5N-8

システムXEROにおける高水準データ定義／操作言語

脇田建　深澤寿彦　加藤和彦　益田隆司

東京大学　理学部

1.　はじめに

XERO は、分散環境における data intensive application [加藤89]の構築を支援するシステムである。XERO 開発の主な目標は永続的データの管理、データ操作の高度な記述、分散環境での効率的なデータ操作の実現などである。本稿では、XEROに高水準のデータ定義、データ操作の記述力を与えるデータ記述言語について述べる。この言語は関数型言語をもととし、集合の内包的記述法を取込むことにより、大量のデータの流れを自然に記述できる。関数型言語は、その参照透明性ゆえに、プログラムの形式的操作が容易であることが期待される。この性質を利用して、プログラム変換の技術によりソース言語レベルでより効率の良いプログラムに変換できることを示す。さらに、このような変換を施した後に、手続き型言語に変換することにより効率よく実行することができることを示す。

2.　記述言語の表現

本稿で述べる記述言語は、ＺＦ記法と呼ばれる集合の内包的な表現を積極的に取り入れることによって大量のデータの流れの自然な記述が可能となる。このため、関係データベース演算のようにデータの流れがはっきりしたものが簡明に表現できる。

2．1.　ＺＦ記法

D.A.Turner の導入したＺＦ記法は、集合の内包的記法に generate-filter-map の意味を与えリスト処理を記述するために用いられた記法である。Turner は、このＺＦ記法を導入した言語を設計し、リスト処理の多くの問題を簡明に記述できることを示した[Turn85]。本記述言語は、ＺＦ記法をデータ操作記述に適用したものである。以下に関係代数演算の中の選択演算と直積演算を定義した例を挙げる。

```
selection cond R = { r | r ← R; cond r }                (1)
```

式の右辺は、リスト R の全要素 r につき、条件 cond を満足するもののリストを表わす。

```
cartesian_product R S = { r^s | r←R; s←S }              (2)
```

^ は二つのタプルを結合する演算子とする。上式は、リスト Rと S のすべての要素どうしの対を作る操作の定義を表わす。

2．2.　データ構造記述

記述言語は、基本データ型として整数型（integer）、論理型(bool)、実数型(real)、文字列型(string)を提供し、リストとタプルとを導入することでより複雑な構造を記述する。リストは、同じ型のオブジェクトの列とする。タプルは、任意の型のオブジェクトの有限順序列とする。但し、オブジェクトは基本データ型のデータ、オブジェクトのリスト型のデータ、オブジェクトのタプル型のデータとする。例えば、文字列と整数のタプルのリストといったようなものもオブジェクトとみなされる。以下のように定められるオブジェクト labs は、ある研究所の研究室の構成員を保持する。'{','}' はリストを、'<','>' はタプルを表わす。この例は、研究所が複数の研究室より成り、lab_name で特定される各研究室には一人の教授と複数の学生がおり、各学生に対し写真と年齢が記録されている様子を表わす。

```
labs = {< lab_name  : string(30),
          prof_name : string(30),
          students  : {< name : string (30),
                         fig  : picture >} >}            (3)
```

次に、このデータベースの上での問い合わせの例を挙げる。

```
stds ln = { students | <lab_name,
                        prof_name,
                        students> ← labs;
                        lab_name = ln }                 (4)

  { name | <name, fig> ← (stds "Lab1")}                 (5)
```

(4) は、引数 ln で特定される研究室に属する学生の情報を返す関数の定義を、(5) は、関数 stds を用いて、研究室　"Lab1" に属する学生の名前をとりだす問い合せを表わす。

3.　問い合せの実行

前章で述べた、記述言語の実行形態について述べる。記述言語は、変換規則に基づいて手続き的な表現に変換して実行される。手続き的な表現にはストリームを扱うプリミティブを用意し、単一、複数プロセッサのいずれの実行環境においても効率的にストリームが使用されるような機構を提供する。本章においては、記述言語の手続き的な表現への変換を紹介する。

3．1.　手続き型プログラムへの変換

関数型言語の参照透明性により記述言語を他の言語に変換する作業は比較的容易である。ＺＦ記法がいくつかの翻訳規則によって手続き型に変換できることを[加藤88]で示した。以下に、関係演算の結合演算での変換の例を挙げる。

```
join cond R S = { r^s | r←R; s←S; cond r s }
          ==> foreach r in R
                foreach s in S
                  { r^s | cond r s}
          ==> foreach r in R
                foreach s in S
                  if ( cond r s )
                    put( r ^ s ) ;
              put( EOS ) ;                              (6)
```

High Level Description Language for Data Definition and Data Management on System XERO
Ken WAKITA, Toshihiko FUKASAWA, Kazuhiko KATO, and Takashi MASUDA
Univ. of Tokyo

このようにして、ＺＦ記法が手続き的な表現に落とされた。ここで、foreach と put はタプルの列（ストリーム）を操作するためのプリミティブである。前者はストリームを先頭から最後まで順に参照し、後者は出力ストリームに要素を一つ出力するものである。実行時には、一つのＺＦ式が一つのスレッドとして実現され、高機能チャネル[猪原89]を通してストリームのデータが受け渡される。

３．２．　大量データの効率的な扱い

問い合せ（5）が、実行される場合には前章に述べたように、二つのスレッドができ、それらがストリームを通してレコードの受け渡しをする。一方のスレッドは、データベース labs のレコードで lab_name が該当するものについて、students 属性の値をストリームの要素として出力し、他方はストリームから取り出した全レコードの name フィールドの値をストリームに出力する。この問い合わせにおいては、name フィールドのみが必要であるにも関わらず、ストリーム中を参照されることのない項目 fig（画像データ）が流れ、大きなオーバーヘッドとなってしまう。本言語では、全データに記憶されている場所を表わすオブジェクトＩＤを割当て、このような参照の行なわれないデータについては、データの実体でなくＩＤを渡すことにより、ストリームを流れるデータ量を減らす。

４．　ストリーム処理の拡張

ＺＦ式では、ストリームの走査を途中で止めることができない。しかし、ストリームをすべて走査する必要がないことも多い。以下、関係代数演算の共通部分演算を例として挙げる。

```
intersection R S = { r | r ← R; s ← S; equal r s }
```

ここで R と S はストリーム、'equal r s' は r と s が等しいか否かを判定する関数である。この intersection を手続き型プログラムに変換すると、次の様になる。

```
foreach r in R
  foreach s in S
    if ( equal r s )
      put( r ) ;
put( EOS ) ;
```

r と s が等しいことが判った時点で、それ以上 S を走査する必要はない。そこで、ストリームの要素の流れを制御するために、'yield 〈式〉'、'next 〈変数〉'、'return 〈式〉'、'->' をスペシャルフォームとして導入する。'yield 〈式〉' は、〈式〉の値をストリームの要素として出力することを表わす。'next 〈変数〉' は〈変数〉で示されるストリームの現在の要素を捨て、次の要素の処理をはじめることを表わす。また、'B-> 〈スペシャルフォーム〉' では、B は論理式であり、B が真のとき〈スペシャルフォーム〉が実行される。'return 〈式〉' は関数の返す値を〈式〉の値とすることを表わす。例としてスペシャルフォームで定義した共通部分演算と、変数 r の値がストリーム S に含まれるか判定する関数 member を挙げる。

```
intersection R S = {| r←R:yield EOS ;
                      s←S ;
                      (r = s) -> [yield r ; next r]|
member r S = {| s←S: return False ;
                (r = s) -> return True }
```

R と S はストリームである。ここで 'r<-R:〈スペシャルフォーム〉;' は R の要素をすべて走査し終えたら、〈スペシャルフォーム〉を実行することを表わす。

５．　プログラム変換によるスレッドの融合

データベースへの問い合せをＺＦ記法で記述された処理を組み合せて行なうと、３．１．で述べたように各ＺＦ式がそれぞれ一つのスレッドとして実行されるので、一つの問い合せが複数のスレッドによって処理されることになる。互いにデータの受け渡しを行なうスレッド群が同一プロセッサに配置される場合は、それらのスレッド群を一つのスレッドに融合することにより、スレッド間での中間結果の受け渡しやスレッドの文脈切り換えによるオーバヘッドをなくすことができる。ＺＦ式においては、入れ子になった複数のＺＦ式を一つのＺＦ式に変換することによって、スレッドの融合を実現することができる[加藤88]。以下に、関係演算の結合演算と選択演算の組み合せを変換の例として挙げる。下線部が変換の対象となる部分である。

```
join cond1 (selection cond2 R) (selection cond3 S)
  ==> { r^s | r←{ r | r←R; cond2 r };
              s←{ s | s←S; cond3 s };
              cond1 r s }                          (置換)
  ==> { r^s | r'←R; cond2 r'; r==r';
              s'←S; cond3 s'; s==s';
              cond1 r s }                          (展開)
  ==> { r^s | r'←R; r==r'; cond2 r';
              s'←S; s==s'; cond3 s';
              cond1 r s }                          (==交換)
  ==> { r'^s' | r'←R; cond2 r';
                s'←S; cond3 s';
                cond1 r' s' }                      (==消去)
```

'v == 〈式〉' は〈式〉の値を変数 v に束縛することを表わす。

６．　おわりに

現在、記述言語の言語仕様を設計し、手続き型言語への変換規則をまとめている。またプログラム変換について、スペシャルフォームを含めた変換規則を検討中である。

参考文献

[猪原89] 猪原茂和、他: システム XERO におけるプロセス間大量データ通信機能、情報処理学会38回全国大会講演論文集、1989.

[加藤88] 加藤和彦、他: ＺＦ記述に基づいたデータベース処理の記述と変換、日本ソフトウェア科学会第５回大会論文集、1988.

[加藤89] 加藤和彦、他: Data Intensive Application 構築支援システム XERO の構想、情報処理学会38回全国大会講演論文集、1989.

[Turn85] Turner, D.A.,: 'Miranda − a non-strict functional language with polymorphic types'. In Proc. Conference on Functional Programming Languages and Computer Architecture, Nancy, 1-16., LCNS 201, Springer Verlag, 1985.