Vol. 29 No. 12 情報処理学会論文誌 Dec. 1988

プロダクションシステムにおける条件記述の最適化†

石田　亨††

プロダクションシステムはワーキングメモリ中のデータ量が増大すると急激に性能が劣化することが知られている．これは，①ルールの条件判定に含まれる一連の join 演算に要する時間が，通常の処理系ではデータ量の2乗に比例すること，②不適切な条件の記述により，多量の中間的な演算結果が生成され，後続する join 演算の処理量をさらに増大させることが原因と考えられる．本論文ではプロダクションシステムの条件記述を，join 演算の順序や組合せ方法（join 構造）を変更することによって最適化するアルゴリズムを示す．本アルゴリズムは以下の特徴を有する．（1）プロダクションシステムの動作特性を表す統計データを基に最適化を行う．（2）複数のルールを一括して最適化し，join 演算を可能な限り複数のルールにまたがり共通化する．また，このアルゴリズムに基づきプロダクションシステムの最適化機能を試作し，既存のプロダクションシステムプログラムに適用した結果，人手による改善を凌ぐ効果が得られたので報告する．

1. まえがき

プロダクションシステムは，データ量が増大すると急激に性能が劣化することが知られている[1]．これは，①ルールの条件判定に含まれる一連の join 演算に要する時間が，多くの処理系ではデータ量の2乗に比例すること*．②不適切な条件の記述により，多量の中間的な演算結果が生成され，後続する join 演算の処理量をさらに増大させることが原因と考えられる．

この問題を解決するために，YES/OPS[14]，ART[2] 等では，OPS 5[3] を拡張し，利用者がルールの条件部で条件要素（すなわち，join 演算）の順序や組合せ方法（join 構造）を自由に指定できるようにしている．また，効率の良い join 構造を作成するために様々なヒューリスティクスも知られ始めている．しかし，これらのヒューリスティクスは互いに競合するため，どれか1つを用いても必ずしも効率のよい join 構造を導かないという問題がある．したがって，最適化機能**がない段階では，プログラムの改善は試行錯誤的に行わざるを得なかった．最適化機能はプログラムの改善を自動化するものであるが，その他にも以下の意義を有している．

（1）エキスパートシステム利用者による最適化を可能とする．

join 構造の実行効率は，プロダクションシステムプログラムの動作特性に依存する．したがって，実行中に統計データを測定し，それをもとに join 構造の選択を行う必要がある．このことは，たとえルールは同一でも，対象とするデータが異なれば（例えば交換機の故障診断システムでは対象とする機種が異なれば）効率の良い join 構造が異なることを意味している．すなわち，最適化はエキスパートシステム開発側だけの問題ではなく，利用側でも行う必要がある．しかし，人手にたよっている段階では，利用側で最適化を行うことは実質的に不可能である．

（2）保守性を損なわずに性能を向上させる．

プロダクションシステムが広く用いられているのは，ルールによる知識の記述が保守性に優れているためである．ところが，性能のための最適化はプログラムの読解性を劣化させてしまう．短期的な性能向上を得るためにソースプログラムを変更し，長期的なメンテナビリティを犠牲にすることには問題が多い．プロダクションシステムの利点を損なわないためには，保守されるべきソースプログラムファイルと実行されるべき最適化されたプログラムファイルとを独立に持つ必要がある．すなわち，ルールが更新される度に，ソースプログラムファイルを最適化し実行プログラムファイルを生成する必要があるが，このためには自動的な最適化機能が必要である．

本論文では，人間の手を煩わすことなく高性能なプロダクションシステムプログラムを生成することを目的に，join 演算の処理コストを削減するよう，ルールの条件記述を最適化するアルゴリズムを提案する．本

† Optimizing Condition Parts of Production System Programs by Toru Ishida (NTT Communications and Information Processing Laboratories).

†† NTT 情報通信処理研究所

* OPS 5 を始めとする多くの処理系では，データは単純なリストで管理され，join 演算は2重ループで計算されている．ここでの記述はこの事実に基づいたものであるが，本論文で提案するアルゴリズムはデータがハッシュ[5] されている場合にも有効である．

** 本論文で用いる「最適化」という用語は，ルールの左辺に記述された条件の順序や組合せを変更することにより，ルールの実行性能を向上させる処理を意味する．コンパイラにおける目的コードの最適化と同様，必ずしも「最適な解」の追求を意味するものではない．

アルゴリズムは以下の特徴を有する．

① プロダクションシステムの動作特性を表す統計データをもとに最適化を行う．

② 複数のルールを一括して最適化し，join 演算をルール間にまたがり可能な限り共通化する．

以下ではまず，2章で最適化問題を定義する．3章でプロダクションシステムのコストモデルを示し，4章で最適化アルゴリズムを提案する．このアルゴリズムは，これまでに知られている効率化のためのヒューリスティクスを直接適用しようとするものではない．むしろ可能な join 構造を数え上げ，その中からコストが最小のものを選択するというアプローチをとっている．指数オーダで存在する join 構造を，各種の制約を用いて効率よく絞り込むところにアルゴリズムの中心がある．

また本アルゴリズムに基づき，実験用プロダクションシステム PLANET 上に最適化機能を試作した[6),8)]．既存のエキスパートシステム[9)]の最適化に適用した結果を5章に述べる．最適化アルゴリズムは，過去に行われた人手による最適化を凌ぐ効果を示している．

2. 最適化問題の定義

2.1 プロダクションシステム

プロダクションシステムは，ルールを格納するプロダクションメモリ (PM) とデータを格納するワーキングメモリ (WM)，および PM, WM の双方を参照して実行するインタプリタから構成されている．WM 中のデータはワーキングメモリエレメント (WME) と呼ばれる．

それぞれのルールは，left-hand-side (LHS) と呼ばれる条件要素 (condition element) の連接 (conjunction) と，right-hand-side (RHS) と呼ばれる動作 (action) の集合から構成されている．LHS には正の条件要素 (positive condition element) と負の条件要素 (negative condition element) を記述することができる．正の条件要素は条件に合致する WME が存在すれば成立する．逆に負の条件要素は条件に合致する WME が存在しなければ成立する．RHS には条件成立時に実行される WME の追加削除が記述される．本論文で対象とするルールの構文を表1に示す．

表1 ルールの構文則
Table 1 Syntax of rules.

構文則	〈プロダクションルール〉::= 　(defrule 〈ルール名〉〈条件連接〉--〉〈動作〉)⁺ 〈条件連接〉::=〈条件要素〉⁺ 〈条件要素〉::=〈正の条件要素〉|〈負の条件要素〉| 　(and 〈条件連接〉) 〈正の条件要素〉::= 　〈パターン〉|(bind 〈パターン変数〉〈パターン〉) 〈負の条件要素〉::=(not 〈パターン〉) 〈動作〉::=〈正の動作〉|〈負の動作〉 〈正の動作〉::=(make 〈パターン〉) 〈負の動作〉::=(remove 〈パターン変数〉) 〈パターン〉::=(〈クラス名〉〈値〉*) (注) ⁺は1回以上，*は0回以上繰り返すことを表す．
説明	①〈パターン変数〉は‘?’で始まるシンボル． ②〈パターン〉中の〈値〉は，パターン変数を含むことができる．〈クラス名〉は bind, not であってはならない． ③〈条件連接〉はすべての〈条件要素〉が成立した時に成立する．先頭は〈正の条件要素〉でなければならない． ④〈正の条件要素〉では，〈パターン〉中のパターン変数に WME の対応する値が束縛される．また bind では〈パターン〉と照合が成功した WME が〈パターン変数〉に束縛される． ⑤〈負の条件要素〉での〈パターン〉中のパターン変数への束縛は以降の条件照合に影響しない． ⑥〈正の動作〉では，〈パターン〉中のパターン変数を束縛値に置き換えた結果得られる WME が WM に追加される． ⑦〈負の動作〉では，〈パターン変数〉に束縛された WME が除去される．

プロダクションシステムインタプリタは，以下のサイクルを繰り返し実行する．

① 条件照合：すべてのルールについて，LHS とその時点の WM との照合を行い，インスタンシエーション（照合が成功したルールと，成功に寄与した WME を組み合わせたもの）を生成する．

② 競合解決：照合が成功したインスタンシエーションの中から1つを定められた戦略に従って選び出す．

③ 動作：選択されたインスタンシエーションを実行し，WME の追加除去を行う．一度実行の対象となったインスタンシエーションは再び実行されることはない．

RETE アルゴリズム[4)]では，ルールの条件部は RETE ネットワークと呼ばれるデータフローグラフに変換される．動作フェーズで WME が追加されると，その WME（トークンと呼ばれる）が RETE ネットワーク中に流し込まれ，変化に伴うネットワークの更新が行われる．この更新（条件照合フェーズに相当する）は，以下の手順で進められる．まず，LHS の各条件に対して1つの条件内で完了する条件要素内テスト (intra-condition test または selection) が1

入力ノード（one-input node）で行われ，テストを通過したトークンがネットワーク内（α-memory）に蓄えられる．その後，条件間でパターン変数（join 演算に用いられる場合には特に join 変数と呼ぶ）の値が矛盾しないかどうかを調べる条件要素間テスト（inter-condition test または join）が2入力ノード（two-input node）で順に行われ，テストを通過した WME の組（同じくトークンと呼ばれる）がネットワーク内（β-memory）に蓄えられていく．LHS のすべての条件を満たした WME の組（インスタンシエーション）は，RETE ネットワークの終端（terminal）に到達し，対応するルールを発火可能とする．RETE ネットワークの更新が完了すると，処理は競合解決フェーズに移行する．

2.2 トポロジカルな等価変換

図1は4個の条件要素からなるルールが取り得る join 構造を示している．and はまず内部の join を先に計算することを表している．join は基本的には交換則，結合則が成り立つ演算であるので，正の条件要素だけからなる join 構造はどのように変形してもよい．ただし，競合解決戦略として MEA[3] が用いられているとノード *a* の位置は変えることができない．この場合でも，ノード *b*，*c*，*d* については，どのような順序になってもよいので，結局24通りの（一般には，指数オーダの）join 構造が存在することになる．

負の条件要素は，その役割りがトークンをフィルタすることにあるので，join 構造のどこに位置してもよいというわけにはいかない．例えば，負の条件要素中に現れるパターン変数は，join 演算が行われる前に，正の条件要素により束縛されていなければならない．また，OPS 5 や PLANET 等のシステムでは，負の

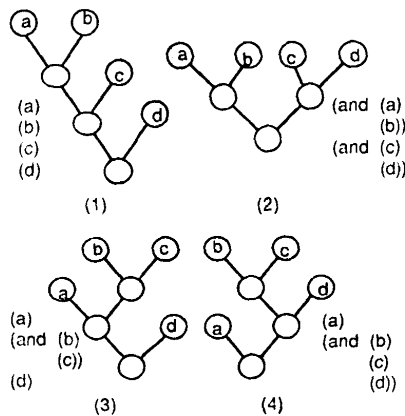


図1 種々の join 構造
Fig. 1 Variations in join structures.

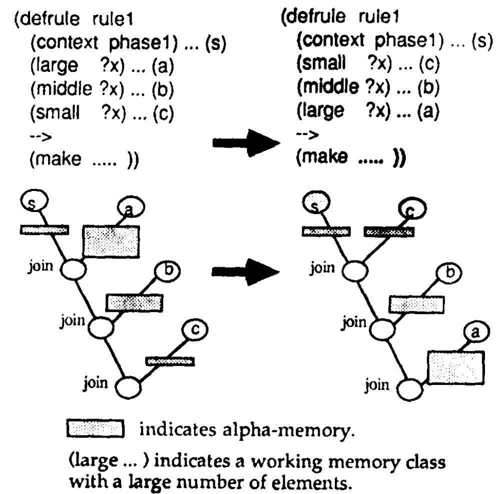


図2 ヒューリスティクス1の適用例
Fig. 2 Example of Heuristics 1.

条件要素は条件連接の先頭に位置することはできない．

以下では議論が複雑になるのを避けるため，等価変換のための細かな制限にはふれない．また，2進木状の join 構造のみを考える．例えば，図1のルールの条件部は，(a)(and (b)(c))(and (b)(d))と記述することもできる．この場合には，*b* は共通化されるので join 構造は2進木状にならないが，このような join 構造はアルゴリズムの対象外とする．

2.3 最適化ヒューリスティクス

最適化のためのヒューリスティクスには以下のものが知られている[2]．

(1) ヒューリスティクス1：データ量の小さな条件要素から join する．

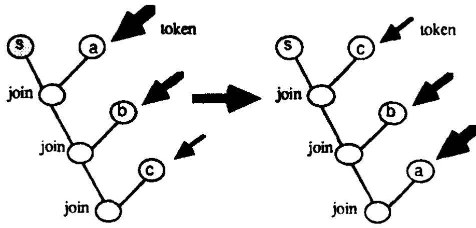
join 演算の実行順によって，計算途中で生成される中間データ（トークン）の総数が大きく異なる場合がある．後続する join 演算コストは，中間データの量に強く依存するので，演算結果のデータ数が小さい条件要素から join するのが望ましい．図2にヒューリスティクス1の適用例を示す．データ量の小さなクラスと照合する条件要素から順に join されるよう並び替えが行われている．（図2では，small，middle，large の順にデータ量が増大する．）

(2) ヒューリスティクス2：変更量の小さな条件要素から join する．

n 個の条件要素からなるルールを考える．いま，更新されたデータが，ルールの先頭に位置する条件要素と照合が成功すると，図1(1)の join 構造では $n-1$ 回の join 演算が実行されることになる．しかし，も

```
(defrule rule1
  (context phase1) ... (s)
  (frequently    ?x) ... (a)
  (occasionally ?x) ... (b)
  (seldom        ?x) ... (c)
  -->
  (make ..... ))
```

```
(defrule rule1
  (context phase1) ... (s)
  (seldom        ?x) ... (c)
  (occasionally  ?x) ... (b)
  (frequently    ?x) ... (a)
  -->
  (make ..... ))
```



indicates an amont of tokens.
(frequently ...) indicates a frequently cahnged working memory class.

図 3　ヒューリスティクス2の適用例
Fig. 3 Example of Heuristics 2.

しその条件要素がルールの末尾にあれば，join 演算は1回実行されるだけである．したがって，更新される頻度の小さな条件要素から順に join 演算を実行するのが望ましい．**図3**にヒューリスティクス2の適用例を示す．更新頻度の小さなクラスと照合する条件要素から順に join されるよう並び替えが行われている．(図3では，seldom, occasionally, frequently の順に更新頻度が増大する．)

(3) ヒューリスティクス3：join 構造を共有する．

2入力ノードを n 個のルールで共有すれば，各々の join 演算量は実質的に $1/n$ になる．したがって，可能な限り多くの共有が行われるよう，共通の join 演算をくくり出すことが必要である．(例えば，図1でノード c, d の join を他のルールと共有するためには，(2)の構造をとる必要がある．) **図4**にヒューリスティクス3の適用例を示す．class-a と class-b の join 演算を他の join 演算と独立に実行することによって，2個のルールにまたがる join 演算の共有を実現している．

上記の3種のヒューリスティクスは独立ではない．したがって，3種のヒューリスティクスを考慮しながらも総コストが最小となる解を求める必要がある．

3. コストモデル

3.1 パラメータ

図5に join 演算のコストモデルを示す．以下ではノード n を下限とする join 構造を「n の join 構造」と呼ぶ．例えば，図5はノード c の join 構造を表している．各ノードには以下に示す5種のパラメータが定義される．

```
(defrule rule1
  (context phase1) ... (s1)
  (class-a ?x) .............. (a)
  (class-b ?x) .............. (b)
  (class-c ?x) .............. (c)
  -->
  (make ..... ))

(defrule rule2
  (context phase2) ... (s2)
  (class-a ?x) .............. (a)
  (class-b ?x) .............. (b)
  -->
  (make ..... ))
```

```
(defrule rule1
  (context phase1) .... (s1)
  (and (class-a ?x) ..... (a)
       (class-b ?x)) ... (b)
  (class-c ?x) .............. (c)
  -->
  (make ..... ))

(defrule rule2
  (context phase2) .... (s2)
  (and (class-a ?x) .... (a)
       (class-b ?x)) ... (b)
  -->
  (make ..... ))
```

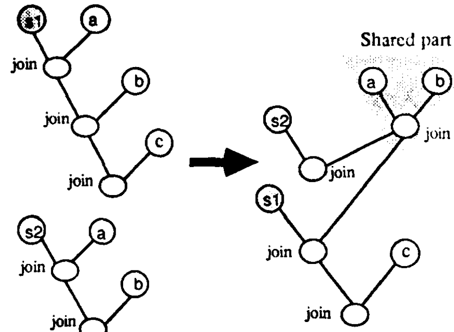
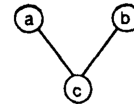


図 4　ヒューリスティクス3の適用例
Fig. 4 Example of Heuristics 3.



When b is a positive condition element:
Ratio(c) = Ratio(b)
Test(c) = Token(a) * Memory(b) + Token(b) * Memory(a)
Token(c) = Test(c) * Ratio(c)
Memory(c) = Memory(a) * Memory(b) * Ratio(c)
Cost(c) = Cost(a) + Cost(b) + Test(c)

When b is a negative condition element:
Ratio(c) = Ratio(b)
Test(c) = Token(a) * Memory(b) + Token(b) * Memory(a)
Token(c) = Token(a) * Ratio(c)
Memory(c) = Memory(a) * Ratio(c)
Cost(c) = Cost(a) + Cost(b) + Test(c)

図 5　コストモデル
Fig. 5 Cost model.

① Token(n)：ノード n から後続ノードに送られるトークンの総数である．

② Memory(n)：ノード n の α-memory，または β-memory に保持されるトークンの平均値である．

③ Test(n)：ノード n で実行される条件要素間テストの総数である．到達する1個のトークンとメモリ中の1個のトークンとの join 演算を1回の条件要素間

テストと数えている.

④ Cost(n):ノード n の join 構造内で実行される条件要素間テストの総コストである.コスト関数は後述する.

⑤ Ratio(n):ノード n で条件要素間テストが成功する確率である.

最適化が実行される前に,プロダクションルールは一度実行され,その測定結果を基に1入力ノードのパラメータが決定される.最適化のプロセスでは,数多くの2入力ノードが生成評価されるが,それらのパラメータは後述する計算式を用いてその都度計算される.

3.2 1入力ノードのパラメータ

いま,ノード b を1入力ノードとしよう.1入力ノードでは join は計算されないため,Test(b),Cost(b) は常に0である.他のパラメータは以下のように決定される.

(1) Token(b)

ノード b が送出したトークンの総数が測定され設定される.測定値はプログラムの動作特性にのみ依存し,join 構造には依存しない.したがって,どのような join 構造となっても Token(b) の値はそのまま有効である.

(2) Memory(b)

α-memory 中のデータ数の平均のとり方には様々な方法が考えられる.ここではプロダクションサイクルごとに,α-memory に保持されているトークン数を測定しその平均値をとっている.Momory(b) も join 構造には依存しない.

(3) Ratio(b)

すべての2入力ノードでの join の成功確率が測定される.次に,ノード b を右入力とする2入力ノードの成功確率が Ratio(b) に設定される.この値は測定時の join 構造に依存する.したがって,join 構造が変化したときにはこの値は近似値を与えるにすぎない.後述する最適化アルゴリズムでは種々の join 構造のコストがこの値を用いて評価されるので,より正確な近似値を得るために以下の手法を用いている.

① join 変数が0個の時に Ratio=1.0 である.一般に join 変数の個数が増えるに従って Ratio は小さくなる傾向にある.そこで,Ratio は join 変数の個数ごとに記録する.例えば,1入力ノードが複数のルールで共有されている場合には,そのノードを右入力とする2入力ノードが複数個存在し,それぞれ join 変数の個数が異なることがある.このような場合には,join 変数の個数別に Ratio を計測記録する.

② 1入力ノードでの Ratio の平均を,join 変数の個数ごとに計算しておく.測定値が得られなかった1入力ノードの Ratio には,join 変数の個数に従ってあらかじめ計算した平均値を設定する.

```
one   (context phase1)
  |      token: 1   memory: 0.5
  |------ one   (large ?x)
  |               token: 8   memory: 4.0
!two   token: 8   memory: 2.0  test: 8
  |      * one  (small ?x)
  |        |      token: 4   memory: 2.0
  |        |---- * one   (middle ?x)
  |        |               token: 6   memory: 3.0
  |----- * two  token: 12   memory: 3.0   test: 24
two   token: 24  memory: 3.0  test: 48
terminal  instantiation: 24

one:  one-input node
two:  two-input node
! : inter-condition test with no join variables
* : shared node
```

図 6 計測データの表示例
Fig. 6 Example of displaying statistics.

PLANET では RETE ネットワークの各ノードにパラメータの値を格納するエリアを設け,トークンの到着や join 演算の実行を契機として値の更新を行っている.この計測に要するオーバヘッドは,プロダクションシステムの実行に要する時間の5%以下にすぎないため,PLANET では計測を常時行っている.したがって,任意の時点で**図6**に示すような測定結果の表示を得ることができる.

3.3 2入力ノードのパラメータ

2入力ノード(図5では c)のコストモデルは右入力の1入力ノード(図5では b)が,正の条件要素であるか,負の条件要素(not)であるかによって異なってくる.図5は,それぞれの場合の計算式を示している.

(1) Test(c)

トークンが左方向から入力される場合には,c に到達したトークン数(Token(a))と右入力のメモリ内のトークン数(Memory(b))を掛け合わせた数(Token(a)*Memory(b))の条件要素間テストが行われる.トークンが右入力される場合にはその反対(Token(b)*Memory(a))である.したがって,両者を加えたもの(Token(a)*Memory(b)+Token(b)*Memory(a))が,このノードで実行される条件要素間テスト回数(Test(c))となる.

(2) Token(c)

テストの結果生成されるトークン数(Token(c))

は，右入力が正の条件要素の場合には条件要素間テスト回数（Test(*c*)）にテストの成功確率（Ratio(*c*)）を掛けた値（Test(*c*)∗Ratio(*c*)）である．一方，右入力が負の条件要素の場合には，左入力のトークン(Token(*a*)))がフィルタされた結果の値（Token(*a*)∗Ratio(*c*)）となる．

（3） Memory(*c*)

メモリ中の平均的なトークン数（Memory(*c*)）は右入力が正の条件要素の場合には，左右のノードのメモリ内の平均トークン数を掛け合わせた値（Memory(*a*)∗Memory(*b*)）に条件要素間テストの成功確率(Ratio(*c*)）を掛けたもの（Memory(*a*)∗Memory(*b*)∗Ratio(*c*)）と考える．一方，負の条件要素を右入力とする場合には，左入力のメモリ（Memory(*a*)）をフィルタした結果の値（Memory(*a*)∗Ratio(*c*)）とする．

（4） Cost(*c*)

左右のノードのコスト（Cost(*a*), Cost(*b*)）に自ノードの計算コストを加えたものである．条件要素間テストのコストは一般に $A*\mathrm{Test}(c)+B*\mathrm{Token}(c)$ で表すことができる．A, B は適当な定数である．本論文は最適化目標を明確に設定するために，条件要素間テストの総数を最小化することとし，$\mathrm{Cost}(c)=\mathrm{Cost}(a)+\mathrm{Cost}(b)+\mathrm{Test}(c)$（すなわち，$A=1$, $B=0$）とする．

しかし一般には，定数 A, B は処理系に合わせて適切に設定する必要がある．例えば，OPS 5 等，2重ループで join 演算を実行している処理系では，A が大きいと考えられる．一方，データがハッシュされていれば[5]，むしろ条件要素間テストによって生成されるトークン数が問題となる．このような場合には，例えば $\mathrm{Cost}(c)=\mathrm{Cost}(a)+\mathrm{Cost}(b)+\mathrm{Token}(c)$（すなわち，$A=0$, $B=1$）とするなど，B を大きくする必要がある．

（5） Ratio(*c*)

条件要素間テストの成功確率（Ratio(*c*)）はどのパターン変数が join 変数となるか，および join 演算対象のデータにどのような相関関係があるかに依存し，正確な値を得ることは困難である．そこで近似値として，右入力の1入力ノードに登録された成功確率(Ratio(*b*)）のうち，join 変数の個数が一致するものを用いることとする．一致するものがなければ，あらかじめ計算した成功確率の平均値を設定する．

4. 最適化アルゴリズム

4.1 アルゴリズムの概略

既に述べたように，プロダクションシステムの最適化では，独立でない要求が絡み合っているため，特定のヒューリスティスクを適用しても必ずしも良い join 構造を得ることはできない．例えばあるヒューリスティクスを用いた結果，join 演算の共有度が下がり，全体の性能がむしろ低下する場合等が生じる[2]．一方，可能な join 構造は指数オーダで存在するため，単純な生成・評価手法（generate and test）でこの問題を解くことはできない．そこで以下では，生成・評価手法をベースに，生成される join 構造を種々の制約を用いて削減するアプローチをとる．アルゴリズムの概略を図7に示す．要点は以下のとおりである．

① ルールは事前に実行させて計測したコスト（条件要素間テストの総数）が最も大きなものから最適化する．これはルール間にまたがる最適化で，最大の自由度をコストの高いルールに保証するための措置である．ルール間の最適化については4.3で述べる．

② ルールの最適化を開始する前に，各ルールのノードリスト（node-list）に，そのルールの条件要素と対応する1入力ノードと，計算済み2入力ノード（pre-calculated 2-input node：後述）を登録する．計算済み2入力ノードは join 構造の削減と，join 演算の共有化を促進するのに用いられる．

③ 各ルールの最適化プロセスでは，ノードリスト中

```
clear the rule-list(R);
push all rules to R;
sort R in descending order of cost;
for r from the first rule to the last rule of R;
  clear the node-list(N);
  push all one-input nodes of r to N;
  let k be the number of one-input nodes;
  append pre-calculated two-input nodes to N;
  for i from the 2nd node to the last node of N;
    for j from the 1st node to the i-1th node of N;
      if all constraints are satisfied then do;
        create a two-input node n to join i and j;
        calculate parameters of n;
        push n to just after the max(i,k)th node of N
      end
    end
  end
  find the lowest-cost complete join structure;
  generate an optimized version of r
end
```

図7 最適化アルゴリズムの概略
Fig. 7 Outline of the optimization algorithm.

の2個のノードを組み合わせてそれらの join を行うノードを次々に生成する.（2進木状の join 構造を対象とするため，この処理は必ず終了する.）新しく生成された2入力ノードは優先的に次の join 対象となる．これは，ルールの条件部全体を構成する join 構造を可能なかぎり早期に生成するための措置である．この間，4.2 で述べる制約が組合せを削減するために用いられる．

④ すべての2入力ノードの生成が終了した時点で，ルールの条件部全体を構成する join 構造のうち，最もコストの低いものを選択する．

4.2 組合せ削減のための制約

生成される join 構造を削減するための制約としては以下のものを用いている．

（1） コスト制約（Minimal-Cost Constraint）

既に生成されている join 構造よりコストが高い構造を生成しないための制約である．ノード n に至るまでに join された1入力ノードの集合を Conditions(n) で表す．もし n が1入力ノードであれば Conditions(n)$=\{n\}$ とする．いま，ノード n に対して，

$$^{\exists}m \in \text{node-list}$$
$$\text{such that Conditions}(n) \subseteq \text{Conditions}(m),\ \text{and}$$
$$\text{Cost}(n) \geq \text{Cost}(m)$$

であればノード n は生成しない．逆にもし n が既にノードリスト中に存在し，その後 m が生成されたのであれば，n をノードリストから除去し，m で置き換える．このコスト制約は最適解を保証する．

コスト制約を有効に生かすには，1入力ノードを多く含む低コストの join 構造を早期に生成すればよい．PLANET では1つの手段として，最適化前のルールの join 構造に含まれるノードを，計算済み2入力ノードとしてあらかじめ登録するようにしている．これによって，少なくとも最適化前の（すなわち，利用者の記述した）ルールよりもコストが高い join 構造の生成を防ぐことができる．図8にコスト制約の例を示す．

（2） 連結制約（Connectivity Constraint）

join 変数が無い場合には，条件要素間テストは到達したトークンとメモリ中のトークンのすべての組合せを生成する．連結制約はこのような事態を避けるための制約である．より形式的には次のように定義できる．いま p と q を1入力ノード，Variables(n) を Conditions(n) 中のパターン変数としよう．ここで，もし以下の両方の条件が成立すれば，n と m を join する2入力ノードは生成しない．

（i） $\text{Variables}(n) \cap \text{Variables}(m) = \phi$, and

（ii） $^{\exists}p,\ q \notin \text{Conditions}(n) \cup \text{Conditions}(m)$ such that

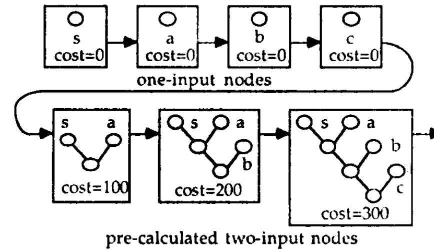
$$\text{Variables}(n) \cap \text{Variables}(p) \neq \phi,\ \text{and}$$
$$\text{Variables}(m) \cap \text{Variables}(q) \neq \phi$$

図9に連結制約の例を示す．連結制約は a と c を join する2入力ノードの生成を抑止する．なぜなら，a と c には join 変数が存在せず（(i)が満足される），かつ a と b，または b と c の join 演算を先に行

<Production Rule>

```
(defrule rule1
  (context phase1) ... (s)
  (class-a ?x) ............ (a)
  (class-b ?x) ............ (b)
  (class-c ?x) ........... (c)
  -->
  (make ..... ))
```

<Node-List>



<Minimal Cost Constraint>

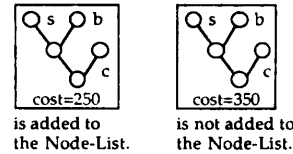


図 8　コスト制約の例
Fig. 8 Example of minimal-cost constraint.

<Production Rule>

```
(defrule rule1
  (context phase1) ... (s)
  (class-a ?x ?y) ...... (a)
  (class-b ?y ?z) ...... (b)
  (class-c ?z ?w) ...... (c)
  -->
  (make ..... ))
```

<Connectivity Constraint>

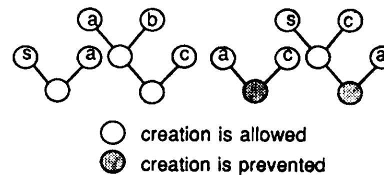


図 9　順位制約の例
Fig. 9 Example of connectivity constraint.

うことによって，a と c の join を避けられる可能性がある（(ii)が満足される）からである．一方，s と a の join 演算は抑止しない．（この点が従来の connectivity heuristics[15] と異なっている．）s と a との間には，join 変数は存在しないが，s は何らかのノードと遅かれ早かれ join 変数なしで join 演算を行わざるを得ないからである（すなわち，(ii)が満足されない）．

（3） 順位制約（Priority Constraint）

動作特性を用いて，1入力ノードに優先順位をつけることができたとしよう．順位制約は優先度の高いノードと join が可能な場合に，優先度の低いノードとの join を実行する2入力ノードの生成を抑止する．より形式的には次のように定義できる．いま p と q を1入力ノード，p が q より優先順位が高いことを $p \gg q$ で表す．順位制約は以下の場合にノード n，m を join する2入力ノードの生成を抑止する．

$^{\exists}p \notin$ Conditions $(n) \cup$ Conditions (m),

$^{\exists}q \in$ Conditions (m)

such that

（i） $p \gg q$, and

（ii） Variables $(n) \cap$ Variables $(p) \neq \phi$, or
Variables $(n) \cap$ Variables $(m) = \phi$

(ii)は n と m の join が順位制約によって抑止されているにもかかわらず，n と p の join が連結制約によって抑止されることを防ぐための条件である．5章で行う評価では，Token$(p) \geq$ Token(q) かつ Memory$(p) \geq$ Memory(q) の場合に $p \gg q$ と定義している．図10に順位制約の例（$m=q$ の場合）を示す．従来の cheapest-first heuristics[15] との違いは順位が部分的にしか定義されないところにある．すなわち，順位は直接解を決定するのに用いられるのではなく，順位を守らない解を排除するための制約として用いられている．

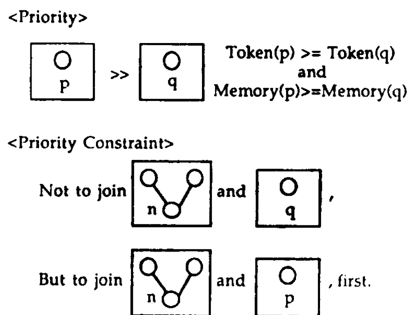


図 10 順位制約の例
Fig. 10 Example of priority constraint.

なお，連結制約と順位制約は大きな効果を持つ制約であるが，必ずしも最適解を保証しないことに注意する必要がある．

4.3 複数のルールにまたがる最適化

複数のルールによって joni 演算を共有できれば，ルール当りの演算コストは低下する．そこで，ルール間の共有を促進するために，以下の方法をとる．

① 2入力ノード n を生成する時点では，n が可能な限りのルールで共有されることを仮定する．すなわち，Conditions(n) を条件部に含むすべてのルールによって n が共有されると仮定する．さらにこの仮定に基づいて，Cost(n) を再計算する．すなわち Cost(n) は，本来のコストをそのノードを共有し得るルールの数で割ったものとする．

② ルールを最適化する時点では，既に作成された2入力ノードはコスト0で利用できると仮定する．すなわち，最適化処理済みのルールに含まれる2入力ノードを計算済み2入力ノードとして登録する．計算済みノードのコストには0を設定する．

上記の手法を用いることによって，組合せ爆発を引き起こすことなく複数のルールにまたがる最適化が可能となる．ルールは1個ずつ最適化されるが，あたかもすべてのルールが一括して最適化されたような効果が得られる．

5. 評　　価

5.1 最適化アルゴリズムの評価

前節で述べた最適化アルゴリズムを適用して，PLANET 上に最適化機能を試作し，既存のエキスパートシステムに適用した．最適化機能は，利用者が記

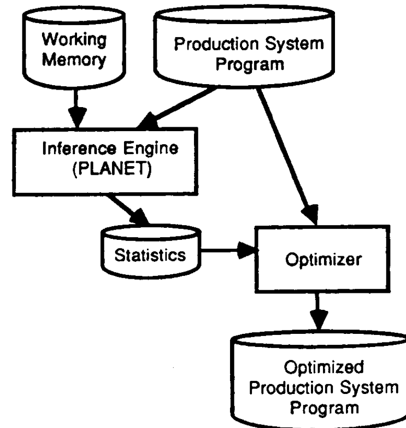


図 11 最適化システムの構成
Fig. 11 System configuration.

述したプログラムと動作特性の計測結果を入力とし，最適化されたプログラムを出力する．システム構成図を図11に示す．適用したプログラムは，論理回路の最適化を行うエキスパートシステム[9]で107ルールから構成されている．論理回路としてはWME数が300〜400の比較的小規模なものを用いた．

このプログラムを評価対象として選択した理由は2つある．第1に，このプログラムは条件要素数が大きな（20以上）ルールを多数含んでいる．したがって，制約を用いて組合せ爆発を防ぐという本論文のアプローチにとっては簡単なプログラムではない．第2に，このプログラムはかつてエキスパートシステム開発者自身の手によって join 構造の改善が図られている．この改善は人手によって，2〜3日を要して行われている．したがって，最適化アルゴリズムの能力を人間のそれと比較することができる．

以下，主要モジュール（33ルール）を最適化した結果を述べる．最適化は人手改善前のプログラムを対象として行っている．なお，この評価は Symbilics ワークステーションの Common Lisp 上で行ったものである．

（1） 最適化効果

最適化の効果を表2に示す．条件要素間テストの総数が1/3に，CPU 性能が約2倍に向上している．特筆すべきことは，最適化機能が多くのルールでエキスパートシステム開発者自身による性能改善を凌ぐ効果を導き出していることである．表中，＊印を付けたルールでは，最適化後の条件要素間テスト回数が最適化前を上回っている．一般にこのような失敗は，コストモデルによって推定した処理コストと，実際に走行させた処理コストが完全には一致しないことが原因となって生じる．しかし，表2の場合に限っていえば，必ずしも最適化に失敗している訳ではない．例えばルール1とルール14は多くの join 演算を共有している．2ルールの条件要素間テスト回数の和を比較すると，むしろ最適化効果が現れていることが分かる．

一方，最適化に要する処理時間は展開ノード数の2乗に比例する．評価対象のエキスパートシステムには，条件数が20を越える大型のルールが多数存在するため，これらの制約を用いても現在最適化に10分以上を要している．しかし，条件要素数が5程度の通常のプログラムでは，数分で最適化が完了する．実用に供するには，処理系の性能を上げると共に，特に展開ノード数が多くなるルールについて，最適化を途中で切り上げるなどの現実的な解決策をとることが必要である．

表2 最適化の効果
Table 2 Effectiveness of optimization.

ルール	条件	最適化前	最適化後		入手改善後
番号	要素	テスト回数	テスト回数	ノード数	テスト回数
1	21	46,432	22,396	179	47,888
2	18	29,548	494	198	27,244
3	17	29,548	494	92	27,244
4	22	25,513	144	236	7,813
5	21	25,313	144	94	7,813
6	18	10,322	3,749	552	3,749
7	17	10,322	3,749	194	3,749
*8	15	9,966	12,539	228	9,966
9	7	9,830	1,180	99	1,180
10	6	9,278	630	20	630
11	17	8,566	4,640	116	8,566
12	7	7,656	760	44	520
13	6	7,544	648	22	408
*14	23	3,160	13,780	76	4,616
15	11	1,918	1,865	119	1,918
16	20	1,336	1,325	413	1,336
17	19	845	834	97	845
*18	19	814	2,792	68	814
19	12	525	467	200	525
20	16	472	348	284	472
21	11	403	345	61	403
22	15	348	225	81	348
23	12	310	286	159	310
*24	16	252	296	206	252
25	11	202	178	64	202
26	20	181	181	304	181
27	19	181	181	109	181
*28	15	136	180	94	136
29	23	67	39	175	67
30	24	67	39	104	67
31	2	46	46	3	46
32	2	23	23	3	23
33	2	5	5	3	5
計33 ルール	平均 14.2	241,329 (1.00)	75,002 (0.53)	平均 131.8	159,517 (0.69)

- （ ）内は PLANET での CPU 時間比．
- ルール間で共有されている条件要素間テストの回数は，共有されているルール数で割った後，加算している．

（2） 制約の効果

表2の結果は，4.2 で提案した3種の制約をすべて用いて得たものである．以下では，個々の制約がそれぞれどのような効果を発揮したかを，より詳細に考察する．

制約のメリットは最適化処理の性能向上として現れる．一方，デメリットは最適化効果の減少である．既

表 3 制約の効果
Table 3 Effectiveness of constraints.

ルール番号	条件要素	2種の制約を使用(コスト制約+連結制約)		1種の制約を使用(コスト制約)	
		テスト回数	ノード数	テスト回数	ノード数
1	21	19,621	680	19,621	1,370
2	18	494	438	494	2,448
3	17	494	162	494	197
4	22	136	887	N/A	>3,000
5	21	136	139	N/A	209
6	18	376	>3,000	N/A	>3,000
7	17	376	160	N/A	>3,000
8	15	10,407	514	10,407	856
9	7	118	99	1,180	121
10	6	630	20	630	22
11	17	4,194	383	4,212	663
12	7	761	44	761	68
13	6	649	22	649	24
⋮	⋮	⋮	⋮	⋮	⋮

- 順位制約は条件要素が10を越えるルールで用いている.
- N/Aは展開ノード数が3,000を越えたため，あるいは越えたルールとの共有ノードが多いため，結果が得られなかったことを示す.

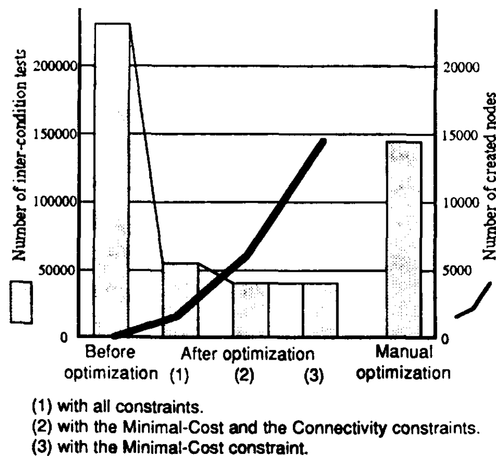


図 12 制約の効果と損失
Fig. 12 Merits and demerits of constraints.

に述べたようにコスト制約は最適解を保証するので，問題は連結制約と順位制約によりどの程度最適化効果が減少するかということになる.

表3に，2種の制約（コスト制約と連結制約）を用いた場合と，1種の制約（コスト制約）を用いた場合の評価結果を示す．計算量が多い上位13ルールの条件要素間テスト回数と展開ノード数を表示している．また，**図12**には表2と表3をまとめた結果を示す．展開ノード数が3,000を越え，処理が打ち切られた場合には，展開ノード数は3,000と数えている．またこの場合の条件要素間テスト回数は，処理打ち切り前に結果が得られていればその値，そうでなければすべての制約を用いて得られた最良の結果と同一になると仮定して計算している.

この評価結果から以下のことがいえる.

① 条件要素間テスト回数は，3種の制約を用いた場合に比べ，2種の制約を用いた場合の方が，3割程度少ない．この値は1種の制約を用いた場合（すなわち，コストモデル上の最適解）とほぼ同一である．(むしろ，1種の制約を用いた場合の方が実際の処理コストが僅かに増大している．これはコストモデルに基づく推定に誤差があるためである.）この結果から，順位制約の使用を制限する方針をとる必要があることが分かる．現在，順位制約は条件要素数が10を越えるルールでのみ用いることにしている.

② 一方，展開ノード数は，2種，1種の制約を用いた場合は，3種の制約を用いた場合に比べそれぞれ，3倍，7.2倍に増大している．処理時間は展開ノード数の2乗に比例するので，計算上はそれぞれ9倍，52倍に増大することになるが，実際にはこの程度ではすまない．1種の制約を用いた場合には，ガーベジコレクションが頻発し最適化に約5日を要した．この事実は，各種の制約が最適化処理時間の短縮に大きな役割を果たすことを示している.

5.2 他の最適化方式との比較

（1） データベースにおける問合わせ最適化

これまでに，データベースの問合わせ最適化に関する研究が数多く報告されている．join 構造の最適化に関連する研究としては，以下のものがある.

① Jarke[10] は多数の問合わせを一括して最適化する方式（Multiple Query Optimization）を提案している．これは本論文で提案する多数のルールの一括最適化とよく似た問題である.

② Kim[11]はさらに，プログラム中で類似の問い合わせが繰り返し実行される場合に，それらを一括して最適化する方式(Global Query Optimization)を提案している．プロダクションシステムの動作は例えば，

```
LOOP;
  Q1 ;……ルール1の条件照合
  Q2 ;……ルール2の条件照合
  ⋮
  Update database;
END
```

で近似できるので，Global Query Optimization はプロダクションシステムの最適化と極めて近い問題であ

るということができる.

しかしこれらの研究は, 同時に実行される問合わせの最適化が目的であり, プロダクションシステムのように繰り返し同じ問合わせが実行されることは想定していない. したがって, RETE アルゴリズムのように, 既に行われた問合わせの結果を保存し再利用するという発想は生まれてこない. データベースでは最適化にデータ特性 (データ数) を用いるが, プロダクションシステムの最適化ではそれに加えてプログラムの動作特性 (データ更新回数) を用いるのは, この相違に基づくものである.

(2) 問題解決における問合わせ最適化

問題解決の分野では Smith[15], Warren[16] らによって, 条件の連接により構成される問合わせの最適化が研究されている. データベースの最適化同様, プログラムの動作特性が考慮されていない点がプロダクションシステムの最適化と異なっている. 本論文で提案した各種制約はこれらの文献で提案されているヒューリスティクスと以下に示すような関係がある.

① Connectivity heuristics[15] は, join 変数に着目して問合わせを独立な部分に分解し, 性能のよい join 構造を得る手法である. 本論文の連結制約はこのヒューリスティクスを基にしている. ところがプロダクションシステムにこの方法を適用すると, かえって性能が劣化する場合がある. すなわち, プロダクションシステムのように動作特性を考慮しなければならない場合には, 必ずしも良い join 構造を導くことができない. 例を図 13 に示す. そこで, 本論文では connectivity heuristics の条件を緩めて, 生成・評価すべき join 構造を削減するための連結制約として用いている.

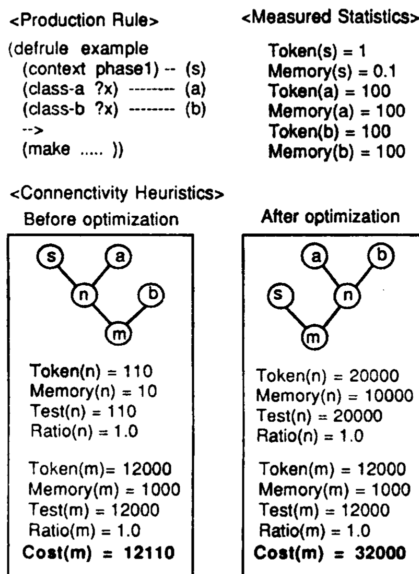


図 13 Connectivity heuristics の適用失敗例
Fig. 13 Example of connectivity heuristics.

② Cheapest-first heuristics[15] はコストのかからない連接から順に計算する手法である. 本論文の順位制約はこの手法に基づいている. しかし, この手法は問合わせ最適化においてさえ常に有効とは限らないことが明らかとなっている. そこで本論文では, ごく限られた条件下の順位付けを行い, この順位を, 評価すべき join 構造を削減するための制約として用いている.

(3) プロダクションルールの最適化

TREAT[2] では join 構造は seed-ordering と呼ばれる方法を用いて実行時に最適化される. そこでは, 変更を受けた α-memory に対応する条件要素が最初に join の対象となる. 残りの条件要素の join 順序はプログラムの記述順である. このように, 実行時の最適化ではオーバヘッドが無視できないためそれほど凝った手法をとることはできない.

一方, SOAR では, Smith らによって提案されたヒューリスティクスをプロダクションシステムの最適化に直接適用している. SOAR は OPS 5 をベースにしたプロダクションシステムである. SOAR にはチャンキング (chunking) と呼ばれる学習機能があって, 推論の実行過程でルールが生成される. このルールは一般に条件要素数が多いため, 生成時に最適化することが試みられている[13]. この場合にも推論実行中の最適化であるため簡単なヒューリスティクス (主として connectivity heuristics や SOAR の言語仕様に存在した手法) が用いられているにすぎない. 既に述べたようにプログラムの動作特性を考慮せずに, 問合わせ最適化で得られたヒューリスティクスをそのまま適用しても良い結果は得られない.

6. むすび

プロダクションルールの条件部の join 構造を最適化するアルゴリズムとその効果について述べた. 評価対象としたエキスパートシステムは比較的小規模なものであるが, join 演算のコストが 1/3 に削減されるという大きな効果が得られた. 特筆すべきことは, 過去に行われたエキスパートシステム開発者自身の手による最適化を凌ぐ効果が得られたことである.

一般にルール数，データ量が増えれば最適化の効果は増大する．最近の商用ツールの中には，join 演算の実行順を利用者に指定させるものも現れてきている．しかし，性能改善のためのヒューリスティクスが競合するため，利用者による最適化は容易な作業ではない．したがって，エキスパートシステムの実用化が進むにつれ，最適化機能はその有効性を増すものと考えている．

また，プロダクションシステムの性能向上を目的として並列条件照合[5]や，並列実行[7]に関する研究が盛んに行われている．これらの多くは RETE アルゴリズムを前提としたものであるので，ここで提案した最適化アルゴリズムは並列処理環境でも有効である．

謝辞 日頃御指導いただく NTT 情報通信処理研究所中野良平主幹研究員，評価用プログラムを提供していただいた石川雄三主幹研究員，ならびに討論いただいた桑原和宏氏，横尾真氏に感謝いたします．また，データベースの観点からコメントをいただいた神戸大学田中克己助教授，京都産業大学吉川正俊博士に感謝します．

参考文献

1) Brownston, L., Farrell, R., Kant, E. and Martin, N.: *Programming Expert System in OPS 5: An Introduction to Rule-Based Programming*, Addison-Wesley, Menlo Park, Ca. (1985).
2) Clayton, B. D.: ART Programming Tutorial, Volume Three: Advanced Topics in ART, Version 3.0, Inference Corp. (1987).
3) Forgy, C. L.: OPS 5 User's Manual, CS-81-135, Carnegie Mellon University (1981).
4) Forgy, C. L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artif. Intell.*, Vol. 19, pp. 17-37 (1982).
5) Gupta, A., Forgy, C. L., Kalp, D., Newell, A. and Tambe, M.: Results of Parallel Implementation of OPS 5 on the Encore Multiprocessor, CS-87-146, Carnegie Mellon University (1987).
6) 石田 亨：データ駆動型プロダクションシステムによる意味ネットワークの探索，情報処理学会論文誌，Vol. 28, No. 10, pp. 1021-1030 (1987).
7) Ishida, T. and Stolfo, S. J.: Towards the Parallel Execution of Rules in Production System Programs, *International Conference on Parallel Processing*, pp. 568-575 (1985).
8) Ishida, T.: Optimizing Rules in Production System Programs, *AAAI-88*, pp. 699-704 (1988).
9) 石川，仲西，中村：論理ゲート最適化エキスパートシステムについて，第 34 回情報処理学会全国大会論文集，pp. 1391-1392 (1987).
10) Jarke, M.: Common Subexpression Isolation in Multiple Query Optimization, in *Query Processing in Database Systems*, Kim, W., Reiner, D. and Batory, D. Eds., pp. 191-205, Springer, New York (1984).
11) Kim, W.: Global Optimization of Relational Queries: A First Step, in *Query Processing in Database Systems*, Kim, W., Reiner, D. and Batory, D. Eds., pp. 206-216, Springer, New York (1984).
12) Miranker, D. P.: TREAT: A Better Match Algorithm for AI Production Systems, *AAAI-87*, pp. 42-47 (1987).
13) Scales, D. J.: Efficient Matching Algorithms for the SOAR/OPS5 Production System, STAN-CS-86-1124, Stanford University (1986).
14) Schor, M. I., Daly, T. P., Lee, H. S. and Tibbitts, B. R.: Advances in RETE Pattern Matching, *AAAI-86*, pp. 226-232 (1986).
15) Smith, D. E. and Genesereth, M. R.: Ordering Conjunctive Queries, *Artif. Intell.*, Vol. 26, pp. 171-215 (1985).
16) Warren, D. H. D.: Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *7th VLDB*, pp. 272-281 (1981).

(昭和 62 年 11 月 26 日受付)
(昭和 63 年 10 月 7 日採録)

石田　亨（正会員）

昭和 28 年生．昭和 51 年京都大学工学部情報工学科卒業．昭和 53 年大学院修士課程修了．同年日本電信電話公社に入社．昭和 58～59 年，コロンビア大学客員研究員．現在，NTT 情報通信処理研究所知識処理研究部勤務．プログラミング環境，知識ベースシステム，並列処理，分散処理に興味を持つ．電子情報通信学会，人工知能学会，ソフトウェア科学会，AAAI 各会員．