prec (lhs-of-equation equation)
                       (rhs-of-equation equation)
                       label3))
         (rest-label1 (logandc2 label3 add-bitvector1))
         (add-bitvector2
          (orient-prec (rhs-of-equation equation)
                       (lhs-of-equation equation)
                       rest-label1))
         (rest-label2 (logandc2 rest-label1 add-bitvector2)))
    (block update
      (write-label1 (logior (get-label1 equation-node) add-bitvector1)
                    equation-node)
      (write-label2 (logior (get-label2 equation-node) add-bitvector2)
                    equation-node)
      (write-label3 rest-label2 equation-node)
      equation-node)))

;;;
;;;
;;;
(defun only-exist-third-label-p (equation-node)
  (and (= 0 (get-label1 equation-node))
       (= 0 (get-label2 equation-node))
       (/= 0 (get-label3 equation-node))))

(defun all-empty-label-p (equation-node)
  (and (= 0 (get-label1 equation-node))
       (= 0 (get-label2 equation-node))
       (= 0 (get-label3 equation-node))))

;;;
;;;
;;;
```

```
(defun l-critical-pair (equation-node1 equation-node2)
  (let* ((equation1 (get-equation equation-node1))
         (equation2 (get-equation equation-node2))
         (label1-1 (get-label1 equation-node1))
         (label2-1 (get-label2 equation-node1))
         (label1-2 (get-label1 equation-node2))
         (label2-2 (get-label2 equation-node2))
         (env1 (logand label1-1 label1-2))
         (env2 (logand label1-1 label2-2))
         (env3 (logand label2-1 label1-2))
         (env4 (logand label2-1 label2-2))
         (cp1 (delete-trivial-equation
                (and (/= 0 env1)
                     (critical-pair equation1 equation2))))          ;env1
         (cp2 (delete-trivial-equation
                (and (/= 0 env1)
                     (critical-pair equation2 equation1))))          ;env1
         (cp3 (delete-trivial-equation
                (and (/= 0 env2)
                     (critical-pair equation1 (reverse equation2))))) ;env2
         (cp4 (delete-trivial-equation
                (and (/= 0 env2)
                     (critical-pair (reverse equation2) equation1)))) ;env2
         (cp5 (delete-trivial-equation
                (and (/= 0 env3)
                     (critical-pair (reverse equation1) equation2)))) ;env3
         (cp6 (delete-trivial-equation
                (and (/= 0 env3)
                     (critical-pair equation2 (reverse equation1))))) ;env3
         (cp7 (delete-trivial-equation
                (and (/= 0 env4)
                     (critical-pair (reverse equation1)
                                    (reverse equation2))))) ;env4
         (cp8 (delete-trivial-equation
                (and (/= 0 env4)
                     (critical-pair (reverse equation2)
                                    (reverse equation1))))) ;env4
         (eq-env-pair1 (make-equations-and-env-pair
                         (union cp1 cp2 :test #'equal-equation) env1))
         (eq-env-pair2 (make-equations-and-env-pair
                         (union cp3 cp4 :test #'equal-equation) env2))
         (eq-env-pair3 (make-equations-and-env-pair
                         (union cp5 cp6 :test #'equal-equation) env3))
         (eq-env-pair4 (make-equations-and-env-pair
                         (union cp7 cp8 :test #'equal-equation) env4)))
    (nconc eq-env-pair1 eq-env-pair2 eq-env-pair3 eq-env-pair4)))

;;;
;;;
;;;
(defun make-equations-and-env-pair (cp env)
  (mapcar #'(lambda (equation)
              (make-equation-node equation 0 0 env))
          cp))

(defun delete-trivial-equation (equations)
  (delete-if #'trivial-equation-p equations))

;;;
;;;
;;;
(defun all-l-critical-pair (equation equations)
  (nconc (l-critical-pair equation equation)
         (mapcan #'(lambda (x)
                     (l-critical-pair equation x))
                 equations)))

;;;
;;;
;;;
(defun delete-l (equations)
  (remove-if #'all-empty-label-p equations))

;;;
;;;
;;;
```

```
(defun encompassment> (term1 term2)
  (and (matching-position-and-subst term1 term2)
       (not (matching-position-and-subst term2 term1))))

;;;
;;;
;;;
(defun rewrite-1-aux-1 (equation-node1 equation-node2)
  (let ((equation1 (get-equation equation-node1))
        (equation2 (get-equation equation-node2))
        (new-label1 (logand (get-label1 equation-node1)
                            (get-label1 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label1 equation-node2)))
        (condition (logand  (get-label2 equation-node1)
                            (get-label1 equation-node2))))
        (when (and (or (/= 0 new-label1) (/= 0 new-label3))
                   (or (not (encompassment> (rhs-of-equation equation1)
                                            (lhs-of-equation equation2)))
                       (= 0 condition)))
          (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                             (list (rename-variables equation2)))))
            (if (eql rewrited-term (rhs-of-equation equation1))
                nil
                (block update
                  (write-label1
                   (logandc2 (get-label1 equation-node1)
                             (get-label1 equation-node2))
                   equation-node1)
                  (write-label3
                    (logandc2 (get-label3 equation-node1)
                              (get-label1 equation-node2))
                   equation-node1)
                  (if (equal (lhs-of-equation equation1) rewrited-term) nil
                      (make-equation-node (list (lhs-of-equation equation1)
                                                '=
                                                rewrited-term)
                                          new-label1 0 new-label3))))))))

(defun rewrite-1-aux-2 (equation-node1 equation-node2)
  (let ((equation1 (get-equation equation-node1))
        (equation2 (reverse (get-equation equation-node2)))
        (new-label1 (logand (get-label1 equation-node1)
                            (get-label2 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label2 equation-node2)))
        (condition (logand (get-label2 equation-node1)
                           (get-label2 equation-node2))))
        (when (and (or (/= 0 new-label1) (/= 0 new-label3))
                   (or (not (encompassment> (rhs-of-equation equation1)
                                            (lhs-of-equation equation2)))
                       (= 0 condition)))
          (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                             (list (rename-variables equation2)))))
            (if (eql rewrited-term (rhs-of-equation equation1))
                nil
                (block update
                  (write-label1
                    (logandc2 (get-label1 equation-node1)
                              (get-label2 equation-node2))
                   equation-node1)
                  (write-label3
                    (logandc2 (get-label3 equation-node1)
                              (get-label2 equation-node2))
                   equation-node1)

                  (if (equal (lhs-of-equation equation1) rewrited-term) nil
                      (make-equation-node (list (lhs-of-equation equation1)
                                                '=
                                                rewrited-term)
                                          new-label1 0 new-label3))))))))

(defun rewrite-1-aux-3 (equation-node1 equation-node2)
  (let ((equation1 (reverse (get-equation equation-node1)))
        (equation2 (get-equation equation-node2))
        (new-label1 (logand (get-label2 equation-node1)
```

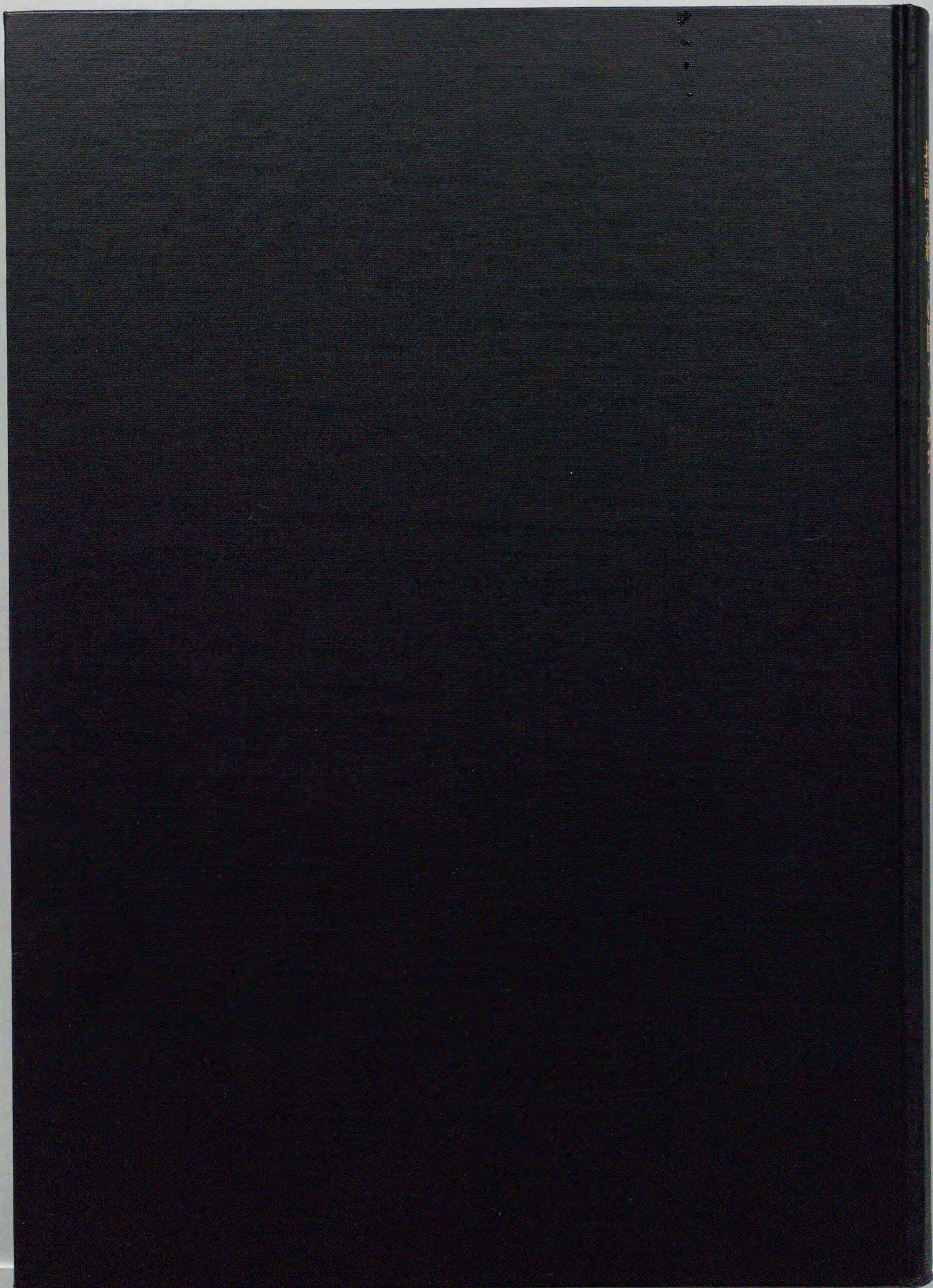
```
                         (get-label1 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label1 equation-node2)))
        (condition (logand (get-label1 equation-node1)
                           (get-label1 equation-node2))))
        (when (and (or (/= 0 new-label1) (/= 0 new-label3))
                   (or (not (encompassment> (rhs-of-equation equation1)
                                            (lhs-of-equation equation2)))
                       (= 0 condition)))
          (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                             (list (rename-variables equation2)))))
            (if (eql rewrited-term (rhs-of-equation equation1))
                nil
                (block update
                  (write-label2
                    (logandc2 (get-label2 equation-node1)
                              (get-label1 equation-node2))
                    equation-node1)
                  (write-label3
                    (logandc2 (get-label3 equation-node1)
                              (get-label1 equation-node2))
                    equation-node1)

                  (if (equal (lhs-of-equation equation1) rewrited-term) nil
                      (make-equation-node (list (lhs-of-equation equation1)
                                                '=
                                                rewrited-term)
                                          new-label1 0 new-label3))))))))

(defun rewrite-1-aux-4 (equation-node1 equation-node2)
  (let ((equation1 (reverse (get-equation equation-node1)))
        (equation2 (reverse (get-equation equation-node2)))
        (new-label1 (logand (get-label2 equation-node1)
                            (get-label2 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label2 equation-node2)))
        (condition (logand (get-label1 equation-node1)
                           (get-label2 equation-node2))))
        (when (and (or (/= 0 new-label1) (/= 0 new-label3))
                   (or (not (encompassment> (rhs-of-equation equation1)
                                            (lhs-of-equation equation2)))
                       (= 0 condition)))
          (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                             (list (rename-variables equation2)))))
            (if (eql rewrited-term (rhs-of-equation equation1))
                nil
                (block update
                  (write-label2
                    (logandc2 (get-label2 equation-node1)
                              (get-label2 equation-node2))
                    equation-node1)
                  (write-label3
                    (logandc2 (get-label3 equation-node1)
                              (get-label2 equation-node2))
                    equation-node1)

                  (if (equal (lhs-of-equation equation1) rewrited-term) nil
                      (make-equation-node (list (lhs-of-equation equation1)
                                                '=
                                                rewrited-term)
                                          new-label1 0 new-label3))))))))

;;;
;;;
;;;
(defun rewrite-2-aux-1 (equation-node1 equation-node2)
  (let ((equation1 (get-equation equation-node1))
        (equation2 (get-equation equation-node2))
        (new-label1 (logand (get-label1 equation-node1)
                            (get-label1 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label1 equation-node2)))
        (condition (logand (get-label2 equation-node1)
                           (get-label1 equation-node2))))
    (when (and (/= 0 condition)
               (encompassment> (rhs-of-equation equation1)
```

```
                                (lhs-of-equation equation2)))
      (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                         (list (rename-variables equation2)))))
        (if (eql rewrited-term (rhs-of-equation equation1))
            nil
            (block update
              (write-label1
                (logandc2 (get-label1 equation-node1)
                          (get-label1 equation-node2))
                equation-node1)
              (write-label2
                (logandc2 (get-label2 equation-node1)
                          (get-label1 equation-node2))
                equation-node1)
              (write-label3
                (logandc2 (get-label3 equation-node1)
                          (get-label1 equation-node2))
                equation-node1)
              (if (equal (lhs-of-equation equation1) rewrited-term) nil
                  (make-equation-node
                   (list (lhs-of-equation equation1)
                         '=
                         rewrited-term)
                   new-label1 0
                   (logior condition new-label3)))))))))

(defun rewrite-2-aux-2 (equation-node1 equation-node2)
  (let ((equation1 (get-equation equation-node1))
        (equation2 (reverse (get-equation equation-node2)))
        (new-label1 (logand (get-label1 equation-node1)
                            (get-label2 equation-node2)))
        (new-label3 (logand (get-label3 equation-node1)
                            (get-label2 equation-node2)))
        (condition (logand (get-label2 equation-node1)
                           (get-label2 equation-node2))))
    (when (and (/= 0 condition)
               (encompassment> (rhs-of-equation equation1)
                               (lhs-of-equation equation2)))
      (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                         (list (rename-variables equation2)))))
        (if (eql rewrited-term (rhs-of-equation equation1))
            nil
            (block update
              (write-label1
                (logandc2 (get-label1 equation-node1)
                          (get-label2 equation-node2))
                equation-node1)
              (write-label2
                (logandc2 (get-label2 equation-node1)
                          (get-label2 equation-node2))
                equation-node1)
              (write-label3
                (logandc2 (get-label3 equation-node1)
                          (get-label2 equation-node2))
                equation-node1)
              (if (equal (lhs-of-equation equation1) rewrited-term) nil
                  (make-equation-node
                   (list (lhs-of-equation equation1)
                         '=
                         rewrited-term)
                   new-label1 0 (logior condition new-label3)))))))))

(defun rewrite-2-aux-3 (equation-node1 equation-node2)
  (let* ((equation1 (reverse (get-equation equation-node1)))
         (equation2 (get-equation equation-node2))
         (label1 (get-label1 equation-node1))
         (label2 (get-label2 equation-node1))
         (new-label1 (logand label2 (get-label1 equation-node2)))
         (new-label3 (logand (get-label3 equation-node1)
                             (get-label1 equation-node2)))
         (condition (logand label1 (get-label1 equation-node2))))
    (when (and (/= 0 condition)
               (encompassment> (rhs-of-equation equation1)
                               (lhs-of-equation equation2)))
```

```
      (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                         (list (rename-variables equation2)))))
        (if (eql rewrited-term (rhs-of-equation equation1))
            nil
            (block update
              (write-label1
                (logandc2 label1 (get-label1 equation-node2))
                equation-node1)
              (write-label2
                (logandc2 label2 (get-label1 equation-node2))
                equation-node1)
              (write-label3
                (logandc2 (get-label3 equation-node1)
                          (get-label1 equation-node2))
                equation-node1)
              (if (equal (lhs-of-equation equation1) rewrited-term) nil
                  (make-equation-node
                   (list (lhs-of-equation equation1)
                         '=
                         rewrited-term)
                   new-label1 0 (logior condition new-label3))))))))


(defun rewrite-2-aux-4 (equation-node1 equation-node2)
  (let* ((equation1 (reverse (get-equation equation-node1)))
         (equation2 (reverse (get-equation equation-node2)))
         (label1 (get-label1 equation-node1))
         (label2 (get-label2 equation-node1))
         (new-label1 (logand label2 (get-label2 equation-node2)))
         (new-label3 (logand (get-label3 equation-node1)
                             (get-label2 equation-node2)))
         (condition (logand label1 (get-label2 equation-node2))))
    (when (and (/= 0 condition)
               (encompassment> (rhs-of-equation equation1)
                               (lhs-of-equation equation2)))
      (let ((rewrited-term (rewrite-term (rhs-of-equation equation1)
                                         (list (rename-variables equation2)))))
        (if (eql rewrited-term (rhs-of-equation equation1))
            nil
            (block update
              (write-label1
                (logandc2 label1 (get-label2 equation-node2))
                equation-node1)
              (write-label2
                (logandc2 label2 (get-label2 equation-node2))
                equation-node1)
              (write-label3
               (logandc2 (get-label3 equation-node1)
                         (get-label2 equation-node2))
                equation-node1)
              (if (equal (lhs-of-equation equation1) rewrited-term) nil
                  (make-equation-node
                   (list (lhs-of-equation equation1)
                         '=
                         rewrited-term)
                   new-label1 0 (logior condition new-label3))))))))

;;;
;;;
;;;
(defun rewrite-aux (node1 node2)
  (or (rewrite-1-aux-1 node1 node2)
      (rewrite-1-aux-2 node1 node2)
      (rewrite-1-aux-3 node1 node2)
      (rewrite-1-aux-4 node1 node2)
      (rewrite-2-aux-1 node1 node2)
      (rewrite-2-aux-2 node1 node2)
      (rewrite-2-aux-3 node1 node2)
      (rewrite-2-aux-4 node1 node2)))

(defun rewrite (node1 node2 &optional rewrited-nodes)
  (let ((rewrited-node (rewrite-aux node1 node2)))
    (if rewrited-node
        (rewrite node1 node2 (cons rewrited-node rewrited-nodes))
        rewrited-nodes)))
```

```
(defun rewrite-1-step (node nodes &optional generated-nodes)
  (let ((label1 (get-label1 node))
        (label2 (get-label2 node))
        (label3 (get-label3 node)))
    (loop
      (when (and (endp nodes)
                 (and (= label1 (get-label1 node))
                      (= label2 (get-label2 node))
                      (= label3 (get-label3 node))))
        (return (values 'no-rewrite generated-nodes)))
      (when (all-empty-label-p node) (return (values nil generated-nodes)))
      (setf generated-nodes
            (nconc generated-nodes (rewrite node (first nodes))))
      (if (and (= label1 (get-label1 node))
               (= label2 (get-label2 node))
               (= label3 (get-label3 node)))
          (setf nodes (rest nodes))
          (return (values node generated-nodes))))))

(defun all-rewrite (node nodes)
  (if (endp nodes) node
      (let ((generated-nodes nil))
        (loop
          (multiple-value-bind (node1 new-nodes)
              (rewrite-1-step node nodes generated-nodes)
            (cond ((null node1) (return generated-nodes))
                  ((eq 'no-rewrite node1) (return generated-nodes))
                  (t (setf generated-nodes new-nodes))))))))


(defun all-rewrite-s (nodes1 nodes2 &optional temp)
  (cond ((null nodes2) nodes1)
        ((null nodes1) temp)
        (t (let ((rewrited-nodes (all-rewrite (first nodes1) nodes2)))
             (if (null rewrited-nodes)
                 (if (all-empty-label-p (first nodes1))
                     (all-rewrite-s (rest nodes1) nodes2 temp)
                     (all-rewrite-s (rest nodes1) nodes2 (cons (first nodes1) temp)))
                 (if (all-empty-label-p (first nodes1))
                     (all-rewrite-s (append (rest nodes1) rewrited-nodes) nodes2 temp)
                     (all-rewrite-s (append (rest nodes1) rewrited-nodes) nodes2
                                    (cons (first nodes1) temp))))))))


;;;
;;;
;;;
(defun trivial-equation-p (equation)
  (equal (lhs-of-equation equation) (rhs-of-equation equation)))

;;;
;;;
;;;
(defun diff-close-open (close open)
  (set-difference close open :test #'equal-equation :key #'first))

;;;
;;;
;;;
(defun multiple-comp (open close orderings)
  (cond ((terminate-p open close orderings) close)
        ((null open) 'fail)
        (t (let* ((equation-node (pop open))
                  (rewrite-1-results1
                     (delete equation-node
                             (all-rewrite-s (list equation-node) close)
                             :test #'eq))
                  (new-open1 (sort (nconc open rewrite-1-results1)
                                   #'< :key #'get-weight)))
             (cond ((all-empty-label-p equation-node)
                    (multiple-comp new-open1 close orderings))
                   ((only-exist-third-label-p (l-orient equation-node))
                    (multiple-comp new-open1
                                   (cons equation-node close)
```

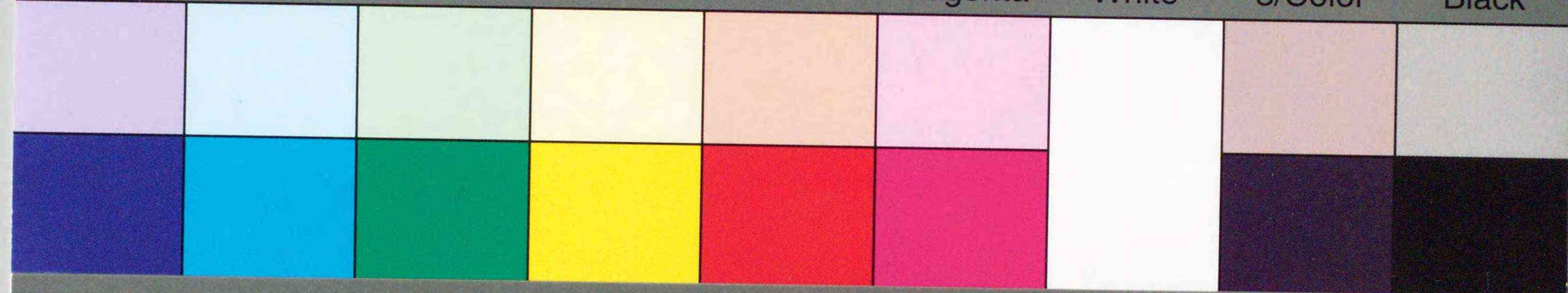
```
                                 orderings))
                  (t (let* ((rewrite-results2
                             (delete-1 (all-rewrite-s close
                                                      (list equation-node))))
                            (new-close1 (intersection rewrite-results2
                                                     close :test #'equal))
                            (open2 (set-difference rewrite-results2 new-close1
                                                  :test #'equal))
                            (new-open2  (nconc new-open1 open2))
                            (cp (all-1-critical-pair equation-node new-close1))
                            (new-open3 (sort (nconc new-open2 cp)
                                             #'< :key #'get-weight)))
                       (multiple-comp new-open3
                                      (cons equation-node new-close1)
                                      orderings)))))))

;;;
;;;
;;;
(defun terminate-p (open close precs)
  (some #'(lambda (prec)
            (and (not-exist-in-third-label prec close)
                 (not-exist-in-all-labels prec open)))
        precs))

;;;
;;;
;;;
(defun not-exist-in-third-label (prec close)
  (every #'(lambda (equation)
             (= 0 (logand prec (get-label3 equation))))
         close))

;;;
;;;
;;;
(defun not-exist-in-all-labels (prec open)
  (every #'(lambda (equation)
             (= 0 (logand prec (logior (get-label1 equation)
                                       (logior (get-label2 equation)
                                               (get-label3 equation))))))
         open))

;;;
;;;
;;;
(defun count-atom (term)
  (cond ((null term) 0)
        ((atom term) 1)
        (t (+ (count-atom (first term)) (count-atom (rest term))))))
```



Kodak Color Control Patches

Blue Cyan Green Yellow Red Magenta White 3/Color Black

© Kodak, 2007 TM: Kodak



Kodak Gray Scale

A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19

© Kodak, 2007 TM: Kodak