数理解析研究所講究録
第 482 巻 1983 年 215-239

Design and Implementation of A Highly Modularized Functional language

Nobuyuki Saji

Software Product Engineering Laboratory
Nippon Electric Company

Akinori Yonezawa

Department of Information Science
Tokyo Institute of Technology

I. Introduction

The recent rapidly growing interest in functional programming is a practical one in contrast to the purely theoretical one in the past. This shift of interest is the reflection of two recent phenomena: the much publicized "Software Crisis" and the promised development of the VLSI technology. The crisis due to the difficulty in constructing and maintaining reliable software in imperative (Von Neumann type) languages drives us to seek semantically simple languages such as functional languages as a very attractive alternative. At the same time the VLSI technology affords us to build hardware with a large number of processing elements, which in turn allows us to take advantage of a very large degree of concurrency in computation. Again functional languages, by their nature of side-effect freeness, distinguish themselves as powerful notational systems that are quite suitable for exploiting parallelism.

For functional programming to be practical, one must, of course, be able to write and maintain large "Functional Software" less costly (not prohibitedly costly). We believe, for this purpose, "methodologies" for functional programming are indispensable as they are for conventional programming. In particular, structuring and modularization of programs are of paramount importance. This paper concentrates on language issues in functional programming. First, we discuss program structuring and modularization concepts and propose a new functional language called FLADT (Functional Language with Abstract Data Types), which incorporates such concepts. The concepts discussed include abstract data types, higher order functions, type parametrization, and function modules. Next, we present a novel environment retention method which uniformly handles both higher order functions and type parametrization on conventional (Von Neumann type) architectures. Our method can implement a quite general type parametrization scheme more efficiently than the previously proposed methods [ALS78, Yua79].

Our adherence to conventional architectures is justified by our view that we must experiment and accumulate experiences of functional programming on conventional architectures until reasonably efficient architectures for executing functional program come to hand. Taking this view further, we also consider functional programming as "executable specification" writing. In this sense, our work is relevant to conventional programming methodologies which emphasize specification writing in every phase of program construction and maintenance.

I Structuring Concepts

The following concepts seem important in structuring and modularization of large functional programs. Those concepts are made concrete as language constructs in the language FLADT [Yon81, Saj82a, Saj82b].

1. Abstract Data Types

In order to write a program which solves a given problem in a natural fashion, one should be able to define (user-defined) new data types which naturally reflect data components of the problem domain. Furthermore, the user must be able to use such data types without knowing how they are represented. The user of such data types should rely on only their abstract properties. When the user defines a new data type, the description of its representation and the definitions of its associated operations should be textually put together and the linguistic mechanism (the language semantics) must prohibit any operations <u>not</u> associated with the type to be applied to data of that type. This is the idea of abstract data types. Of course, the concept of abstract data type has been developed [DDH72] and implemented in a number of imperative languages [e.g., CLU, Iota, Euclid, Ada] to provide the programmer with a powerful modularization vehicle. We feel that this concept of abstract data types is equally useful and FLADT language provides a facility for defining abstract data types as a major modularization construct. See the "fadt unit" in the next section.

2. Function Modules

When we define a group of functions which are used for a single complex task, the structure of programs is not easy to comprehend if the definitions of such functions are scattered in the program text. Thus we should provide some mechanisms to put the definitions textually together and form a linguistic unit. This mechanisms is used for grouping functions which are mutually related, but not associated through abstract data types. To enhance program modularity, the names of the functions which are used (or called) outside the linguistic unit should have their names explicitly stated and the semantics rule of language disallow the use of the other functions defined in the unit outside it. In FLADT, such a linguistic unit is called a module definition unit. "Modules" in Euclid [EUC77] and "packages" in Ada [Ada80] are similar linguistic constructs.

3. Higher Order Functions

A function which takes functions as parameters or returns a function as its result value is called a higher order function. Pascal and other conventional imperative languages provide facilities for higher order functions in restricted forms, yet the danger of side-effects substantially reduces its usefulness. In functional programming, however, the use of higher order functions is not only safe, but also extremely powerful in expressing certain kinds of computations naturally. (Henderson's book[Hen80] gives interesting examples of higher order functions such as a parsing program for context free languages. This program consists of definitions of the functions which are precise transliteration of corresponding BNF syntax rules and it seems as natural and succinct as ones written in Prolog for the same purpose.)

Higher order functions also provide us with means to structurize and schematize descriptions of algorithms through functional parametrization and the resulting descriptions are often much shorter and easier to comprehend. This is another important merit to use higher order functions.

A word of caution: considerable care should be taken to avoid miss-match of parameter types in using higher order functions. So we think type specifications for functional parameters and result functions should be made explicit in definitions. (The syntax of FLADT requires explicit type specifications in interfaces of function definitions.)

4. Type Parametrization

One of the most important disciplines for writing reliable software is to maintain the type consistency between operators and their operands (functions and their arguments) that are used in programs. When one wishes to cause similar effects to different types of objects, the type discipline requires us to write different programs individually. For example, two different sorting functions must be written for a sequence of integers and a sequence of character strings even if the same algorithm is used. If the type parametrization facility is provided, one need not write two different sorting functions; a single function with parametrization of object types in the sequence suffices.

The notion of type parametrization is not restricted to functions. The type of components constituting an abstract data type can also be parameterized. Moreover, the data types appearing in the definitions of functions which comprise a function module can be parameterized. Thus the use of type parametrization in various parts of a software system contributes to the reduction of the system size and, of course, it structurizes the whole system.

It should be noted that types to be parameterized often need to satisfy some conditions. For example, the types parameterized in the sorting functions mentioned above must have order relations which are identified by the same name. (E.g., the greater-than relations for integers and character strings must have the same predicate name.) Therefore this kind of restriction should be stated explicitly in the definitions of type-parameterized program units. But if type parametrization is used together with higher order functions, the restriction suggested above can be removed. For example, if the predicate which tests the order relation is abstracted as a <u>functional</u> <u>argument</u> in the type parameterized sorting function, the condition is automatically satisfied by the formal argument name, (which is of course unique) for the predicate.

III. The language FLADT

In this section, we discuss a new functional (applicative) language FLADT (Functional Language with Abstract Data Types) which incorporates the structuring concepts introduced in the previous section. The implementation of FLADT was written in our CLU system [Sad81] which also supports powerful abstraction features [CLU79]. Moreover, we discuss the FLADT system which we plan to implement as a total system for effective functional software development.

1. FLADT

Besides typical applicative language features [Lan66, Rey70], FLADT supports user controlled delayed/forced evaluation and higher order functions, and provides language constructs for data abstraction and function modularization. The data abstraction and the function modularization can be type parametrized.

Programs in FLADT consists of a sequence of units which are explained below. Data types (or data structures) are represented by the use of a rich repertoire of standard types and type generators. (Since FLADT is a functional language, functions and operators are described as expressions.)

1.1 Units

There are seven kinds of units.

(i) type definition unit
(collection of type definitions)
(ii) constant definition unit
(collection of expressions which are computable in compilation time)
(iii) function definition unit
(a definition of a global function)
(iv) fadt (functional abstract data type) definition unit
(definitions of abstract data types and their associated operators, see II.1.)
(v) module definition unit
(collection of function definitions, see II.2.)
(vi) expression unit
(an expression which can be executed)
(vii) interface unit
(collection of interface informations)

1-2 Expressions

(i) literal
(ii) name
(iii) data object constructor
(iv) type converter
(v) abort expression
(vi) prefix/infix expression

(vii) conditional expression
(viii) where expression
(ix) lambda expression
(x) function application
(xi) delaying/forcing expression
(xii) first/for expression (a la Backus's α)

1-3 Types and Type Generators

A type consists of a set of objects together with a set of operators to manipulate the objects. A type generator is a parametrized type definition, representing a set of related types.

Standard (built-in) types and type generators in FLADT are as follows.

```
types           -- null, bool, int, char, string, stream
type generators -- seq, prod, sum, delay, map
```

These type generators represent sequence type, Cartesian product type, direct sum (discriminated union) type, delayed type, and mapping type.

We can implement a new data type or type generator using a fadt definition. However, the number of type parameters for a defined type generator is fixed. (we cannot define a new type like a type generator prod or sum) An enumeration type like one in Pascal can be uniformly defined by a fadt definition using objects in FLADT.

Now, we show two checking algorithms concerning types; One is for well-formedness of type and another for equivalence of types.

The well-formedness of type is defined as whether or not an object of a type can be created in finite area. For instance, the following type

```
bad = prod[ item:int, rest:bad ]
```

is not well-formed because it is defined by non-terminating recursion.

The algorithms we use for the well-formedness checking and type equivalence are given in Fig. 3.1 and Fig. 3.2, respectively. Note that our algorithm for type equivalence is based on the structural equivalence. The well-formedness and type equivalence are performed by the following function calls.

```
wellf( T, Ø )              @ well-formedness of type T
equiv( T1, T2, Ø )         @ equivalence of type T1 and T2
```

```
func wellf(T:types,BFS:set):bool =
      @ BFS denotes a set of bad-formed types
      @   which can be recursively checked
  if set$is_in(BFS,T) then
      false
    else if T?sum then
      forsome t in types$elements(T)
        suchthat wellf( t, {T} ∪ BFS )
    else if T?prod then
      forall t in types$elements(T)
        suchthat wellf( t, {T} ∪ BFS )
    else if t?fadt_with_params then
      forall t in types$elements(T)
        suchthat wellf( t, {T} ∪ BFS )
    else    @ seq or mapping type
      true
  end wellf
```

Fig. 3.1

```
func equiv(Ti,Tj:types,ES:sets):bool =
      @ ES denotes a set of equivalence pairs
      @   of types which can be recursively checked
  if Ti?basic & Tj?basic then
      Ti = Tj
    else if Ti?basic xor Tj?basic then
      false
    else if Ti = Tj then
      true
    else if sets$is_in(ES,{Ti,Tj}) then
      true
    else
      forall ti,tj in types$elements2(Ti,Tj)
        suchthat equiv( ti, tj, {{Ti,Tj}} ∪ ES )
  end equiv
```

Fig. 3-2

1-4 Names and Objects

The basic elements of FLADT semantics are names and objects. Objects are the data entities that are created and operated by programs. Names are used in a program to refer to objects. We show this definitional framework in Fig. 3-3. A structured object denotes the object which is constructed with some related objects. A record type in Pascal is an example of a structured object. A function is an object which accepts objects as arguments and returns an object as its value.

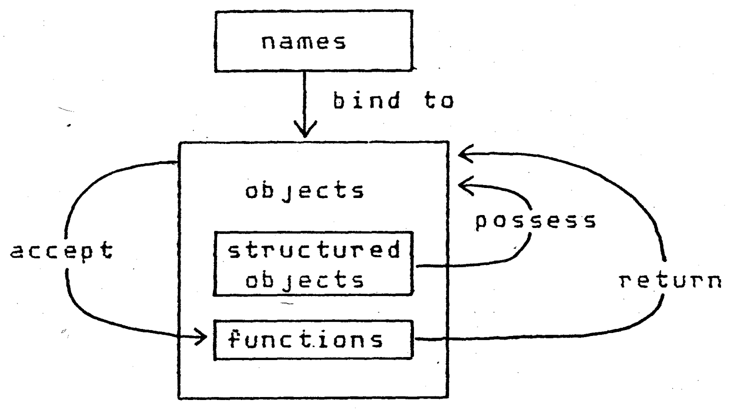


Fig. 3-3

2. Examples of FLADT Program

The program in Fig. 3-4 is an implementation of set type in the FLADT language. In general, a fadt (functional abstract data type) definition consists of two part: the interface part and the definition part. The interface part specifies the names and functionalities of the operators which are basic to the abstract data type being defined. ("$" denotes the type being defined.) When the fadt is type-parameterized, the restriction to the type parameters must also be specified in the interface part. In the case of the example program, the type of set elements must have "equal" as its basic operators with the specified in the definition part.

The definition part specifies the representation of the abstract data type being defined. In the example,

<u>type</u> <u>rep</u> = <u>seq</u>[t]

means that the set type is represented by a standard type generator "<u>seq</u>". The type identifier "<u>cvt</u>" has a special meaning. It does the conversion between the abstract data type and its representation. (The notion of "<u>cvt</u>" are borrowed from CLU.)

```
fadt set[ t ]
  interface
    set[t] :: empty:              -> $
              is_empty: $         -> bool
              insert:   $ * t     -> $
              equal:    $ * $     -> bool
              member:   $ * t     -> bool
              elements: $         -> seq[t]
         t :: equal:    t * t     -> bool
  end set

fadt set[ t ]
  definition
  restrict  t :: equal t * t -> bool

  type  rep = seq[ t ]

  op empty():cvt = rep${}

  op is_empty(s:cvt):bool = s?empty

  op insert(s:cvt,v:t):cvt =
    if set[t]$member( up(s), v ) then s
      else rep$apndl( v, s )

  op equal(s1,s2:set[t]):bool =
    ( forall x:t in elements(s2)
        suchthat member( s1, t ) ) &
    ( forall x:t in elements(s1)
        suchthat member( s2, t ) )

  op member(s:cvt,v:t):cvt =
    if s?empty then false
      else if s.hd = v then true
      else set[t]$member( up(s.tl), v )

  op elements(s:cvt):seq[t] = s

  end set
```

Fig. 3-4

To show the use of higher order functions and function modules in the FLADT language, we will give a recognizer program for a simple context free grammar. The original program in a simple functional language is given in Henderson's book [Hen80]. The defined language can be specified by BNF as follows. (C,V denote terminal symbols)

```
<cseq>      ::=   C  |  C  <cseq>
<vseq>      ::=   V  |  V  <vseq>
<syllable>  ::=   <cseq>  <vseq>
              |   <vseq>  <cseq>
              |   <cseq>  <vseq>  <vseq>
```

Fig. 3-5

The above grammar is intended to describe the common forms of syllables in an English word. Here C stands for "consonant" and V for "vowel".

The program given in Fig. 5-2 is the function module which defines a recognizer for the grammar. The module consists of seven functions: "syllable", "cseq", "vseq", "orp", "seqp", "vowel", and "consonant". Those functional whose definitions are prefixed with "%fn" are internal functions that cannot be referred to from outside the module. The internal functions are used to define the function "syllable which is referred to from outside the module.

```
module syll

   fn syllable(x:string):bool =
     orp( seqp( cseq, vseq ),
          orp( seqp( vseq, cseq ),
               seqp( cseq, seqp( vseq, cseq ) ) ) )

   %fn cseq(x:string):bool =
     orp( consonant, seqp( consonant, cseq ) )

   %fn vseq(x:string):bool =
     orp( vowel, seqp( vowel, vseq ) )

   %fn orp(p,q:string->bool)(x:string):bool =
     if p(x) then true else q(x)

   %fn seqp(p,q:string->bool)(x:string):bool =
     if x = "" then
         cand( p(""), {q("")} )
       else if cand( p(""), {q(x)} ) then
         true
       else
         seqp( λ(y:string):bool.p(string$substr(x,1,1) & y),
               q ) (string$rest(x,2))

   %fn vowel(s:string):bool = s = "V"

   %fn consonant(s:string):bool = s = "C"

   end syll
```

Fig. 3-6

Note that "orp" and "seqp" are higher order functions that take boolean functions defined on the string domain) as parameters. In the definition of "orp" and "seqp", p and q stand for formal functional arguments and the notation:

(x:string):bool

indicates that the returning values of both functions are also boolean function defined in the string domain. The actual argument of the recursive invocation of "seqp":

λ(y:string):bool.p(string$substr(x,1,1)&y)

is a boolean function whose argument is a string type variable y and "&" stands for string concatenation.

It should also be noted that the use of the higher order functions enables us to make the form of the "syllable" definition a precise transliteration of the BNF definition of the grammar.

IV. A New Environment Retention Method

In implementing higher order functions on conventional architectures, how environments should be managed is a critical point for efficiency. The main issue is the interpretation of free variables, namely the FUNARG problem.

Though many environment management methods have been proposed, all of them are not sufficient in expressive power or execution speed.

This section presents a new environment retention method for free variables (called FFV method), assuming the static binding rule is adopted as in the FLADT language. Our method can treat upward/downward FUNARGs uniformly with practical efficiency and furthermore type parametrization can also be implemented by this method.

We introduce a hypothetical language L which permits upward/downward FUNARGs. The main features of L are as follows: 1) L has a block structure like Pascal's for the scope rule, 2) L has unnamed functions (λ-expressions), 3) and in L, any value including functions is permissible as arguments or result of functions. Thus, in L, data objects are generally allocated in heap for it is impossible to determine the life-time of objects by the rule of block structure.

terminology

(i) λ denotes a function

(ii) λ^c denotes a function closure created by evaluating a function λ.

(iii) invocation of λ denotes that λ (more precisely λ^c) is applied to its arguments and then its body is evaluated.

(iv) BODY(λ) denotes λ's body

(v) FREE(λ) denotes the set of free variables in λ

(vi) LOCAL(λ) denotes the set of formal parameters and local variables in λ

(vii) The nesting level of a function λ is denoted by NL(λ). We define NL(λ) = 1 if λ is a global function. Let a function defined in λ_α be denoted by $\lambda_{\alpha,i}$ (i>0), then $NL(\lambda_{\alpha,i}) = NL(\lambda_\alpha) + 1$. α is an sequence of indices k1, k2, . . . , km (m>1).

1. Basic Concepts

To treat upward/downward FUNARGs uniformly, it is sufficient for invocation of λ that all the values of free variables in λ are retained. This idea enables us to make distinction between upward and downward FUNARGs. At the evaluation of function λ, we may create a function closure λ^c with embedded values of free variables. Namely, λ^c is represented by the following triple. (VL denotes a value list of free variables)

< LOCAL(λ), BODY(λ), VL >

In many conventional languages, an environment frame corresponds to a procedure (or a function) and all of the local variables are converted to distinctive values (called displacement) in the environment frame. Thus, the access to local variables is very efficient.

When λ^c is invoked, we copy from VL of λ^c to λ's frame, and when we access to free variables, we may use their displacement in the frame as local variables. Since all of the variables in λ are converted to the displacement in its frame, there is no distinction between free and local variables.

We should note that $\lambda\alpha$ is already invoked when $\lambda\alpha, i$ is evaluated: this is a natural relation derived from the static binding rule. In other words, $\lambda\alpha, i$ is always evaluated in the environment of $\lambda\alpha$.

We can summerize our method as follows.

(i) BODY($\lambda\alpha$) is evaluated with the use of displacements in the $\lambda\alpha$'s frame

(ii) $\lambda\alpha$ is already invoked when $\lambda\alpha, i$ is evaluated.

(iii) evaluation of $\lambda\alpha, i$ means construction of $\lambda^c\alpha, i$. $\lambda^c\alpha, i$ is constructed by copying the values of free variables from the $\lambda\alpha$'s frame.

(iv) When $\lambda\alpha, i$ is invoked, values in $\lambda^c\alpha, i$'s VL are copied to the $\lambda\alpha, i$'s frame.

Next, we show that the following two conditions for efficient execution are held:

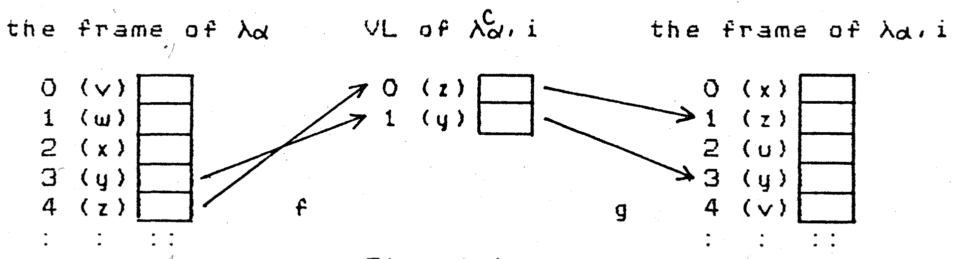
(a) All the values of the free variables of $\lambda\alpha, i$ are already in the $\lambda\alpha$'s frame when \α, i is evaluated.

(b) The cost of creating a closure $\lambda^c\alpha$ is $\mathcal{O}(n)$ ($n=|\mathrm{FREE}(\lambda\alpha)|$), and the cost of copying from VL of $\lambda^c\alpha$ to the frame of $\lambda\alpha$ is also $\mathcal{O}(n)$.

The proof of (a) is give in the Appendix.

Proof of (b).

We already know that $\lambda\alpha, i$ is always evaluated in the environment frame of $\lambda\alpha$. Each variable has been converted to the displacement with any rule (occurrence order, lexicographic order, etc.) (See. Fig. 4-1) So, we may determine the mapping function f and g. f denotes a mapping between the displacement of free variables of $\lambda\alpha$ to the displacement of VL of $\lambda^c\alpha, i$, and g represent a mapping between the displacement of VL of $\lambda^c\alpha, i$ to the displacement in the frame of $\lambda\alpha, i$. f and g are easily determined at compilation time. Note that, using f and g, we can satisfy the cost mentioned in (b). $\square$



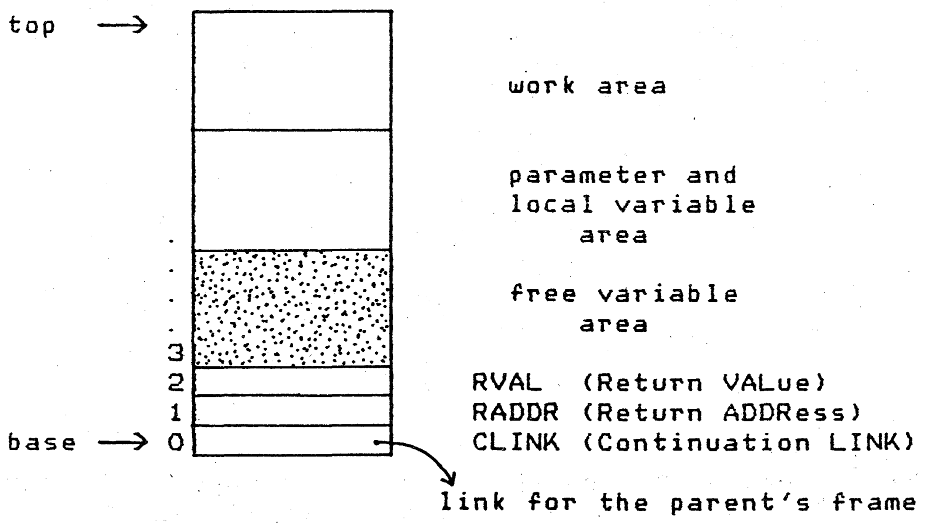
the frame of λ_α
VL of $\lambda^c_{\alpha,i}$
the frame of $\lambda_{\alpha,i}$
0 (v)
1 (w)
2 (x)
3 (y)
4 (z)
0 (z)
1 (y)
0 (x)
1 (z)
2 (u)
3 (y)
4 (v)
f
g

Fig. 4-1

2. Frames with embedded Free Variables (FFV)

2.1 FFV

We introduce a new frame structure called FFV (Frames with embedded Free variables) based on the basic concept explained in the previous subsection.



top
work area
parameter and local variable area
free variable area
3
2
1
0
base
RVAL (Return VALue)
RADDR (Return ADDRess)
CLINK (Continuation LINK)
link for the parent's frame

Fig. 4-2

We can create FFVs in the heap area as in the case of other data objects, but by using the stack area we gain more efficiency in execution speed. Then, we can uniquely determine the mapping function g which is introduced in the previous sub section, as $g(x)=x+3$ for all functions, if we can take a continuous area for free variables as in Fig 4-2.

2.2 Properties of FFV

(i) There is no static-chain (access-chain).

(ii) The object code for variables becomes compact. On conventional compilers, it has a pair attribute such as <block-depth, displacement>, but on the compiler using FFV, only the displacement suffices.

(iii) The size of FFV is usually greater than that of a conventional frame like one in Pascal. (Remark: if the number of free variables is zero, the size of FFV is smaller than that of conventional one because of the extra area for static-chain.)

(iv) If a variable defined in $\lambda\alpha$ is only referred to in $\lambda\alpha, k1, \ldots, km$, it is also free in $\lambda\alpha, k1$, $\lambda\alpha, k1, k2$, $\lambda\alpha, k1, \ldots, k(m-1)$, so it is embedded in their frames.

(v) Values of free variables are copied when λ is evaluated or the frame of λ is constructed. 2*n copy operations are required. (n denotes $|FREE(\lambda)|$)

(vi) Every time λ is evaluated, λ^c (closure of λ) is created.

(vii) Whenever λ is invoked, objects for local variables of λ (ref objects in Algol 68) are created.

Note that (v) and (vi) consume considerable execution time. Now, we show some methods for improvement for each case.

(v)-1 Since the distinction between local variables and free variables can be made at compile time, we can generate code for free variables in the form of indirect referencing. Thus we can omit copying VL of λ^c to λ's frame. (See below)

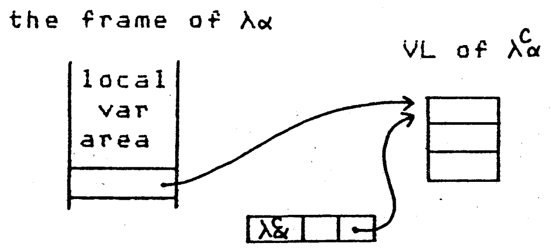


Fig. 4-3

(v)-2 We extend the above idea. If $FREE(\lambda\alpha) = FREE(\lambda\alpha, i)$, the same VL should be allocated to $\lambda\alpha$ and $\lambda\alpha, i$.

(vi)-1 When λ is invoked just after λ^c is created, λ^c is, in fact, not necessary. In this case, we directly construct the frame of λ without creating λ^c.

(vi)-2 A function λ which has no free variables is always evaluated to the same closure object. Once the closure object is created, that object will be used at every evaluation of the λ.

The above improvements bring us a practical efficiency on the FFV method.

2.3 Comparison with Bobrow-Stack

In our model, we treat the free variables directly in the frame with embedded their values, (c.f. shallow binding) so we can access to variables quickly, but context change is a little bit slow. In the Bobrow model, we trouble with the referencing to variables, but the cost of context change is small. (c.f. deep binding) Moreover, the Bobrow model is possible to store some parameters in the frame for handling backtracking, coroutines, and unorthodox control mechanisms with practical efficiency. The FFV method can handle these control mechanisms by storing necessary informations in the frame.

3. Type Parametrization Problem

In FLADT, CLU, and Iota, we can treat types as parameters. This feature enhances modularization of programs. On the other hand, it often becomes a heavy burden to language processors. Several implementation methods for type parametrization have been proposed [ALS78, Yua79, Sad81]. Yet none of them are sufficient: the method reported in [ALS78] generates code which handles actual type parameters at runtime and the area for various table tends to be large. The method by T. Yuasa [Yua79] requires complicated table management and needs more overhead. K. Sado [Sad81] uses compile time macro, but the power of type parametrization must be restricted.

The method we propose here is based on the FFV method and does not have the shortcomings of the previous methods. (In some cases, our method is somewhat slower than the compile time macro method, but no restriction to parametrized types is necessary.)

The FFV method provides us with a solution to this problem. Though our solution is less efficient than that of CLU's in some cases, our solution need not any support routines for treating the problem.

We use the fadt definition of set type in section III to explain our solution (rewritten in Fig. 4-4). The type parameter t in Fig. 4-4 needs an equal operator as a restriction. (In Iota, type t is declared as sype. See [Yua79])

```
fadt set[ t ]
  definition
  restrict  t :: equal t * t -> bool
    ....
  op empty():cvt = ...
    .....
  op member(s:cvt,v:t):bool =
    .....    t$equal   .....
    .....
  end set
```

Fig. 4-4

Our idea is to regard the operator T$equal as a free variable of the operator set[T]$member. (Now T$OPR denotes the operator OPR associated with the abstract data type T) In other words, to evaluate set[int]$member is equivalent to creating a closure with embedded int$equal as a free variable.

Similarly, we consider the following case.

```
set[ u ]$member                @ u is also a type parameter
```

This operator evaluation is proper if and only if u also has an equal operator as a restriction such as in Fig. 4-5. Obviously, there is the operator T$equal (we suppose that the actual type parameter associated with u is T) in the environment frame when set[u]$member is evaluated. Now, you will know how to use the FFV method.

```
fadt example[ u, v ]
  definition
  restrict  u :: equal   u * u -> bool
                 similar u * u -> bool
            v :: .....
    .....
  op exam .....
    .....    set[ u ]$member
    .....
  end example
```

Fig. 4-5

Furthermore, we substitute the above set[u]$member for

```
set[ set[u] ]$member.          @ u is still a type parameter
```

set[...]$member needs an equal operator, and type set[u] needs T$equal, then the evaluation of set[set[u]]$member is the creation of the following closure object.

set[set[t]]$member

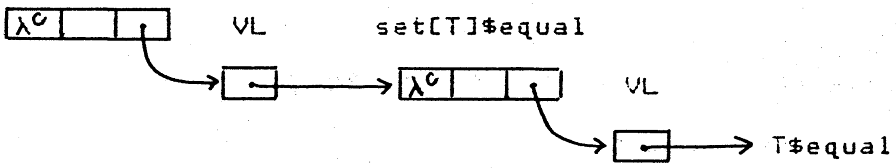


Fig. 4-6

(In this case, each size of VLs happens to be the same.)
The compiler has to generate the code which creates such a closure object. It is possible but not easy.

The above discussion reveals that the type parametrization can also be treated by the FFV method. But it has a poor efficiency as it is. So, we can perform some optimizations introduced in the previous section to the FFV method. The optimizations on usual programs cause remarkable effects:

1) When the actual type parameter contains no formal parameters, (e.g. set[int]$member) the closure object of the operator is created only once.
2) When the operator with the type parameters calls the operator with the same type parameters (for example, the operator set[t]$member directly calls itself with the same type parameter t), the closures associated with the calls are created only once. More precisely, when several operators which have the same restriction with type parameters are called, the same VL is used in each closure. In this case, we need not treat closures as value, they need not be created at all.

The above-mentioned is applied to not only the simple case such as one type parameter, but also to more general cases such as in Fig. 4-7.

```
fadt general[ t1,t2,...,tn ]
  definition
  restrict  t1  :: op11 ...
                   op12 ...
            t2  :: op21 ...
                     ...
              .....
            tn  :: opn1 ...
                     ...
                   opnm ...
    .....
    .....
  end general
```

Fig. 4-7

V. FLADT System

This section describes an interactive FLADT program development system. It consists of a FLADT language processor (translator, interpreter, optimizer), database for interface information and various editors.

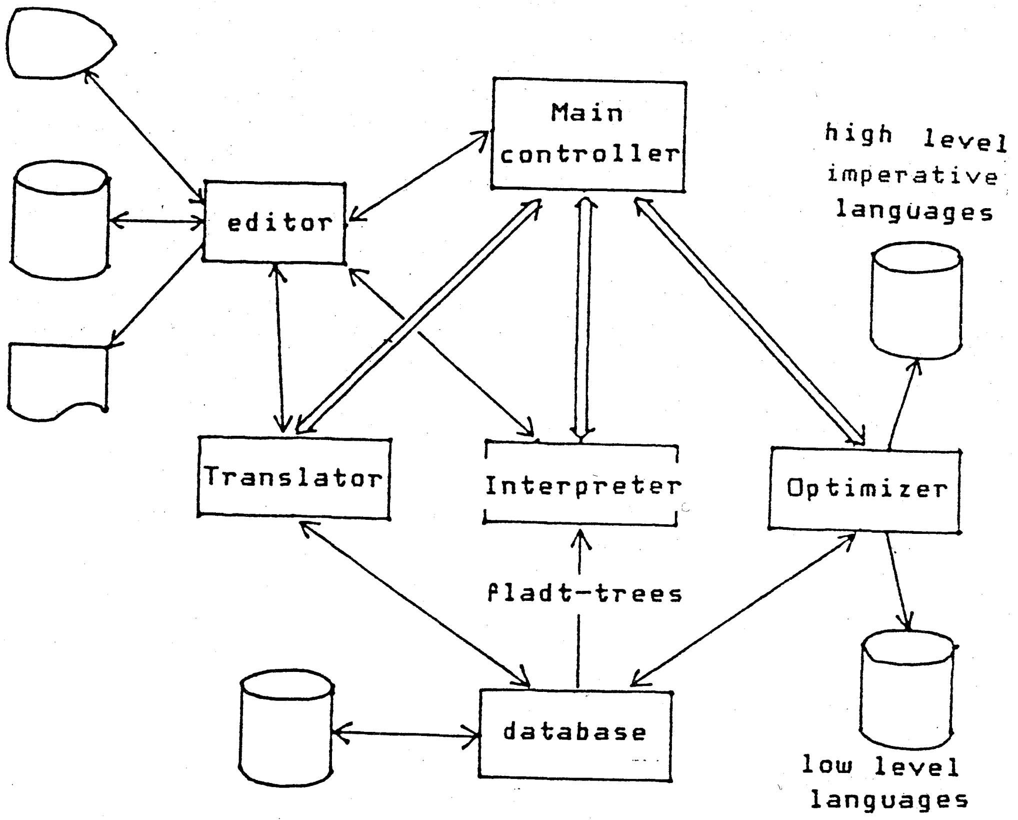


Fig. 5-1

1. Translator

The translator translates source programs in FLADT into intermediate tree structures called fladt-trees. At translation time, it makes rigid checking about types using the informations stored in the database.

2. Interpreter

The interpreter directly interprets fladt-trees.

3. Optimizer

This part is not implemented yet; we are still studying algorithms for optimization, but the planned optimizer is supposed to do the following things.

(i) Optimizes fladt-trees into fladt-trees.
(ii) Translates fladt-trees into programs written in high level imperative languages such as Pascal.
(iii) Translates fladt-trees into FLADT machine code or

programs written in low level languages.

4. Database

The FLADT system has a database of interface information and dependency information about FLADT units. The database is constructed by extracting information from interface units. And the database holds the following information. (See Fig. 5-2, 5-3)

(i) Compile states of units (compiled or uncompiled)

(ii) List of usage information (list of external operators and functions)

(iii) Modification information (notification and detailed information about the modified operators, functions, and interface specifications)

5. Simple Programming Support System

The language FLADT can be considered as an <u>executable specification language</u> which facilitates the software development based on hierarchical abstraction.

We may abstract data structures as abstract data types, and then implement fadt definitions hierarchically. These definitions are stored in the database as a unit. Tests for each unit are easily done by the interpreter. Once all the units are tested, the optimizer translates all the programs into more efficient code.

In such a system, we must guarantee the consistency of interfaces for each unit. A modification to a unit A is notified to all other units associated with A by the system automatically. The notified units are changed to the uncompiled state.

When modifying a unit, the user mainly manages the database for the unit and its related uncompiled units. All the modifications are done to source programs, not directly to fladt-trees. We can debug programs interactively using the interpreter on fladt-trees.

The above programming support system has the following advantages.

(i) The consistency between source programs and object code is always guaranteed.

(ii) The use of the interpreter provides us with effective informations for debugging.

(iii) The modification of interface specification are easily and safely done with the database of interface information and dependency information.

FLADT System

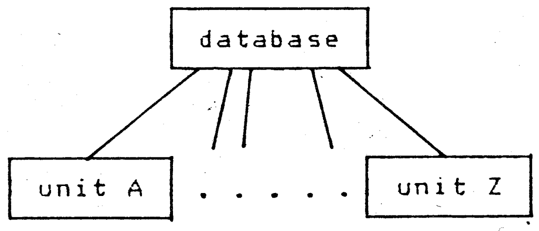


Fig. 5-2

```
unit A  : compiled

unit B  : compiled
  uses unit C, D, E
  .......
...............

unit Z  : XXXXXXXX
  unit A(op1) modified
     .....
  uses unit A
```

Fig. 5-3

Acknowledgement

The first author wishes to express deep appreciation to the members of the Software Product Engeneering Laboratory for their patient guidances and helps. Expressly he acknowledges director Koji Nezu for his encouragements during this work, manager Masanori Teramoto and supervisor Takekuni Kato for their helpful comments. He also would like to thank Emi Miyamoto for her kind help in preparing the draft.

The second author's research has been supported by the funds from the Ministry of Education, Science and Culture, No. 802-579016, No. 802-56790025 and No. 802-57780032.

References

[Ada80] U.S. Dept. of Defense: Reference Manual for the Ada Programming Language (July 1980).

[ALS78] Atkinson, R., Liskov, B., and Sheifler, R.: Aspects of Implementing CLU, Computation Structures Group Memo 167, MIT LCS (Oct. 1978).

[Bob73] Bobrow, D.G.: A Model and Stack Implementation of Multiple Environments, CACM Vol.16, No.10 (1973), 591-603.

[CLU79] Liskov, B.H., Snyder, A., et al.: CLU Reference Manual, TR-225, MIT LCS (Oct. 1979).

[DDH72] Dahl, O.-J., and Hoare, C.A.R.: Hierarchical Program Structures, in Structured Programming, Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R., Academic Press, London (1972), 175-220.

[EUC77] Lampson, B.W., et al.: Report on the Programming Language Euclid, SIGPLAN Notice, Vol.12, No.2 (1977).

[Hen80] Henderson, P.: Functional Programming, Prentice-Hall, Inc., London (1980).

[Lan66] Landin, P.J.: The Next 700 Programming Languages, CACM Vol.9, No.3 (1966), 157-166.

[Rey70] Reynolds, J.C.: GEDANKEN - A Simple Typeless Language based on the Principles of Completeness and Reference Concept, CACM Vol.13, No.5 (1970), 308-319.

[Sad81] Sado, k.: Practical Implementation of Programming Language CLU and Experience with it, (in Japanese) Transactions of Information Processing Society of Japan, Vol.22, No.4 (July 1981), 295-303.

[Saj82a] Saji, N.: Design and Implementation of a Functional Language with Abstract Data Types FLADT (in Japanese), MS thesis, Department of Information Science, Tokyo Institute of Technology (March 1982).

[Saj82b] Saji, N.: FLADT Reference Manual Version-I (in Japanese), Department of Information Science, Tokyo Institute of Technology (Jan. 1982).

[Yon81] Yonezawa, A.: On FLADT, in Software Concepts for New Computer Architectures, (Ohno, Y. ed) Report of Research Supported by the Ministry of Education, Science and Culture, (in Japanese) (March 1981).

[Yua79] Yuasa, T.: Module-wise Compilation for a Language with Type-parametrization Mechanism, RIMS-280, Proc. of RIMS No.363 (1979), 1-40.

Appendix

Proof of (a).

The proof is done by induction on the level of nesting. Before we start the proof, we must notice the following properties

(i) $FREE(\lambda\alpha) \cup LOCAL(\lambda\alpha) \supseteq FREE(\lambda\alpha, i)$ for all α, i
(ii) $\lambda\alpha$ is already invoked when $\lambda\alpha, i$ is evaluated.
(iii) That the frame of $\lambda\alpha$ is well-formed is defined as follows.
  (for all $v \in FREE(\lambda\alpha) \cup LOCAL(\lambda\alpha)$)
   [the value of v is in the $\lambda\alpha$'s frame]
(iv) That $\lambda^{c}\alpha, i$ is well-formed is defined as follows.
  (for all $v \in FREE(\lambda\alpha, i)$)
   [the value of v is in the $\lambda\dot{\alpha}$'s frame]
(v) $\lambda\alpha$'s frame is well-formed, if $\lambda^{c}\alpha$ is well-formed.

The proof might be obvious to the reader familiar with the properties (i) and (ii).

Induction on nesting level

I. $FREE(\lambda k)=\emptyset$ for all k.
 λk is well-formed. (by definition (iv))
 The frame of λk is well-formed. (by definition (iv))
 (λk denotes a global function. free variables of a global function represent either global variables or global functions. Both of them have fixed locations, so we can treat them as constants. Thus, $FREE(\lambda k)=\emptyset$)

II. Suppose that the frame of $\lambda\alpha$ (for all α) is well-formed. For all α, i, $\lambda\alpha, i$ is always evaluated in the frame of $\lambda\alpha$. (by definition (ii)).
 Definition (iv) is held. (by definition (i),(iii))
 i.e. $\lambda^{c}\alpha, i$ (for all i) is well-formed.
 So, the frame of $\lambda\alpha, i$ is well-formed.

From I and II, all the frames of λ are well-formed. ■