論　文

ハッシュ技術を用いた集合関数の処理法*

西原清一** 萩原宏***

Abstract

Executing set functions is one of the basic techniques in the fields of information retrieval, data structure and data base management.

In this paper, it is shown that hashing techniques can effectively be applied to processing set functions, where each set appearing in set functions is a set of keys. Each entry of a hash table contains a key field, a pointer field and a match level indicator field. The last field is used to indicate how well the key satisfies the set function under consideration.

Some algorithms to execute set functions containing no complementary set are given and the efficiency is proved by some experiments.

1. まえがき

最近のデータ管理技術においては，複数個のファイルを統一的に扱うことが1つの目的とされている．ファイルの統一化によって，情報の重複保管による効率低下や不整合の発生などを回避することができるからであるが，さらに複数個のファイルにわたって関係するような処理要求にも応ずることができるという側面がある．データ・ベース用言語の中にもそのような機能を含んでいるものが多く[1]，これらの機能は複数個のデータ（レコード）集合に対する集合演算を基本的に含んでいる．例えば2つのデータの集合がそれぞれある条件を満たしているとすれば，それらの共通部分は2つの条件を同時に満たすデータの集合となる．

また情報検索などデータ構造を取り扱う他の分野でも集合演算が必要とされる場合が多い[2]．とくに複数の利用者から同時に処理要求が生じているような状況では，通常，集合演算は順次1つずつしか処理されないので，他の要求は全て待たされるという結果になる．データ・ベースへの問い合わせがオンライン化されつつある現状では，とくに集合演算の処理なども迅速に行われることが望まれる．

本稿は，上記のような種々のデータ構造処理の基本技術のひとつとして，集合演算の一実現方法について述べたものである．まず **2.** では，加法標準形の集合関数をハッシュ表を用いて処理するアルゴリズムを与える．**3.** では，簡単な実験例によって評価を行う．最後に **4.** では，一般の集合関数への拡張の試みとして，否定（補集合）を含まない一般の関数の処理方法について述べる．

2. Hash 表上での集合関数（加法標準形）の処理法

2.1 諸定義および簡単な例

加法標準形の集合関数の処理法について述べるに先立って，本稿で用いる諸用語の定義を行い，問題を明確にしておこう．集合関数の表現式（以後，集合式という）に現われる集合を S や S_i $(i=1,2,\cdots)$ などで表わす．各集合は有限個の要素（以後，キーという）を含んでおり，また S_i 内のキーをつぎつぎと繰り返しなしに取り出す操作ができるものとする．ただしキーをとり出す順序にはとくに制約はない．S_i に含まれるキーの個数を card (S_i) で表わす．集合 S_i と S_j の共通部分 (i.e.$\{K|K\in S_i$ かつ $K\in S_j\}$) を $S_i\cdot S_j$ で，また和集合 (i.e.$\{K|K\in S_i$ または $K\in S_i\}$)

---

* Executing Set Functions by Using Hashing Techniques by Seiichi NISHIHARA (Institute of Electronics and Information Science, University of Tsukuba) and Hiroshi HAGIWARA (Faculty of Engineering, Kyoto University).

** 筑波大学電子・情報工学系
*** 京都大学工学部情報工学科

を S_i+S_j で表わす．さらに，

$$\bigcap_{i=1}^{m} S_i=S_1\cdot S_2\cdot\cdots\cdot S_m,\quad \bigcup_{i=1}^{m} S_i=S_1+S_2+\cdots+S_m$$

とする．加法標準形とは，

$$\bigcup_{i=1}^{m}\bigcap_{j=1}^{n(i)} S_{ij}=S_{11}\cdot S_{12}\cdot\cdots\cdot S_{1n(1)}+\cdots+S_{m1}\cdot S_{m2}\cdot\cdots\cdot S_{mn(m)} \qquad (1)$$

の形をした集合式のことである*．演算子は・が＋に優先する．(1)式において，各共通部分の第1番目の集合（すなわち $S_{11}, S_{21}, \cdots, S_{m1}$）を**'候補集合'**という．また各共通部分の最後の集合（すなわち $S_{1n(1)}$, $S_{2n(2)}, \cdots, S_{mn(m)}$）を**'決定集合'**という．与えられた集合式を満たすキーを，その集合式の'適合キー'と呼ぶ．

つぎにハッシュ法であるが，詳細な説明は文献[3]に譲ることにする．ところで本文で述べる集合式の処理法は，キーの棄却判定（当該キーがハッシュ表内に存在しない場合，それが正しく判定される機能）の機能を必要とする．このため用いるべきハッシュ法としては，キーの棄却判定を迅速に行う機能を備えたもの，たとえば，直接連鎖法[3]，コンフリクト・フラグを用いる方法[4]，予測子を用いる方法[5]などを前提としている．

ハッシュ表の各エントリは，**Fig. 1** に示すように3つのフィールドから成り立っている．すなわち，与えられた集合式に対する当該キーの適合度を表わす指示子を設定するフィールド（この値を'match level'という），キーを保持するフィールドおよびポインタを保持するフィールド（ただしこれは直接連鎖法の場合である．他のハッシュ法ではこのポインタ・フィールドのかわりに，コンフリクト・フラグ[4]や予測子フィールド[5]が用いられる）の3つである．

われわれの問題は，与えられた集合式に対する適合キーを，ハッシュ表上での処理によって求めることである．

〔例〕 ここで簡単な例として，集合式 $S_1\cdot S_2\cdot S_3$ の処理をつぎに述べる．最初，ハッシュ表の全エントリは空とする．

match level	key	pointer

Fig. 1 Structure of an entry.

* ここでは補集合 $\bar{S}$(i.e.$\{K|K\notin S\}$) を扱っていないが，これは補集合も1つの集合と見ることができるし，また後述のように補集合を含む処理には，実用上，特別の配慮が必要であるためここでは触れなかった．

1) まず S_1 の全てのキーを，ハッシュ法に従って表へ登録する．match level は1に設定する．

2) つぎに S_2 からキーを取り出してハッシュ表内で検索する．その結果，

2.1) 同じキーがみつかった場合：そのエントリの match level が1ならば2に変える．1以外のときは何もしない．

2.2) キーが存在しない場合(検索棄却)：何もしない．

この操作を S_2 のすべてのキーについて繰り返す．

3) つぎに S_3 からキーを取り出してハッシュ表内で検索する．その結果，

3.1) 同じキーがみつかった場合：そのエントリの match level が2ならば3に変える．2以外のときは何もしない．

3.2) キーが存在しない場合(検索棄却)：何もしない．

この操作を S_3 のすべてのキーについて繰り返す．

4) 以上の手続きで，match level が3となったエントリ内のキーが，求める適合キーである．

以上が処理の骨子であるが，この例では match level 用のフィールドは2ビットで充分である．

2.2 加法標準形の処理

上記の例をもとに，ここでは加法標準形の処理について述べる．処理は大きく分けて，各集合への番号づけの作業（フェイズ1）と，ハッシュ表上での処理作業（フェイズ2）の2段階で行われる．

フェイズ 1（各集合への番号づけ——前処理）

与えられた加法標準形の決定集合を除いた残りの全集合に左から通し番号をつける．この通し番号を各集合の**'適合値'**という．最後に全決定集合に共通の番号（値は，上の通し番号の最大値＋1）をつける．この決定集合につけた値を**'適合確定値'**という．

例えば

$$\underset{1}{S_1}\cdot\underset{2}{S_2}\cdot\underset{3}{S_3}\cdot\underset{7}{S_4}+\underset{4}{S_5}\cdot\underset{7}{S_6}+\underset{5}{S_7}\cdot\underset{6}{S_8}\cdot\underset{7}{S_9}$$

となる．適合確定値は7である．

フェイズ 2（ハッシュ表上での実行）

ここではフェイズ1の処理が完了しているものとして，(1)式の処理法を与える．まず下記のアルゴリズムの中で「集合 S を登録する」とは，「match level が適合確定値以外の値を示しているエントリは空とみなし，集合 S 内の全キーをハッシュ法に従って格納すること．その際，同時に match level の値としてこ

の集合の適合値（フェイズ1の結果）を設定すること」を意味する．ただし適合確定値をもったキーとして既にハッシュ表内に存在する場合は，再格納する必要はないことに注意．つぎに「集合 S に関して，match level が I 以上のキーを検索・更新する」とは，「集合 S 内の全キーについて，ハッシュ表内で検索し，match level が I 以上でかつ集合 S の適合値（これを J とする）より小さいようなエントリ（i.e. キー）がみつかれば，新しい match level の値として J を設定する．つまり，集合 S による選別に合格したのである．キーが存在しない場合やあるいは存在してもその match level が I 以上 J 未満でない場合（不合格）はそのままにしておく」という操作を意味する．以後，常にこの意味で用いる．

フェイズ2のアルゴリズム（加法標準形）：

Step 1. $i=1$ とおく．

Step 2. 集合 S_{i1} を登録する．

Step 3. $j=2$ とおく．

Step 4. 集合 S_{ij} に関して match level が S_{ij-1} の適合値以上のキーを検索・更新する．

Step 5. $j=j+1$ とおき，$j>n(i)$ ならば Step 6 へ進む．$j\leq n(i)$ ならば Step 4 へもどる．

Step 6. $i=i+1$ とおき，$i>m$ ならば処理完了．$i\leq m$ ならば Step 2 へもどる．

このアルゴリズムを実行した結果，match level が適合確定値に等しいエントリ内のキーが，集合式(1)の適合キーである．

この方法を要約すると，まず適合キーの候補としてまず1つの集合（i.e. 候補集合）に含まれる全キーをひとまずハッシュ表内に格納し，その後これらのキーを残りの集合で徐々に絞ってゆく（すなわち，match level を更新してゆく），そして決定集合による選別にも合格すればそのキーは適合キーとなっているのである．また処理に必要なハッシュ表の大きさの最小値は，$\mathrm{card}\left(\bigcup_{i=1}^{m} S_{i1}\right)$ 以下である．

つぎにハッシュ表の大きさが一定であるとして，上記のアルゴリズムの処理効率を上げるには次のような条件が考えられる．

条件1）最初に候補として格納するキーの個数はできるだけ小さく，すなわち $\mathrm{card}\,(S_i)$ の小さい集合を登録する．

条件2）残りの集合でつぎつぎと絞ってゆく場合，各集合による検索・更新の処理で合格するキーの少ないもの，すなわち絞る度合の大きい集合から先に行う．

条件3）加法標準形を構成する各共通部分 $\bigcap_{j=1}^{n(i)} S_{ij}$ $(1\leq i\leq n)$ のうち，それを満たすキー（適合キーでもある）の個数が小さいものから先に行う．

上の条件のうち，1)により，Step 4 における表へのアクセス回数を少なくすることができる．また2）によれば，match level の値を更新するために生ずる表へのアクセスの回数を少なくすることができる．また条件3）は，ハッシュ表内での適合キーの蓄積をできるだけ遅らせることを意味しており，結果的に表へのアクセス回数を少なく押えることにつながる．

ところで，集合式の処理にあたって，上記の条件のうちとくに 2),3) は前もって予測することが通常困難である．一方，次章で述べるように条件 1) の効果は他に比べて大きいので，実際の処理においては条件 1) にまず留意し，2),3) については，処理を依頼する人間の側にある程度の判断が任されることになるであろう．条件 1),2) については，次章の実験例によって，より具体的に明らかにする．

3. 実　験

3.1 実 験 例

3つの集合 S_A, S_B, S_C の共通部分を求める実験を行った．採用したハッシュ法は，オーバフロー領域を持った直接連鎖法[3]である．表の大きさは 2000．実験は各集合のキーの個数の変化（すなわち表占有率[3]の変化）および集合式の変化(すなわち実行順序の変化)についての変動をみるため，**Table 1** に掲げるような6つの場合（case 1.1〜case 2.3）について行った．キーとしては，混合合同法により生成した擬似一様整数乱数を用いた．その際，キー・アドレス変換によって生ずるハッシュ表での collision[3] の発生頻度が Poisson 分布になっているかどうかについて，χ^2-検定（適合度の検定，危険率5％）を行い，これで棄却されなかったもののみを用いて，各々 10 回繰り返した．

Table 1 The cases executed by simulations.

cardinal number of each set	set function	case no.
card (S_A)=1000, card (S_B)=500, card (S_C)=200, card $(S_A\cdot S_B)$=200, card $(S_B\cdot S_C)$=100, card $(S_C\cdot S_A)$=40, card $(S_A\cdot S_B\cdot S_C)$=30	$S_A\cdot S_B\cdot S_C$	case 1.1
	$S_C\cdot S_B\cdot S_A$	case 1.2
	$S_C\cdot S_A\cdot S_B$	case 1.3
card (S_A)=1800, card (S_B)=900, card (S_C)=360, card $(S_A\cdot S_B)$=360, card $(S_B\cdot S_C)$=180, card $(S_C\cdot S_A)$=72, card $(S_A\cdot S_B\cdot S_C)$=54	$S_A\cdot S_B\cdot S_C$	case 2.1
	$S_C\cdot S_B\cdot S_A$	case 2.2
	$S_C\cdot S_A\cdot S_B$	case 2.3

また乱数発生や検定の所要時間，ハッシュ関数の実行時間を除いた残りの実質の計算時間を求めるのはやや困難であったので，処理効率を表わす因子として，ハッシュ表へのアクセス回数を採用した．すなわち，キーの検索のためチェインを1つずつたどってゆく操作，match level の値を更新する操作（検索・更新），キーおよび match level の値を一括格納する操作(登録）などはいずれも表へのアクセス操作を引き起こすものと考えた．

実験結果は 10 回の平均値を求め，つぎに述べる解析値とともに，**Table 2** に掲げてある．表には，もとの全キーの個数，card (S_A)+card (S_B)+card (S_C) に対する平均アクセス回数も載せている．

3.2 平均アクセス回数の計算（解析値）

ここでは集合式 $S_1 \cdot S_2 \cdot S_3$ の処理におけるハッシュ表への平均アクセス回数を解析的に求める．

まずハッシュ法としては，オーバフロー領域を持った直接連鎖法を採用するとする．キーはハッシュ表の各エントリに等確率で変換されるものとすると，表占有率が x のとき，collision の発生頻度は Poisson 分布 $P(l, x)=e^{-x}\cdot x^l/l!$ に従うと予想される[5]．すなわち1つのアドレス（エントリ）に l 個のキーが変換される確率は $P(l, x)$ である．表の大きさを M とする．また，

$$\left.\begin{array}{l}\text{card}\,(S_1)=k_1,\ \text{card}\,(S_2)=k_2,\ \text{card}\,(S_3)=k_3,\\ \text{card}\,(S_1\cdot S_2)=k_2',\ \text{card}\,(S_1\cdot S_3)=k_3',\\ \text{card}(S_1\cdot S_2\cdot S_3)=k\end{array}\right\} \quad (2)$$

とする（**Fig. 2** 参照）．k はもとの集合式の適合キーの個数である．以下，**2.1** の〔例〕で与えた処理手続きに従って解析を進める．$\alpha=k_1/M$ とする．

まず集合 S_1 を登録するのに要するアクセス回数を求める．空のエントリに対しては，エントリの取り出しとキーの格納の2回．一般に長さ l のチェインに対しては，チェインを1つたどるごとに1回，またチェインの最後のエントリのポインタ設定に1回，オーバフロー領域の空エントリへのキーおよび match level の格納に1回，計 $l+2$ 回のアクセスが発生する．既述のように，表占有率が x のとき長さ l のチェインの生起確率は $P(l, x)$ であるから，平均すると，

$$2\cdot P(0, x)+\sum_{l=1}^{\infty}(l+2)\cdot P(l, x)=2+x$$

回となる．したがって，集合 S_1 を格納するのに要するアクセス回数は，1つのキーに対して平均，

$$T_1=\frac{1}{\alpha}\int_0^{\alpha}(2+x)dx=2+\frac{\alpha}{2} \quad (3)$$

回となる．

つぎに集合 S_2 内のキーに関して検討する．まず S_1 にも含まれるキー（k_2' 個ある）については，キー自身の検索に平均 $1+\alpha/2$ 回必要であり[3]，さらに match level の更新のために1回必要なので，平均，

$$T_2=1+\alpha/2+1=2+\alpha/2 \quad (4)$$

回となる．

また S_1 に含まれない（S_2 内の）キー（k_2-k_2' 個ある）については，検索時に存在しないことが明らかになるので，結局，棄却のための平均探索回数を求めればよい．まず，空のエントリを探しにいった場合のアクセス回数は1回である．また長さ l のチェインに対して探索を試みた場合は，表に対して l 回アクセスを行うことになる．集合 S_1 を格納した状態での表占有率は α であるから，棄却のための平均アクセス回数は，

$$T_3=P(0, \alpha)+\sum_{l=1}^{\infty} l\cdot P(l, \alpha)=e^{-\alpha}+\alpha \quad (5)$$

となる．

最後に集合 S_3 内のキーについて検討する．まず S_1 にも S_2 にも含まれるキー（k 個ある）については，検索と match level の更新の操作が必要で，キー当たりの平均アクセス回数は T_2 と同じになる．つぎに S_1 に含まれるが S_2 には含まれないキー（$k_3'-k$ 個ある）

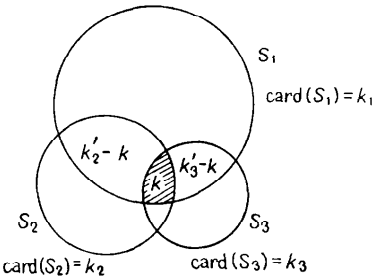

Fig. 2 Venn diagram of $S_1 \cdot S_2 \cdot S_3$

Table 2 Summary of results of simulations and theoretical values.

	observed value		theoretical value	
	total	average	tatal	average
case 1.1	3331	1.96	3289	1.94
case 1.2	2086	1.23	2054	1.21
case 1.3	2026	1.19	1994	1.17
case 2.1	6612	2.16	6532	2.14
case 2.2	3807	1.24	3746	1.22
case 2.3	3699	1.21	3638	1.19

については，検索操作のみであり，match level の更新は不要である．したがってキー当たりの平均アクセス回数は，

$$T_4=1+\alpha/2 \qquad (6)$$

となる[3]．また S_1 に含まれない（S_3 内の）キー（k_3-k_3' 個ある）については，先に求めたように棄却に要する平均アクセス回数 T_3 で与えられる．

(3), (4), (5), (6) および仮定 (2) より，全アクセス回数は，

$$\begin{aligned}T&=T_1\cdot k_1+T_2\cdot(k_2'+k)+T_3\cdot(k_2-k_2'+k_3-k_3')\\&\quad+T_4\cdot(k_3'-k)\\&=k_1\cdot(2+\alpha/2)+(k_2+k_3)\cdot(\alpha+e^{-\alpha})\\&\quad+(k_2'+k_3')\cdot(2-\alpha/2-e^{-\alpha})+k-k_3' \qquad (7)\end{aligned}$$

となる．さらに，1つのキー当たりの平均アクセス回数は，

$$E=T/(k_1+k_2+k_3) \qquad (8)$$

となる．(7), (8) により計算した理論値の例は，Table 2 に実験結果とともに掲げてある．**2.2** で述べた処理効率を上げるための条件 1) および 2) の効果を Table 2 から読みとれるであろう．

4. 一般の集合式への拡張

4.1 拡張の必要性

2. では加法標準形の処理法について述べた．しかし実際には，例えば集合式 $S_1\cdot(S_2+S_3)$ を加法標準形 $S_1\cdot S_2+S_1\cdot S_3$ に一度等価変換してから実行するとなると，その変換自体に時間がかかるばかりでなく，集合 S_1 をハッシュ表に格納する手続きが2重に生ずるので著しく効率が低下することになる．これは，逆に乗法標準形に変換するようにしても効率はよくならない．

そこで本章では一般の集合式を与えられた表現のままで処理（これを‘直接処理’と呼ぶ）できるように拡張を試みる．すなわち補集合を含まない一般の集合式の処理方法について考察する．基本的な考え方は，加法標準形の場合と同様である．まず与えられた集合式を解析し各集合に適合値を割り当てるなどの前処理を行い（フェイズ1），続いて1つずつ集合演算をハッシュ表上で実行してゆく（フェイズ2）．この方法の特徴は，もとの集合式を変換しないで，左から右へ直接処理を行うところにある．

具体的な方法については次章以降で述べるが，その前にここで用語の定義を行っておこう．

まず，集合式の‘決定集合’を拡張し再定義する．

ある集合式がカッコで囲まれた式を含む場合，仮にそのカッコの部分（‘カッコ式’という）を1つの集合と考えると，もとの集合式は加法標準形とみなすことができる．したがって，**2.** で述べたのと同様に決定集合または決定カッコ式が決まるであろう．決定カッコ式については，カッコ内の式に注目すれば再び上と同じようにして，決定集合または決定カッコ式が決まるであろう．このような操作をくりかえして，カッコ式を外側からつぎつぎと展開してゆけば，最後には決定集合のみが出てくるような状態に至るであろう．これらの展開の過程で現われた全ての（**2.** の意味での）決定集合を今後，決定集合と呼ぶことにする．

つぎに‘候補集合’も **2.** で定義した候補集合の拡張概念として上と同様に定義できる（詳細は省略）．

例えば，集合式，

$$(S_1+S_2\cdot S_3)\cdot(S_4+S_5\cdot(S_6+S_7))+S_8 \qquad (9)$$

の決定集合は S_4, S_6, S_7, S_8 の4つ，また候補集合は S_1, S_2, S_8 の3つである．さらにカッコ式 $(S_1+S_2\cdot S_3)$ だけに注目すれば，その決定集合は S_1, S_3 の2つ，候補集合は S_1, S_2 の2つとなる．

4.2 集合式の前処理（フェイズ1）

加法標準形にあらわれる各集合に番号づけ（適合値の割りつけ）を行う処理については **2.** で述べたが，一般の集合式に対する番号づけはつぎのように行う．すなわち，与えられた集合式にあらわれる集合のうち，決定集合以外のものについて，左から通し番号をつけてゆく．ただしその際，もとの集合式に現われる全てのカッコ式については，それぞれの決定集合には同じ番号（適合値）をつけるという条件，さらにそれらの決定集合の位置としては最も右のものを採用するという条件を課すのである．(9) 式の例では，それぞれの適合値は，

$$\left.\begin{array}{cccccccccc}(S_1&+&S_2&\cdot&S_3)&\cdot&(S_4&+&S_5&\cdot(S_6+S_7))+S_8\\2&&1&&2&&4&&3&4\quad 4\quad 4\end{array}\right\} \qquad (9')$$

となる．適合確定値は4である．

〔例〕 ここで集合式 (9') を例にとって，その処理手順を以下に述べる．ただし「登録する」，「match level が I 以上のキーを検索・更新する」などの意味は **2.2** で述べたのと同じである．

1) まず集合 S_1 と S_2 をそれぞれ登録する．
2) 集合 S_3 に関して，match level 1 以上のキーを検索・更新する．
3) 集合 S_4 および S_5 に関して，それぞれ match level 2 以上のキーを検索・更新する．

4) 集合 S_6 および S_7 に関して，それぞれ match level 3 以上のキーを検索・更新する．

5) 集合 S_8 を登録する．

6) 以上の結果，match level=4 のエントリ内のキーが求める適合キーである．

ところで加法標準形の場合は，集合 S_{ij} の検索更新のとき match level を調べるときの値として集合 S_{ij-1} の適合値を用いればよかった（既出，**2.2** の第2フェイズのアルゴリズム Step 4 を参照）．しかし，上の例ではこのようなことは成り立っていない．すなわち，一般の集合式の場合には，集合の検索・更新操作における match level の検査のための値（これをその集合の**'照合値'**と呼ぶ）を指定する必要が生じるのである．

結局，一般の集合式の前処理で行うべきことは，各集合式の適合値を決定すること，および候補集合以外の集合については照合値を決定することの2点である．

適合値の決定の方法は既に述べたので，つぎに照合値の決め方について簡単に述べる．基本的には，集合積演算子（i.e.'・'）がある場合，左の被演算部分の決定集合の適合値が，右の被演算部分の候補集合の照合値となるのである．この方法によって照合値が割り当てられない集合は，もとの集合式の候補集合に外ならない．これらには，仮に照合値0を割り当てる．

(9')の例では，

	$(S_1+S_2\cdot S_3)\cdot(S_4+S_5\cdot(S_6+S_7))+S_8$							
適合値	2	1	2	4	3	4	4	4
照合値	0	0	1	2	2	3	3	0

(9″)

となる．

以上が前処理の概要であるが，具体的なアルゴリズムは付録（次ページ参照）に与えてある．通常のコンパイラで行われている数式の構文解析と同様にスタックを用いて実現される．

4.3 ハッシュ表上での実行（フェイズ2）

集合式の前処理（フェイズ1）に引き続いて，ハッシュ表上の実行（フェイズ2）が行われる．この手続きはつぎのように非常に簡単なアルゴリズムで表わされる．ただし，S_i は集合式の中で左から i 番目に現われる集合，check (S_i) は S_i に割り当てられた照合値，さらに m は集合式を構成する集合の個数である．フェイズ2のアルゴリズム：

Step 1. $i=1$ とおく．

Step 2. check $(S_i)\neq 0$ ならば，Step 3 へ進む．check $(S_i)=0$ ならば，S_i を登録し，Step 4 へ進む．

Step 3. 集合 S_i に関して，match level が check (S_i) 以上のエントリ（i.e. キー）を検索・更新する．

Step 4. $i=i+1$ とおき，$i>m$ ならば処理完了．$i\leq m$ ならば Step 2 へもどる．

このアルゴリズムを実行した結果，matchl evel が適合確定値に等しいエントリ内のキーが，もとの集合式の適合キーである．

このアルゴリズムを(9″)式に適用すると，**4.2** で与えた手続き例と同じになることが確認できるであろう．なお適合確定値は，常にその集合式に含まれる共通部分演算子'・'の個数+1 に等しいという性質が成り立つ(証明略)．したがって match level 用のフィールドは，$\lceil\log_2$(演算子'・'の個数+1)$\rceil$ ビットあればよい*．

5. むすび

ハッシュ表を用いた集合演算の処理技術について述べ，アルゴリズムを与えた．

本稿では補集合を含む演算は扱わなかった．その理由は，例えば $\bar{S}_1\cdot S_2$ のように候補集合として補集合が現われると，$\bar{S}_1$ の登録に支障をきたすからである．すなわち，キーの定義域全体となると非常に膨大となり，また実際の処理要求でも $\bar{S}_1$ だけの問い合わせというのは意味をなさない場合が多いと思われるからである．事実，Codd[6] の ALPHA 言語でも，正しい'alpha expression'の定義の中で，上記のようないきなり否定（補集合）がくるような表現は排除してある．

しかし，集合式 $\bar{S}_1\cdot S_2$ を $S_2\cdot\bar{S}_1$ の形に等価変換すれば，今度は S_2 が候補集合になるので処理が可能になる．すなわち，まず S_2 を登録し，ついで S_1 のキーを検索し，みつかれば match level を0に（すなわち削除）すればよい．しかし本文ではこのような補集合処理のための拡張には全く触れなかった．

また，処理が完了した後に，その結果（i.e. 適合キーの集合）に対してさらに集合演算を追加続行することも，match level の値を見ながらある程度は可能であろう．

さらに，本文で述べた方法では，最後の集合（決定集合）の処理が完了して初めて，まとめて結果が出るであるが，ALPHA 言語[1]の piped mode にみるよう

* $\lceil x\rceil$ は，$\min\{y\mid y\geq x, y$ は整数$\}$ である．

に，処理の過程で1つずつ適合キーが取り出されるような方式も可能であろう．すなわち，詳細は省略するが，集合式を構成する各集合に対して match level フィールドのそれぞれ1ビットを割り当てる．そして集合式を並列に評価（各集合から1つずつ交替にキーを取り出す操作を繰り返す）してゆき，適合キーがみつかるごとに直ちに結果を返すという方法である．ただしこの方法は処理の総所要時間は大となり，また補集合を含む場合はアルゴリズムが複雑になると思われる．

なお，実験には MELCOM 7500 を使用した．実験を進めるにあたって種々ご援助をいただいた筑波大学西村敏男教授，東京教育大学枚田正宏氏に謝意を表する．

参考文献

1) E. F. Codd: A data base sublanguage founded on the relational calculus, IBM San Jose Research Report RJ 893 (July 1971).
2) D. L. Childs: Description of a set-theoretic data structure, FJCC, pp. 557～564 (1968).
3) R. Morris: Scatter storage techniques, Comm. ACM, Vol. 11, No. 1, pp. 38～44 (1968).
4) 古川康一: コンフリクト・フラグをもったハッシュ記憶法，情報処理，Vol. 13, No. 8, pp. 533～539 (1972).
5) 西原，萩原: 予測子を用いた Open Hash 法，情報処理，Vol. 15, No. 7, pp. 510～515 (1974).
6) E. F. Codd: Relational completeness of data base sublanguages, Courant Computer Science Symposia 6, Data Base Systems, pp. 65～98, Prentice-Hall (1972).

付録　フェイズ1のアルゴリズム

フェイズ1の作業領域として用いられるスタックの各エントリは，**Fig. A** のようになっている．address は番地，set id. は集合識別子，match は適合値，check は照合値，また delimiter はカッコの処理用のデリミタを設定する作業用のフィールドである．

下にフェイズ1のアルゴリズムを ALGOL 風の手続きで示す．ただし，集合式はその前後を記号▽によって囲まれているとする．また，p は解析用のポインタで，集合式の記号の左からの位置を示す．ただし集合式の左端の▽の位置を0とし，集合識別子はひとまとまりにして考える．a はスタックのアドレス変数，v は適合値のための通し番号を表わす変数である．p, a, v の初期値はそれぞれ 0, 1, 1 である．

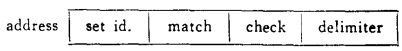

Fig. A Structure of an entry of the stack.

また，match(a), check(a), delimiter(a), setid(a) は，番地 a のそれぞれのフィールド（の内容）を表わす．スタックの全フィールドは最初すべて0になっているものとする．

下記のアルゴリズムにおいて，present の欄は pointer が現在指している集合式の記号を，また next の欄はその1つ次（右）の記号を表わす．operation の

Table A An algorithm of Phase 1.

next	present	operation
(	free	delimiter (a):=delimiter (a)+1; p:=p+1;
SET	free	setid (a):=SET; a:=a+1; p:=p+1;
•	SET	match (a−1):=v; check (a):=v; v:=v+1; p:=p+1;
	)	w:=a; L1: w:=w−1; **if** match (w)=0 **then** match (w):=v; **if** delimiter (w)=0 **then go to** L1; delimiter (w):=delimiter (w)−1; check (a):=match (a−1); v:=v+1; p:=p+1

+	SET	w:=a;
		L2: w:=w−1;
		LL: **if** w=1 **then** L3: **begin**
		check (a):=check (w);
		p:=p+1
		end
		else
		if delimiter (w)=0
		then go to L2 **else go to** L3;

	)	w:=a; L4: w:=w−1; **if** delimiter (w)=0 **then go to** L4; delimiter (w):=delimiter (w)−1; **go to** LL;
)	SET	p:=p+1;

	)	w:=a;
		L5: w:=w−1;
		if delimiter (w)=0 **then go to** L5;
		delimiter (w):=delimiter (w)−1;
		p:=p+1;

▽	free	w:=a;
		L6: w:=w−1;
		if match (w)≠0 **then go to** L7;
		match (w):=v;
		L7: **if** w=1 **then go to** END
		else go to L6;

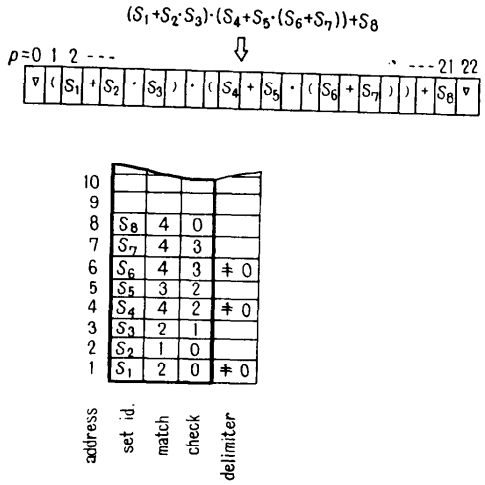

Fig. B An example of preprocessing (Phase 1).

欄は，左の条件（present および next の記号が一致すること）が成立するときに行うべき処理を示す．free とあるのは任意の記号，SET は集合の識別子を表わす．下記の各 operation は，右端の▽が出てくるまで繰り返される．アルゴリズムは，**go to** END において停止する（**Table A**（前ページ）参照）．

〔例〕 本文中でとりあげた集合式の処理結果は **Fig. B** のようになる．太枠の部分が，適合値，照合値の割り当ての結果であり，フェイズ2ではこれを第1番地から実行してゆくのである．

（昭和51年3月18日受付）
（昭和51年6月18日再受付）