ソフトウェア工学 73－16
（１９９０．７．２０）

# 大規模ソフトウェアのための関数型プログラミング言語

新田稔
（株）ＳＲＡ
ソフトウェア工学研究所

鳥居宏次
大阪大学
基礎工学部

最近、ソフトウェア開発の現場にフォーマルなアプローチを導入する試みがいくつかなされているが、我々はその一環として、大規模なソフトウェアに適用できる関数型プログラミング言語を開発した。従来関数型言語は研究室内ではよく使われまた成果も上げているが、実際のソフトウェア開発に用いられた例はほとんどない。この原因は、それらが現実のソフトウェア開発に不適当な面を持っていたからであると思われる。我々は、関数型言語の特徴からくる静的な検査機能を強化するとともに、関数型プログラミング言語大規模なソフトウェアに適用する上で障害となっていたシステム・リソースのアクセスや記述量について構文上の工夫を行なった。

# A Functional Programming Language for Software in-the-Large

Minoru Nitta
Software Research Associates,Inc
Software Engineering Laboratory

Koji Torii
Osaka University
Faculty of Engineering Science

Recently, some attempts to introduce the formal aprroach into software dvelopments are being carried out. We also have developed a functional programming language applicable to software in-the-large. Functional languages are useful and effective in laboratories, but there have been few cases of that they have been used for the industorial software developments. The reason is that they have some unsuitable points for the real software developments. We have devised some notations to serve the system resource access and to reduce the description size, as well as we have enhanced the static checking facility which comes from the nature of functional languages.

## 1. まえがき

我々は、大規模なソフトウェアに適用できる関数型プログラミング言語「akanc」を開発した。

近年ますます大規模化・複雑化してくるソフトウェアに高い品質を確保するため、ソフトウェア開発に代数的な記述言語を用いたフォーマル・アプローチの導入が期待されている。要求分析、仕様記述、設計、プログラミング、保守などのソフトウェア開発の各フェイズ、またはそれらの一貫を対象とした多くの研究が進められている(たとえば[1][2][3][4])が、我々は現状のソフトウェア開発の形態を考慮し、プログラミング・フェイズへのフォーマルな記述言語の導入がもっとも簡単で効果が現れやすいと判断して、大規模なソフトウェアに適用できる関数型プログラミング言語の開発を始めた。

従来、多くの関数型言語が存在し研究室での実験などでは成果を上げているが、それらが実際のソフトウェア開発に用いられた例はほとんどない。その原因は、それらの言語が記述対象となるソフトウェアや作業形態、開発作業を行なう技術者に対して、不向きな点・不都合な点を持っているからであると思われる。我々はそれらの点をなくすという方向で、実際の問題に近い例題を用いながらプロトタイピング法により言語の研究・開発を進めた。研究室内での実験と実際のソフトウェア開発で扱われるソフトウェアには、たとえば次のような相違点がある。

| 研究室 | 実際 |
|---|---|
| 小規模で、小人数・短時間で開発できる。また記述量も少ない | 大規模で、開発には大人数・長時間を要する。また記述量も多い |
| 要求仕様がすべて明確であるか、または不明確な箇所は開発者側の判断で決めてよい | 要求仕様に不明確なところがあり、またユーザ側の要請によって変更される |
| 処理のほとんどがメモリ内だけで行なえる | データベースやネットワーク、高度なユーザ・インターフェイスなど、システム・リソースを頻繁に利用する |
| 保守作業のことはあまり考えなくてよい | 常に保守作業を考慮しなければならない |
| プログラムの内容は開発者側が理解できればよい | プログラム内容をユーザ側に説明し、理解させなければならないこともある |
| 以前に誰かが解いたことがある問題 | いつも初めて出会う問題 |

さて、関数型言語をプログラミング作業に導入したときの利点は、記述が数学的に扱えてプログラムの静的な検査や解析が厳密に行なえること、順序に必然性がある処理と独立・並列に行なえる処理とが自然に書き分けられること、などである。これはソフトウェアの品質や保守性に大きく貢献する。しかし一方、(a)関数の呼び出しに副作用がなく同じ引数の値に対していつも同じ値を返す、(b)値の保存場所としてのグローバルな変数やスタティックな変数がなく必要な値はすべて引数として関数に渡す、などの関数型言語の特徴は、システム・リソースのアクセスの記述には不利であり、また記述量の増大にもつながる。

我々は、プログラムに誤りが入り込み難い、また誤りがあってもシンタックスのエラーとして表面化するよう言語の静的な検査機能を強化し、上の(a)(b)の関数型言語の不都合な面を克服して、スムーズにシステム・リソースのアクセスが記述できるよう、また記述量の削減もできるよう、いくつかの工夫をakancに取り入れた。これらの工夫は、構文に関するものと処理系に関するものに分けられるが、今回は構文に関する工夫について報告する。なお、関数型言語の特徴、大きなソフトウェアのプログラミングに適用する際の問題点、我々の目標、akancに取り入れた工夫、の関係を図1に示す。

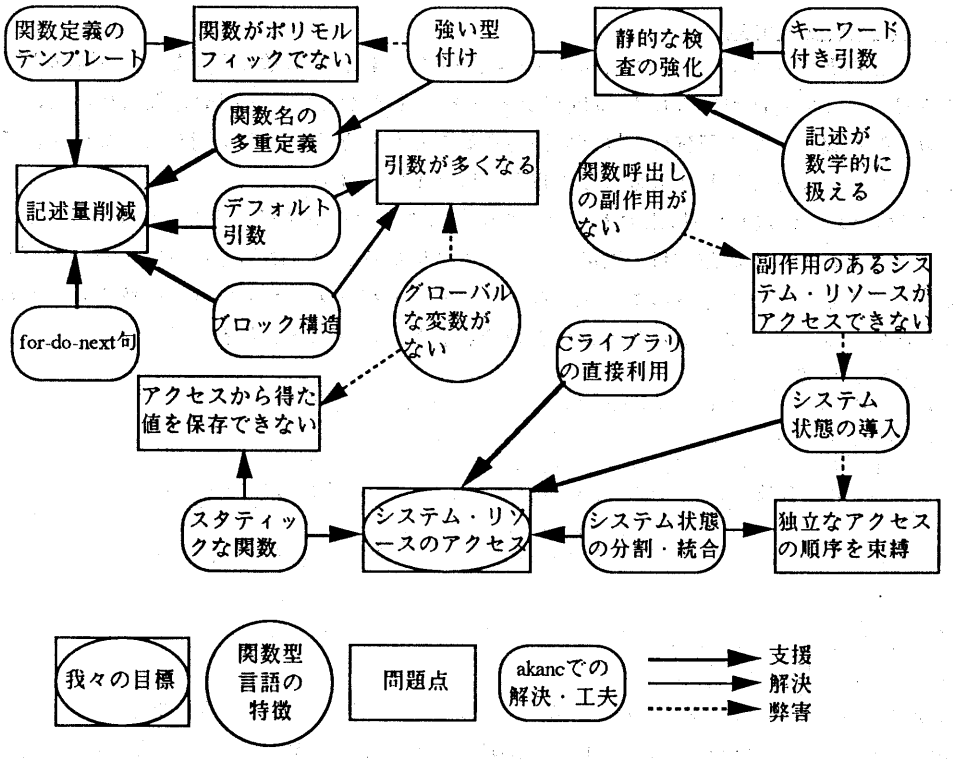


図1 関数型言語の特徴、問題点、akancでの工夫の関係

## 2.静的な検査機能の強化

### 2.1.強い型付け

akancは強い型付けを持つ言語であり、演算子の各項の型、関数の引数の型、演算結果の型、などの整合はすべて静的に検査される。たとえば加算演算子「+」や等値演算子「=」では、その項と演算結果の型は次のように規定されている。

「+」
- ●第1項の型は整数型か実数型
- ●第2項の型は第1項の型と同じ
- ●結果の型は第1項の型と同じ

「=」
- ●第1項の型はすべての型
- ●第2項の型は第1項の型と同じ
- ●結果の型は論理型

ここで整数型、実数型、論理型などは、言語であらかじめ定義されているプリミティブな型である。これらの演算子は、ポリモルフィックなものではなく、条件を満たすすべての項の型について多重定義されている。

プログラマは「typedef　基底になる型の型名　新しく生成する型の型名 ;」の形で、新しい型を生成することができる。たとえば

```
typedef  int     円  ;
```

新しい型が生成されると、基底になると指定されていた型を項の型に持つ演算子も新しい型について生成される。また新しい型の定数は「基底になる型の定数　新しい型の型名」の形で表現される。したがって「10円 + 20円 = 30円 + 40円」は正しい式であるが、「100 + 200円」や「300円 = 400」などでは、型の不整合が起きる(第1項と第2項の型が違う)。なおakancでは

```
ドル == int ;
```

などとして、単に型の別称を定義することもできる。

## 2.2.キーワード付き引数

さて、ほとんどのプログラミング言語(akancも含めて)では、実引数と仮引数は、引数が置かれている位置によって対応づけられる(位置パラメータ)。したがって、実引数と仮引数の対応の誤りは引数の型が異なるなら型整合の検査で検出することができるが、型が同じ実引数の位置を入れ換えてしまったような誤りは型整合の検査からは検出できない。

akancではこのような誤りを検出するため、実引数にキーワードを付けることができる。ただし、キーワードを付けることが記述量の削減にはマイナスになるという議論もあってキーワードを省略してもよいことになっているので、完全なキーワード・パラメータではなく位置も正しくなくてはならない。なおキーワードとしては仮引数の名前が用いられる。たとえば

```
/* 関数の定義 */
putAt(int*[10] array,int index,value) == array@[index]:=value ;

/* 関数の参照 */
foo(int*[10] array) == putAt(array,100,0) ;
     /* 第２引数と第３引数が逆になっていることが静的に検出できない */

bar(int*[10] array) == putAt(array,value=>100,index=>0) ;
     /* 下線部がキーワード、引数が逆になっていることが検出できる */
```

# 3.システム・リソースのアクセス

## 3.1.システムの状態を表す値

システム・リソースのアクセスは通常何らかの副作用を伴い、手続き型言語では関数を参照する順序によってその結果が異なる。しかし関数型言語では、関数の参照には副作用がなく関数は同じ引数の値に対していつも同じ値を返すので、手続き型言語のように、たとえばREADという関数を引数なしかまたは同じ引数の値で繰り返し参照してファイルから順次値を読む、というようなことはできない。

関数型言語ではこの不都合を解決するために、システムの状態を示す値を導入するのが一般的である[5]。つまりシステム・リソースをアクセスする関数にアクセス前のシステム状態を渡し、アクセス後の新しいシステム状態をアクセス結果の値と共に受け取るのである。次にリソースをアクセスする関数にはその新しいシステム状態が渡される。

この方法ではシステムの状態を示す値の入出力をたどればシステム・リソースをアクセスする関数の評価順序になるが、これでは本来独立・並列に処理できるはずのリソースのアクセスにもひとつの評価順序を定めてしまうことになる。これを避けてそのようなリソースのアクセスを自然に記述するため、akancではシステム状態の分割・融合方式を採用した。

システム状態の分割は組込関数「Fork」で、融合は「Join」で行なわれる。たとえば

```
struct {state s,int i} compare(state s)
with  FORK # struct {state s1,s2} Fork(s),
      OPEN1 # struct {state s,int fd} open(FORK.s1,"file1"),
      READ1 # struct {state s,int i} read(OPEN1.s,OPEN1.fd),
      CLOSE1 # state close(READ1.s,OPEN1.fd),
      OPEN2 # struct {state s,int fd} open(FORK.s2,"file2"),
      READ2 # struct {state s,int i} read(OPEN2.s,OPEN2.fd),
      CLOSE2 # state close(READ2.s,OPEN2.fd),
      JOIN # state Join(CLOSE1.s,CLOSE2.s),
      MAX # max(READ1.i,READ2.i)
==    {JOIN,MAX) ;
      /* OPEN1,READ1,CLOSE1 と OPEN2,READ2,CLOSE2 は、独立・並列に処理しえる */
      /* open,read,close,max は他で定義されているものとする */
```

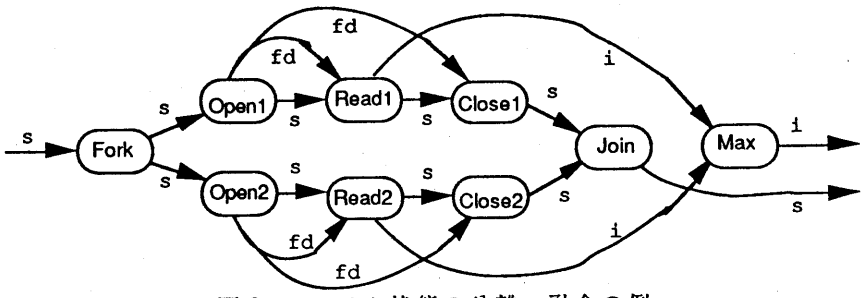


図２　システム状態の分離・融合の例

3.2.デフォルト引数とスタティックな関数

akancのプログラムの実行は、「main」という名前を持つ関数(メイン関数と呼ぶ、「main」は多重定義できない関数名)の評価から始まる。プログラムが起動されるとき、メイン関数には現在のシステム状態を示す値、コマンド・ラインのアーギュメントの数、およびアーギュメントの値が引数として渡される。これらの引数はその値が処理に必要なければ省略できるが、もしすべての引数が省略され実行時に与えられる値がないのなら、プログラムの結果は実行を待つまでもなくプログラムの解析時に導かれる。

さてメイン関数に渡された値は、メイン関数で参照される関数へ、またその関数で参照される関数へと次々に渡されていくなら、すべての関数で参照することができる。そこでそれらの値は引数として陽に記述しなくとも、メイン関数の仮引数名を使ってどの関数でも参照できるものとした。これがデフォルト引数である。デフォルト引数があると、表面的な引数のない関数を定義することができる。それらをスタティックな関数と呼ぶ。スタティックな関数は一度評価されるとその結果を保存し、二度目以降の参照では保存しておいた結果を返すよう処理系で扱われる。

スタティックな関数は、システム・リソースのアクセスに重要な役割を果たす。リソースをアクセスする関数にはシステムの状態を渡してアクセス後の状態を得るが、過去の状態に遡ってリソースをアクセスすることはできないので、関数に渡すシステム状態は常にその時点のシステム状態を示す値でなくてはなない。ところが関数型言語には値の保存場所としてのグローバルな変数がないので、ある時点のアクセス結果を後で何度か参照するには、参照する場所まで引数として持ち回らなければならない。とくに大きなプログラムではその初期化処理の段階でシステムから多くの値を得て後々それを利用するというケースがあり、それらの値を引数として持ち回れば記述の繁雑化は避けられない。スタティックな関数はそれを防いでいる。たとえば

```
state main(state s0)
with  INIT # state initial(),
      PROC1 # struct (state s,int i) process1(INIT),
      PROC2 # state process2(PROC1.s,PROC1.i),
      FINAL # state final(PROC2)
==    FINAL ;

state initial() == struct (state s,int fd) openFile().s ;
struct (state s,int fd) openFile() == struct (state s,int fd) open(s0,"file") ;
FD == openFile().fd ;  /* 以降 FD は、openFile().fd と置き換わる */

struct (state s,int i) process1(state s) == read(s,FD) ;
state s process2(state s,int i) == write(s,FD,i * 2) ;
/* openFile().fd を何回参照しても open(s0,"file") は一度しか評価されない */
```

### 3.3.C言語ライブラリとのリンク

akancではシステム・リソースのアクセスは、形の上で「external」というブロックに属する関数(外部関数と呼ぶ、ブロックについては後述)で行なわれる。どのような外部関数が存在するかは言語では定義されていないが(環境で定義される)、外部関数は演算子や組込関数と同じくプリミティブな関数である。それらの処理はC言語で書かれたライブラリ・ルーチンなどで実現され、akancとのインターフェイスは次のようにとられる。

- ●関数名　　ライブラリ・ルーチンのエントリ名
- ●第１引数　　現在のシステム状態、正当な値かどうか処理系でチェックされる
- ●第２引数以降　そのままライブラリ・ルーチンへ渡される
- ●戻り値　　ライブラリ・ルーチンが値を返さないなら処理系でアクセス後の状態を示す値が生成され返される、ライブラリ・ルーチンが値を返すならアクセス後の状態とその値が構造体として返される
- ●なお、ある大きさのメモリを確保してそのアドレスを渡すと、そこへ値を格納して返すようなライブラリ・ルーチンでは、配列または構造体で確保したメモリに相当するアドレスを渡し、戻り値の構造体の第３要素以降にその引数と同じ型の要素を置いて返された値を受け取る

我々は当初、システム・リソースにアクセスする関数を組込関数として用意すること(どのような関数が存在するかを言語で定義すること)を試みたが、UNIXに存在するユーティリティやシステム・コールに相当する組込関数をすべて処理系に組込むことは難しく、また環境の変化にも柔軟に対応できないので、C言語で書かれたライブラリを直接利用することにした。

```
state hello(state s) == state printf(s,"hello¥n"):external ;
struct (state s,int i) read(state s,int fd)
with READ # struct (state s,int cc,int*[1] i) read(s,fd,(int*[1]),4):external
== (READ.s,READ.i[0]) ;
```

## 4.記述量の削減

### 4.1.関数名の多重定義

akancでは、関数は、関数名と引数の数・型の組合せで識別されるので、引数の数か型が異なれば同じ名前を複数の関数に付けることができる。この関数名の多重定義により、引数の型は違うが同様の処理を行なう関数や、オプショナルな引数を持つ関数に同じ名前を付けることができる。関数型のプログラムでは、大きな関数を少数定義するよりも小さな関数を多く定義した方が、プログラムの可読性、保守、関数の再利用などに有利であるが、関数名の多重定義により、それらの関数に付ける名前に頭を悩ませたり、妙に説明的な長い名前を付けて記述量を増やすということが避けられる。

```
/* 引数の型は違うが同様の処理を行なう関数 */
max(int i,j) == if i > j then i else j ;
max(real a,b) == if a > b then a else b ;

/* オプショナルな引数を持つ関数 */
print(state s,string m) == printf(s,"%s",m) ;
print(state s,string m,int length) == printf(s,"%s",m[0//length])
```

### 4.2.関数定義のテンプレートからの自動生成

前述したようにakancは強い型付けを持つ言語であり、関数の引数の型の整合はプログラムの実行前にすべて調べられる。また関数はポリモルフィックではないので、引数の型が異なる同様の

処理については、それぞれの型について関数を別に定義しなければならない。これは誤りの発見の上からは有利であるが、記述量の面からは不利である。

これを解決するためakancには、型を特定しない仮引数(多様型の引数と呼ぶ)を持つ関数定義のテンプレートから、仮引数の型をその関数が参照されるときの実引数の型とマッチングさせた関数の定義を生成する機能がある。この関数定義のテンプレートからの生成は

●その関数の定義がなく、
●テンプレートと関数名、引数の数、多様型でない引数の型が一致し、
●多様型の引数がマッチし、
●実引数の型を多様型の仮引数に当てはめてもテンプレートに型の不整合がなく、
●生成される関数の型が参照される関数の型と一致する

ときに行なわれる。また、多様型の引数の型の宣言と、それにマッチする実引数の型は次の通り。

| 仮引数の型宣言 | マッチする実引数の型 |
|---|---|
| ●特殊な型名「any」 | どんな型でも |
| ●大きさを指定しない配列 | 要素の型が一致すればどんな大きさの配列でも |
| ●要素が多様型の配列 | 要素がマッチする大きさが同じ配列 |
| ●多様系の要素を含む構造体 | 多様型でない要素の型が一致し、多様型の要素がマッチする |

引数や関数の型が特定はしないが同じ型であるときには、あらかじめ「typedef」を用いて新しい多様型を生成しておく。たとえば

```
/* 関数のテンプレートによるもとの記述 */
typedef any myType ;
myType max(myType x,y) == if x > y then x else y ;

/* 関数の参照 */
foo(real a,b,int i,j) == real max(a,b) * int max(i,j) ;

/* 生成される関数の定義 */
int max(int x,y) == if x > y then x else y ;
real max(real x,y) == if x > y then x else y ;
```

## 4.3.繰返し処理のための再帰的な関数定義の自動生成

一般に関数型言語には繰返しを行なう制御構造がなく、繰返し処理は関数の再帰的な定義で記述される。akancには、このような繰返しのためだけの関数の定義を「for-do-next」句から自動生成する機能がある。たとえば

```
/* FOR-DO-NEXT句を用いたもとの記述 */
real sum(real*[10] a) == real for i=>0 do if i < 10 then next(i + 1) + a[i] else 0.0 ;

/* 生成される関数の定義と参照 */
real sum(real*[10] a) == sum_1(0,a) ;
real sum_1(int i,real*[10] a) == if i < 10 then real_1(i + 1,a) + a[i] else 0.0 ;
```

## 4.4.ブロック構造

akancのプログラムはブロック宣言文を用いていくつかのブロックに分割することができる。ブロックの範囲は、ブロック宣言文から次のブロック宣言文の前まで、あるいはプログラムの終りまでである。あるブロックの範囲で定義された関数はそのブロックに所属する。異なるブロックに所属する関数は、関数名と引数の数・型の組合せが同一でも衝突しないが、あるブロックから他のブロックに所属する関数を参照するには、その関数の所属するブロック名を指定しなければならない。

プログラムの先頭から最初のブロック宣言文が現れるまでは、どのブロックにも所属しない範

囲であり、ここで定義された関数はどのブロックにも所属しない。これらの関数はどこからもブロック名を指定しないで参照されるが、それらと関数名、引数の数・型の組合せが同じ関数を他のブロックで定義することはできない。

またブロックはプライベートであると宣言することもできる。プライベートなブロックは独自のメイン関数を持ち、その引数はそのブロックに所属する関数に対してデフォルト引数となる。プライベートなブロック内のメイン関数以外の関数は、相互には参照できるが、ブロック外からは直接参照されることはなくメイン関数を通じて参照される。たとえば

```
struct (real sum,ave) main() == struct (real sum,ave) main(3.2,4.8):num ;

private block num ;
      struct (real sum,ave)
      main(real a,b) == (real sum(),real ave()) ;
      sum() == a + b ;
      ave() == (a + b) / 2 ;
```

このプログラムのブロック構造により、複数人でのプログラミング作業がスムーズに行なえる。なお、組込関数はどのブロックにも所属しない関数であり、外部関数は形の上で「external」というブロックに所属する関数である。

## 5.あとがき

我々は、ソフトウェア開発へのフォーマル・アプローチの導入の一環をなすべく、関数型プログラミング言語にいくつかの工夫を取り入れて、大規模なソフトウェアの記述の適用させようとしている。コンパイラの試作から、実行速度やメモリ効率においては関数型言語は充分実用になることが確認されているので、これから実作業への普及には技術者の教育やユーザの理解などが問題となるだろう。

我々はいま、C言語などによってすでに開発され納品されたプログラムをakancで書き直す実験を行なっているが、もとのプログラムに隠れていた誤りが記述段階で発見されたこともあった。このような実績を重ねて、関数型言語の有用性を広くアピールしたいと考えている。

なおここで報告した内容は、大阪大学基礎工学部と株式会社ＳＲＡの共同研究プログラム内で、本稿の第１著者が同大学同学部内で作業中に行なわれたものである。

## 文献


[1]梶谷,伊藤,松浦,谷口,嵩 オフィス・ワークの代数的記述と検証
電子情報通信学会技術研究報告 AL84-38

[2]A.T.Nakagawa, K.Futatsugi Stepwise Refinement Process with Modularity:An Algebraic Approah
The 11th International Conference on Software Enginerring

[3]佐藤 FDSSを使った関数的使用記述 情報処理学会第39回全国大会

[4]K.Inoue, T.Ogihara, T.Kikuno, K.Torii A Formal Adaptation Method for Process Description
The 11th International Conference on Software Enginerring

[5]稲田,杉山,鳥居 代数的仕様記述方に基づく抽象的順序機械へのシステムコールなどの導入
情報処理学会第35回全国大会

[6]新田 akancによる変数を使わないプログラミング
ソフトウェア・ツール・シンポジウム'90

[7]新田 関数型プログラミング言語の「Software in-the-large」への適用
第10回ソフトウェア・シンポジウム