数理解析研究所講究録
第 547 巻 1985 年 196-229

Non-Strict Partial Computation with a Dataflow Machine

Satoshi ONO, Naohisa TAKAHASHI and Makoto AMAMIYA

Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation
3-9-11 Midoricho Musashino-shi Tokyo 180 Japan

Abstract

This paper proposes a new partial computation method for functional programming languages, named the projected function method. This method makes it possible to execute general partial computation without the pre-binding capability. Pre-binding is essential to the partial computation of non-strict functions in the conventional method, but is quite difficult to implement in dataflow machines.

This paper also presentes a new concept named a Dependency Property Set (DPS). The DPS indicates the dependency relation between parameters of functions and result values. This concept plays an important role in the projected function method. An algorithm to compute DPSs based on data flow analysis is also shown.

The projected function method has an excellent conformity with the dataflow computation model. Therefore, this method offers promises for realizing highly-parallel and highly-effective functional programming machines.

Index terms: Functional programming, tabulation, dataflow analysis, dependency analysis, reduction, normalization

1. Introduction

Partial computation is customizing a general program into a more efficient program based upon its operating environment [1]. This concept is useful for pattern matching, syntax analysis, compiler generation and so on [1]. Functional programming languages [2] have clean mathematical semantics, and are especially suitable for automated partial computation. A partial computation algorithm has been discussed for the class of recursive program schemata [3], and attempts have been made to develop partial computation programs for LISP language [4,5].

In contrast to the theoretical or interpreter based approach, machine architecture that can execute partial computation directly has not yet been proposed. The authors have proposed a new dataflow computation model named Generation Bridging Operator (GBO) model, and have provided detailed discussions on one category of the GBO model named the Dynamic-Coloring Static-Bridging (DCSB) model [6]. Although the DCSB model has a parallel partial computation capability, this model is limited only to the partial computation of strict functions [6].

This paper presents a new partial computation method named the Projected Function method. The method makes it possible to execute general partial computation only with the restricted computational power of the DCSB model.

The most important concept in this method is the notion of the Dependency Property Set (DPS). The DPS indicates the dependency relation between parameters of functions and result values. An algorithm to compute DPSs based on the data flow analysis method [7] is also presented.

2. Dependency Property Set

2.1 Functional Programming Language

This sub-section introduces the functional programming language used in this paper. This language is similar to Valid [8], and is the same as the language used by Ono et al.[6].

The factorial function can be defined as follows:

```
fact = ^[[n] if n==0 then 1 else n * fact(n-1) fi ]
```

The above expression is an example of a _function definition_. The identifier "fact" is a _function name_, and "n" is a _formal parameter_. The right-side of the equation (i.e. "^[[n] if ... fi]") is referred to as a _function_, and "if ... fi" is a _function body_. The name "n" is a _formal parameter_. If more than one formal parameter exists, they should be separated by a comma, and enclosed by a square bracket such as "[p1,p2,...,pn]".

Computation is the combination of _function applications_ and _simplifications_. A function application replaces the function name with its body, and substitutes actual parameters for formal parameters. Some functions are _primitive_ and defined as axioms. Replacing a primitive function application with its resultant value, is called a simplification. In the following, infix operators such as "+", "*", "==" as well as "if-then-else-fi" are assumed to be primitive.

For example, computation of fact(3) are shown.

```
fact(3) = ^[[n] if n==0 then 1 else n * fact(n-1) fi ](3)
        = (if 3==0 then 1 else 3 * fact(3-1) fi)
        = (3 * fact(2))
        = (3 * (2 * (fact(1))))
        = (3 * (2 * (1 * (1))))
        = 6
```

A value definition equates its left-side identifier (or a value name) to its right-side expression. A block expression is a sequence of expressions enclosed by "{" and "}". The value of the block expression is determined by a return expression in the block expression. The order of the expressions in a block expression has no meaning. Therefore, in the following example, all expressions have the same meaning :

```
{ y=x-1 ; z=y**2 + 2*y + 3 ; return z }          (2.1)
{ z=(x-1)**2 + 2*(x-1) + 3 ; return z }          (2.2)
{ z=y**2 + 2*y + 3 ; y=x-1 ; return z }          (2.3)
```

where the expression (2.2) can be obtained by substituting "y" in the expression (2.1) with "x-1", and the expression (2.3) can be obtained by transposing the first and second value definitions in the expression (2.1).

Function names and value names are generically called variables. Identifiers defined in the block expression are named bound variables as are the formal parameters in the function body. The scope of the value/function definitions placed in a block expression is limited to within the block expression. This language adopts the static binding rule. Namely, a free variable in a block expression is bound to the formal parameter of the lexically closest surrounding function definition or to the function/value definition of the lexically closest surrounding block expression [8].

2.2 Computation Rules

A computation rule determines the evaluation order when actual parameters of a function application themselves contain other function applications. The Parallel-Innermost Computation (PIC) rule, and the Parallel-Outermost Computation (POC) rule are especially important for parallel computation.

The PIC rule first selects all actual parameters for evaluation, and then, a function is applied to those parameters. The POC rule first applies a function to unevaluated actual parameters. These parameters are evaluated at the time their values are actually required for expression evaluation.

Hereafter, in an analogy to the case for sequential computation, the PIC rule is called call-by-value, while the POC rule is labeled call-by-name.

2.3 Definition of Dependency Property Set

Sections 2.3 and 2.4 provide the definitions of the DPS and related concepts required in the following discussions. In this paper, primitive functions are always assumed to be monotonous [9], and call-by-name is assumed unless explicitly specified as otherwise.

(1) Requisite parameter

Given a function "f" which takes parameters "x1", "x2", ..., "xn", a parameter "x1" is said to be a requisite parameter of "f", if and only if both of the following conditions are satisfied:

(a) The function "f" is not a totally undefined function. Namely, "f" returns a defined value for some "x1", "x2", ...,"xn".

(b) If "x1" is undefined, "f" is always undefined for any "x2", ..., "xn".

(2) Strict function

A function "f" is said to be strict if and only if all of its parameters are requisite parameters.

For example, primitive arithmetic functions such as "+" (addition), "-" (subtraction), "==" (equality) etc. are strict functions, whereas an "if-then-else-fi" function, a parallel-or (that returns a "true" when one of the parameters is "true" even if some parameters are undefined) are non-strict functions.

(3) Sufficient parameter set

Given a function "f" which takes parameters "x1", "x2", ..., "xn". A parameter set { x1, ..., xm } ($m \leq n$) of "f" is said to be <u>sufficient</u> if and only if the function "f" returns a defined value for some "x1", ..., "xm", even if the rest of parameters "xm+1" ..., "xn" are undefined.

As can be easily shown, the union of sufficient parameter sets is also a sufficient parameter set, and the requisite parameter of "f" is always included in any sufficient parameter set of "f".

(4) Minimally sufficient parameter set

Given a function "f" which takes parameters "x1", "x2", ..., "xn", a sufficient parameter set S = { x1, ..., xm } ($m \leq n$) is said to be <u>minimal</u>, if and only if there exists a case such that, for some "x1", ..., "xm" where the function "f" returns a defined value, then "f" always becomes undefined when one or more of the parameters in "S" become undefined.

(5) Dependency property set (basic idea)

A <u>dependency property set</u> (DPS) of a function "f" is a set which contains all minimally sufficient parameter sets of "f" as elements. (This definition is extended in the next sub-section.)

For example, suppose that

```
    add_3   = ^[[x,y,z] x+y+z] .
```

Then, all parameters are requisite parameters and a sufficient parameter set is uniquely determined as {x,y,z}. Thus, the DPS of this function is { {x,y,z} }. In general, the DPS of strict functions consists of only one element that is a set of whole parameters. The DPS of constant functions is { {} }, and the DPS of totally undefined functions is { }.

Consider the non-strict function

```
    if_func = ^[[x,y,z] if x then y else z fi] .
```

In this case, only "x" is a requisite parameter. In addition, {x,y}, {x,z}, {x,y,z} are sufficient parameter sets, whereas {x,y}, {x,z} are minimal. Therefore, the DPS of "if_func" is { {x,y}, {x,z} }.

The intersection of all elements in the DPS of "f" is a set of requisite parameters of "f". In addition, if a formal parameter "x" of "f" is not contained in the union of all elements in the DPS of "f", the parameter does not affect the result of "f".

For example, suppose

```
    f = ^[[x,y,z] if x>0 then x else f(x+y,y,x+z) fi]
```

The DPS of "f" is { {x}, {x,y} } (The algorithm to compute the DPS will be discussed in Section 4). The intersection of all elements in the DPS is {x}, and the union is {x,y}. Therefore, "x" is a requisite parameter of "f" whereas "z" is never used in "f".

2.4 The DPSs in Functional Programming

(1) Dependency property set (general)

In the functional programming language described in Section 2.1, the concept of the DPS should be generalized.

[Example 2-1]

```
  { f=^[[x,y] if x>0 then f(x-1,y+1) else g(x,y) fi];
    g=^[[x,y] if x==0 then y else f(-x,-y) fi] }
```

The DPS of "f" depends upon the DPS of "g" as well as the DPS of "f" itself. Following notation is used to describe the DPS of "g(x,y)":

(g {{x}} {{y}})

where "g" is a function name, and {{x}}, {{y}} are DPSs corresponding to the DPSs of the first and second parameters for "g", respectively. Therefore, the DPS of "f" can be written as:

{ {x,(f {{x}} {{y}})}, {x,(g {{x}} {{y}})} }

The DPS can also be defined for expressions.

[Example 2-2]

```
exp = { x = p + q;  y = x * x;  return y }
```

Then, the DPS of "x" is { {p,q} }. The DPS of a block expression is the DPS of its return value. Therefore, the DPS of "exp" is equal to the DPS of "y", and is equal to { {p,q} }.

Formally, the syntax of DPSs can be described as follows:

[Definition 2-1] The syntax of DPSs

```
DPS       = { MSPS-seq }
MSPS-seq = MSPS | MSPS, MSPS-seq
MSPS      = { P-seq }
P-seq     = P | P, P-seq
P         = value-name | function-application-form
function-application-form = (function-name parameter-DPS-list)
parameter-DPS-list        = DPS | DPS parameter-DPS-list
value-name    = variable
function-name = variable
variable      = IDENTIFIER
```

where the statement "a = x" means that "a" is defined as "x", and the statement "a = x | y" means that "a" is defined as "x" or "y".

(2) Tagged DPS (TDPS)

The DPS is an attribute of functions and values. To explicitly declare such relations, tagged DPSs are used.

[Definition 2-2] The syntax of TDPSs

```
TDPS  =  (function-name  (formal-parameter-list)  DPS) |
         (value-name      "EMPTY"                  DPS)
formal-parameter-list = formal-parameter |
                        formal-parameter formal-parameter-list
formal-parameter      = variable
```

Dependency Property Set

For example, the DPS of "f" in Example 2-1 is described using a TDPS as follows:

```
   ( f  (x y)  { {x,(f {{x}} {{y}})}, {x,(g {{x}} {{y}})} } )
```

Similarly, the TDPS of "x" in Example 2-2 is

```
           ( x  EMPTY  { {p,q} } )
```

The leftmost field of a TDPS is called the <u>name of the TDPS</u>.

(3) Dependency environment

To analyze the dependency of variables, it is desirable to gather all the TDPSs visible in a given scope. For this purpose, a set of TDPSs named a <u>dependency environment</u> (DE) is introduced.

[Definition 2-3] The syntax of dependency environments is

```
DE       = { TDPS-seq }
TDPS-seq = TDPS | TDPS, TDPS-seq
```

For example, the DE of a block in Example 2-1 is

```
  { ( f  (x y)  { {x,(f {{x}} {{y}})}, {x,(g {{x}} {{y}})} } ),
    ( g  (x y)  { {x,y}, {x,(f {{x}} {{y}})} } ) }          (2.4)
```

Similarly, the DE of a block in Example 2-2 is

```
        { ( x  EMPTY  { {p,q} } ),
          ( y  EMPTY  {  {x}  } ) }                          (2.5)
```

(4) Normal form of DPSs

The concept of free variables is extended to DEs. A variable "n" of a TDPS "d" is said to be <u>free</u> in the DE "E" if and only if

(a) "n" does not appear in any name of the TDPSs in "E", and

(b) if "d" is a TDPS of a function, "n" does not appear in the formal parameter list of "d".

For example, "p" and "q" in the DE (2.5) shown above are free, whereas "x" is not free.

A TDPS "d" is said to be in a <u>normal form</u> in a DE "E", if and only if

(a) if "d" is a TDPS of a function, the DPS of "d" contains only formal parameters of "d" and free variables in "E".

(b) if "d" is a TDPS of a value, the DPS of "d" contains only free variables in "E".

A DPS is said to be in the normal form in a DE "E", if and only if its TDPS is in the normal form in "E". A DE is said to be in the normal form, if and only if it consists of only normal-form TDPSs.

The normal forms of DEs (2.4) and (2.5) are shown below.

[Normal form of DE (2.4)]

{ (f (x y) { {x,y} }), (g (x y) { {x,y} }) }

[Normal form of DE (2.5)]

{ (x EMPTY { {p,q} }), (y EMPTY { {p,q} }) }

The algorithm for normalizing DPSs are described in Sections 4.

3. Projected Function Method

3.1 General Partial Computation

The computation described in Section 2.1 requires that all actual parameters be <u>known</u> (or bound) even though it permits some actual parameters to remain unevaluated. Thus, it is called <u>total computation</u> [1]. In contrast, partial computation can proceed even though some parameters remain unknown *). Such unknown parameters can be bound after or during the partial computation.

A partial computation algorithm for functional programming languages (recursive program schemata) has been discussed by Ershov [3]. The term tabulation in his paper contains both the operations specific to partial computations and a general optimization technique known as tabulation [10]. In addition, when Ershov's approach is used, it becomes rather complicated discussing the limitations of a dataflow machine's computational power. Therefore, to avoid ambiguity and to clarify present discussion, the authors will present their own view on partial computation concepts in functional programming.

Partial computation consists of <u>partial applications</u>, <u>pre-binding applications</u> and <u>partial simplifications</u>.

(1) Partial application

Partial application stands for the application of a function to known parameters and the computation of a function that takes the value of the rest of the parameters (i.e. unknown parameters). This can be achieved by currying [11] known parameters from the function, and then applying these parameters.

For example, suppose that

```
                f  = ^[[x,y] (x+1)*x + y]
```

If "x" is known to be 2, then a partial application is possible. The result is as follows:

```
                fx = ^[[x] ^[[y] (x+1)*x + y ]].
                f2 = fx(2)
                   = ^[[x] ^[[y] (x+1)*x + y]](2)
                   = ^[[y] (2+1)*2 + y]
```

*) ---

Take care to distinguish between "unknown" and "undefined" variables. Unknown variables can be made known at any time by binding values to these variables. On the contrary, undefined is a special state of the known variables. Therefore, the undefined variables remain undefined throughout the entire computation.

```
                 = ^[[y] 6 + y]
```

The function "fx" is obtained by currying a known parameter "x" from "f". Then, an actual parameter value 2 is applied to "fx", and the result ^[[y] 6 + y] is computed.

(2) Pre-binding application

The result of partial applications is a function that takes only unknown parameters. Pre-binding applications are essentially the same as function applications. The difference is that all actual parameters are unknown in pre-binding applications.

For example, suppose that "f2" is the function defined above, and "u" is an unknown variable. Then,

```
            f2(u) = ^[[y] 6 + y](u)
                  = 6 + u
```

(3) Partial simplification

Partial simplification is the simplification of expressions containing unknown variables. Partial simplification has significance for non-strict primitive functions. As an example, suppose

```
        expr = (if x>0 then x else y fi) + x
```

and x is known to be 2. Then,

```
        expr = (if 2>0 then 2 else y fi) + 2
             = (if true then 2 else y fi) + 2
```

In this case, the non-strict function if-then-else-fi can be partially simplified. The result is

```
        expr = 2 + 2
             = 4
```

3.2 Tabulation Technique in Partial Computation

Tabulation (in context of total computation) is a well-known technique to improve the computational efficiency [10]. Tabulation means to keep track of function applications, and to store a return value with the function name and its actual parameters. When the same function application is encountered, the return value can be immediately obtained from the table instead of having to recompute the function application.

The approach presented in this paper adopts a currying operation, and generates functions that take only known parameters. Therefore, tabulation techniques for total computation [10,12] can be easily applied by storing the results of partial applications in tables or lists.

For example, suppose

```
   ack   = ^[[x,y] if x==0 then y+1 else
                      if y==0 then ack(x-1,1)
                         else ack(x-1,ack(x,y-1)) fi fi]
```

Then, "ack(0,y)" named "inc" can be computed as follows:

```
   ackx = ^[[x] ^[[y] if x==0 then y+1 else
                          if y==0 then ack(x-1,1)
                             else ack(x-1,ack(x,y-1)) fi fi]]
   inc   = ackx(0)
         = ^[[y] y+1 ]
```

Similarly, "ack(1,y)" named "add" can be computed as follows:

```
   add   = ackx(1)
         = ^[[y] if y==0 then ack(0,1)
                    else ack(0,ack(1,y-1)) fi]
         = ^[[y] if y==0 then ackx(0)(1)
                    else ackx(0)(ackx(1)(y-1)) fi]
         = ^[[y] if y==0 then inc(1)
                    else inc(add(y-1)) fi]
```

Projected Function Method

In the above example, the results of "ackx(0)" and "ackx(1)" are stored in the table (may be constructed using hashing), and the computation process of these results can be shared among other computation of the "ack" function.

General partial computation in functional programming is achieved by the method described in Section 3.1 in accordance with such tabulation techniques for total computation.

3.3 Projection of Functions

(1) Projection of a function

A projection of a function "f" by a parameter set "u" is a partial computation of "f", specifying "u" as unknown parameters and the rest of parameters as undefined.

For examples, suppose that a function "f" is defined as

```
f   = ^[[x,y,z] e(x,y,z)]
```

where "e(x,y,z)" is an arbitrary expression of "x", "y" and "z". Then, a projection of "f" by {x} named "fx" is defined as

```
fx  = ^[[y,z] ^[[x] e(x,y,z)]](w,w)
```

where "w" (please read it "omega" in this paper) stands for an undefined value.

Similarly, a projection of "f" by a parameter set {x,y} named "fxy" is defined as

```
fxy = ^[[z] ^[[x,y] e(x,y,z)]](w)
```

If function applications with undefined parameters appear in the body, these functions must also be projected by the parameter set excluding these undefined parameters. For example, suppose

```
f   = ^[[x,y] if x>0 then x else f(x+y,y) fi]        (3.1)
```

Then, a projection of "f" by "{x}" named "fx" is

```
fx = ^[[y] ^[[x] if x>0 then x else f(x+y,y) fi](w)
   = ^[[x] if x>0 then x else f(x+w,w) fi]
```

Projected Function Method

Since the primitive function "+" is strict, the function obtained by projecting "+" by the first parameter is a totally undefined function. Thus,

```
    fx = ^[[x] if x>0 then x else f(w,w) fi]
```

Then, a projection of "f" by {} named "fw" should be computed.

```
    fw = ^[[x,y] ^[[] if x>0 then x else f(x+y,y) fi](w,w)
       = ^[[] if w>0 then w else f(w+w,w) fi]
       = ^[[] W()]
```

where "W" stands for a totally undefined function. Therefore,

```
        fx = ^[[x] if x>0 then x else w fi]            (3.2)
```

(2) Projection of a DPS

Given a function "f" which has the DPS "s", assume that a function "fa" is obtained by projecting "f" by a parameter set "a". Then, the DPS of "fa" named "sa" can be computed from "s" and "a" as follows:

(a) If the DPS "s" is the null set, then, so is "sa".

(b) For the case where "s" has elements "ei" (i=1,..,n) (n>0), "sa" is the set of "ei" (i=1,..,n) which satisfies the condition

$$ei \subseteq a \qquad (i=1,..,n).$$

Since "sa" can be computed using only "s" and "a", "sa" is called a projection of "s" by "a".

For example, suppose "f" is the function defined in Expression (3.1). Then, the DPS of "f" named "s" is { {x},{x,y} }. The projection of "s" by {x} is { {x} }. This matches the DPS of "fx" defined in Expression (3.2). Similarly, the projection of "s" by "y" is the null set, indicating that the projection of "f" by "y" (named "fy") is a totally undefined function. This can be confirmed as follows:

```
   fy  = ^[[x] ^[[y] if x>0 then x else f(x+y,y) fi]](w)
```

projected Function Method

```
    = ^[[y] if w>0 then w else f(w+y,y) fi]
    = ^[[y] W()]
```

3.4 Restricted Class of Partial Computation

There exists a computation model which has a restricted partial computational power. For example, the DCSB model has the following limitations:

(1) Its computation is based on call-by-value.

(2) It cannot perform pre-binding applications.

(3) It has only restricted power on partial simplifications.

Limitation (1) can be overcome by introducing a lazy-evaluation mechanism [2] into the dataflow model [13]. Limitations (2) and (3), however, are more substantial, and are difficult to overcome. The origin of these limitations is in the dataflow model itself. In the dataflow model, computation is controled by tokens that carry data. Normally, such data are the resultant values of previous computation. In a lazy-evaluation context, data are either evaluated values or recipes that are to be evaluated [2]. In any event, tokens must reach a node to initiate computation in that node. Nevertheless, unknown values correspond to the state where a token has not yet reached a node. Therefore, computation is not possible for unknown values.

The limitations (2) and (3) become a serious problem when functions to be computed are non-strict. For example, suppose

```
f = ^[[x,y] if x > 0 then x else y fi];
e = f(x,u) + x
```

and "x" is known to be 2. Then,

```
e = f(2,u) + 2
  = ^[[x] ^[[y] if x > 0 then x else y fi]](2)(u) + 2
```

```
     = ^[[y] if 2 > 0 then 2 else y fi](u) + 2
```

If general partial simplification is possible, the above expression can be simplified as follows:

```
   e = ^[[y] 2](u) + 2
```

However, such simplification cannot be executed with the DCSB Model [6].

If pre-binding applications are allowed, the expression "e" can be further reduced as follows:

```
   e = 2 + 2
     = 4
```

As shown above, the limitations of the DCSB model significantly confine the partial computation process of programs.

In total computation, call-by-value is widely used, even though it has only a restricted power in contrast to call-by-name. This is because call-by-value exhibits superior execution speed and ease of implementation. Since pre-binding applications are more general in concept than call-by-name applications, it seems reasonable to consider the sub-class of partial computation which excludes pre-binding applications. The following sub-section will present a discussion of the partial computation method under this constraint.

3.5 Projected Function Method for Restricted Partial Computation

Given a function "f" which has the DPS "s", consider the case where "f" is partially computed by the parameter set "a". The <u>projected function method</u> is then defined as the following computation process:

(1) Compute the projection of "s" by "a", and name it "sa".

(2) If "sa" is null, then, go to Step (5).

(3) Compute a projection of "f" by "a", and name it "fa".

Projected Function Method

(4) Totally apply actual values of "a" to "fa". If a defined value is returned, the value is the result of the partial computation. If an undefined value is explicitly returned, then go to Step (5).

(5) If "a" contains at least one requisite parameters of "f", then go to Step (6). Else terminate partial computation of this function application.

(6) Partially apply actual values of "a" to "f",

(7) If the result of Step (6) contains function applications with unknown parameters, partially apply these functions using this method.

For example, consider the example discussed in Section 3.4.

```
f  = ^[[x,y] if x > 0 then x else y fi];
e  = f(x,u) + x
```

where "x" is known to be 2.

The DPS of "f" is { {x},{x,y} }, and the projection by {x} is { {x} }. Since it is not null, the expression is computed as follows:

Step (1): a projection of "f" by {x} named "fx" is computed.

```
fx = ^[[y] ^[[x] if x > 0 then x else y fi]] (w)
   = ^[[x] if x > 0 then x else w fi]
```

Step (2): "fx" is totally applied to a known parameter.

```
e = fx(2) + 2
  = ^[[x] if x > 0 then x else w fi](2) + 2
  = (if 2 > 0 then 2 else w fi) + 2
  = 2 + 2
  = 4
```

As shown above, the projected function method makes it possible to execute partial computation without pre-binding applications, at the sacrifice of computational complexity. This method can also be applied to non-strict primitives such as "if-then-else-fi".

The key idea of this method is the concept of the DPS. The next section is devoted for discussing DPSs.

4. Data Flow Analysis for DPSs

4.1 Overview of Data Flow Analysis for DPSs

Figure 4-1 shows the outline of our approach to data flow analysis for obtaining DPSs. The goal of the data flow analysis is to obtain a normal form of DPSs for all functions, which consists of the following two sub-goals. The first is to transform each function into an initial DPS which is obtained through the data flow analysis only in the function body. The second is to obtain the normal form of DPSs by reducing the results of the first sub-goal with the DE where function definitions are placed. The second sub-goal is achieved through a stepwise reduction of DPSs, called the DPS normalization procedure.

$$\begin{tikzcd}
\text{Source Program} \arrow[d] & \\
\text{DPS} \arrow[r, "\text{Flattening}"] \arrow[ddd, "\text{Normalization}"'] & \text{DPS\$} \arrow[d, "\text{DPS Reduction}"] \\
& \text{DPS\$} \arrow[d, "\text{DPS Reduction}"] \\
& \vdots \arrow[d, "\text{DPS Reduction}"] \\
\text{Normal Form of DPS} & \text{Converged DPS\$} \arrow[l, "\text{\$-Elimination}"]
\end{tikzcd}$$

Fig. 4-1 Outline of Data Flow Analysis for DPS

In the DPS normalization procedure, three data structures, namely, DPS$, TDPS$ and DE$ are introduced corresponding to DPS, TDPS and DE defined in Section 2.4, respectively. DPS$, which is called a flat form of DPS, is the same as DPS except that all function application forms in DPS are replaced with the symbol "$". After the introduction of DPS$, it is easy to develop TDPS$ and DE$ from the definitions of TDPS and DE, i.e. TDPS$ and DE$ are the same as TDPS and DE, respectively, except that DPS in TDPS is replaced with DPS$, and TDPS in DE is replaced with TDPS$. Using the above three data structures, the DPS normalization procedure can be described as follows.

First, the normal form of each DPS is assumed to be DPS$, and the normal form of DE is also assumed to be DE$. Next, all DPS$ in DE$ are modified with an operation called <u>DPS</u> <u>reduction</u> which replaces each parameter in DPS with DPS$ if TDPS$ of the parameter is in DE$. The DPS reductions for all DPS$ provide new DPS$ and DE$

which become the refined assumptions for normal forms of DPS and DE. Then, all DPS$ are again modified through DPS reduction using DPS$ and DE$ obtained by the preceding DPS reduction. Such a modification continues until all DPS$ converge, i.e. all DPS$ do not change by the DPS reduction. Finaly, normal forms of DPSs are obtained by an operation called <u>$-elimination</u> which removes Minimally Sufficient Parameter Sets (MSPSs) containing the symbol "$" from DPS$.

The algorithm of the data flow analysis for obtaining DPSs is described from the bottom up in the following sub-sections in detail, i.e. the DPS reduction, the DPS normalization procedure and the algorithm for transforming a source program to a normal DPS form are described in Section 4.2, 4.3 and 4.4, respectively. Furthermore, examples of the DPS computation described in Section 5 will facilitate an understanding of the algorithm.

4.2 Primitive Operators for the DPS Reduction

In this sub-section, some primitive operators for DPSs are introduced to make it possible to define an algorithm for obtaining DPSs in the following sub-sections.

(1) Primitive set operators

Operators + and * for DPSs provide a union and a Cartesian product for two DPSs, respectively. For example,

$$\{\{x,y\},\{x\}\} + \{\{x,z\},\{z\},\{x\}\} = \{\{x,y\},\{x\},\{x,z\},\{z\}\}$$

and $$\{\{x,y\},\{x\}\} * \{\{x,z\},\{z\},\{x\}\} = \{\{x,y,z\},\{x,y\},\{x,z\},\{x\}\}.$$

(2) Composite set operators

To facilitate an understanding a union and a Cartesian product in a set of DPSs, operators $\sum$ and $\prod$ are defined as follows.

$$\sum_{(\text{for } i=1 \text{ to } n)} \text{DPSi} = \text{DPS1} + \sum_{(\text{for } j=2 \text{ to } n)} \text{DPSj} \quad (\text{for } n > 1)$$
$$= \text{DPS1} \quad (\text{for } n = 1)$$

$$\prod_{(\text{for } i=1 \text{ to } n)} \text{DPSi} = \text{DPS1} * \prod_{(\text{for } j=2 \text{ to } n)} \text{DPSj} \quad (\text{for } n > 1)$$
$$= \text{DPS1} \quad (\text{for } n = 1)$$

(3) DPS reduction operators

In the DPS normalization procedure, DPSs are iteratively reduced by DPS reduction, i.e. parameters in each DPS are replaced with the DPS$ of the parameters in DE$. Since a DPS of a function "f" is a set which contains all MSPS of "f", the result of DPS reduction is equal to the union of the results which are obtained by replacing parameters in each MSPS of the DPS with DPS$ of the parameters in DE$. Such a replacement is called <u>MSPS reduction</u>. Therefore, using an operator for MSPS reduction, named "reduce-m", an operator for DPS reduction , named "reduce", can be defined as follows:

$$\text{reduce(DPS, DE\$)} = \sum_{(\text{for all MSPS in DPS})} \text{reduce-m(MSPS, DE\$)}$$

On the other hand, since a MSPS of "f" is a set of minimal parameters with which "f" returns a defined value, the result of MSPS reduction is a Cartesian product of the results which are obtained by replacing each parameter in the MSPS with DPS$ of the parameter in DE$. Such a replacement is called <u>parameter reduction</u>. Consequently, using an operator for parameter reduction, named "reduce-p", "reduce-m" can be defined as follows.

$$\text{reduce-m(MSPS, DE\$)} = \prod_{(\text{for all parameter in MSPS})} \text{reduce-p(parameter, DE\$)}$$

When a parameter in MSPS is "P", the operation reduce-p(P,DE$)

is informally defined according to the attribute of "P" as follows.

```
reduce-p(P, DE$)
       = {case
          {P is a formal parameter} -> {{P}};

          {P is a variable name other than a formal parameter
           and there exists no TDPS$ of P in DE$} -> {{P}};

          {P is a variable name other than a formal parameter
           but there exists a TDPS$ of P, refered to as "T",
            in DE$}     -> { return DPS$ in "T" };

          {P is function-application-form, which refers to
           function "f" and there exists no TDPS$ of "f"
           in DE$}      -> {{P}};

          {P is function-application-form with a function
           name "f" and parameter-DPS-list (d1 d2 ... dn).
           In addition, there exists a TDPS$ of "f",
           referred to as "T", in DE$,
           where  T = (f (x1 x2 ... xn) D$)}
               ->
               {begin
                 {A list of DPS$ for actual parameters,
                  named "A$", is obtained by the
                  following operations.
                  A$=(a1 a2 ... an) and
                  ai=reduce(di,DE$) for i=1 to n}
                 {All formal parameters in DPS$ of "T" are
                  replaced with "A$", i.e. each xi in D$
                  is replaced with ai for all i = 1 to n}
                end}
          end}
```

4.3 Algorithm for the DPS Normalization

It is necessary to replace variable names, which are defined in a block expression, with their DPSs in order to reduce a DPS to its normal form in the block. An algorithm for such a replacement is trivial if no DPS in a block expression contains function-application-forms. However, it requires rather complicated data flow analysis, if function names are mutually referenced in two function definitions in a block expression or are cyclically referred in some function definitions in a block expression. In this sub-section, an iterative procedure for reducing DPSs to normal forms in block expressions is provided.

An operation N(ENV) transforms a DE "ENV" into a new DE "NENV" which is in a normal form, i.e. all DPSs in "NENV" contain only formal parameters and free variables in "ENV". Therefore, if "ENV" contains DPSs for all functions defined in a scope, "NENV" contains all of the DPSs which are transformed into normal forms in the scope. The operator N is an iterative procedure which consists of four phases as shown in Fig. 4-2.

```
                DE
                |
                V
     PHASE 1 : Flattening
                |<-----------------
                V                 |
     PHASE 2 : DPS Reduction      |
                |                 | Not Converge.
                V                 |
     PHASE 3 : Convergence Check --
                |
                | All DPS$ converge.
                V
     PHASE 4 : $-Elimination
                |
                V
        Normalized DE
```

Fig. 4-2 DPS Normalization Procedure

Functions of the four phases in the procedure are described as follows.

[Algorithm of DPS Normalization Operator N]

PHASE 1 -- Initialize Iteration : Flattening --

All DPSs in the input DE "ENV" are transformed into DPS$s, called the flat form, i.e. all function-application-forms in each DPS are replaced with the symbol "$". In addition, TDPS$ and DE$ "ENV$" are constructed using DPS$. Then, PHASE 2 is activated.

PHASE 2 -- DPS Reduction --

Each DPS in "ENV" is reduced using "ENV$", resulting in a TDPS$. The TDPS$s for all DPSs constitute a new DE$ named "NENV$". The reduction proceeds as follows:

(1) A TDPS "t" is chosen from "ENV" for the DPS reduction, where the name of "t" and the DPS in "t" are referred to as "n" and "d", respectively.

(2) A new DPS$ "nd$" for "n" is obtained by DPS reduction, i.e. nd$=reduce(d,ENV$).

(3) The TDPS$ with the name "n" is chosen from "ENV$" and is referred to as "t$". A DPS$ of "t$", named "d$", is also chosen.

(4) "t$" is set to be in the "CONVERGE" status if nd$=d$. Otherwise, it is set to be in the "TEMPORARY" status.

When all DPSs in "ENV" are reduced, PHASE 3 is activated.

PHASE 3 -- Check End of Iteration : Convergence Check --

If all TDPS$ in "NEVN$" converge, i.e. if no TDPS$ in "NENV$" is in "TEMPORARY" status, the iteration is terminated and PHASE 4 is activated. Otherwise, PHASE 2 is activated to further reduce DPSs, using "NENV$" as a new "ENV$".

PHASE 4 -- $-Elimination --

An MSPS in each TDPS$ in "NENV$" is eliminated if the MSPS contains the symbol "$". The results of the elimination constitute a normal form of "ENV".

4.4 Algorithm for Obtaining Normalized DPSs

In this sub-section, an algorithm for obtaining normalized DPSs is described by means of set-operators for DPSs and a DPS normalization operator introduced in the previous sub-sections. According to an attribute of an input expression defined in Valid syntax [8], an operator, named D, transforms the expression in a program into the normal form of the expression's DPS. The algorithm of the operator D is informally defined as follows.

```
D(e) =
  { case
   free-variable-name -> {{e}};

   local-defined-variable-name -> D(value-definition of e);

   strict-primitive-function-application ->  $\prod$    D(ei);
                                        (for all operand
                                      expressions ei in e)

   if-then-else-fi ->
          D(predicate-expression) * D(then-part-expression)
        + D(predicate-expression) * D(else-part-expression);

   value-definition -> D(defined-expression);

   function-definition -> D(defined-function-body);

   non-primitive-function-application
        -> {{(function-name D(arg1) D(arg2) ... D(arg-n))}};

   block-expression ->
        {begin
        { Transform the return expression of this block-expression
        to the value definition with a unique name.
        Suppose the name is "&r".
        { As the results that the operator D is applied to
        each variable definition in the block, a DE "ENV"
        in the block expression is created. }
        { The "ENV" is transformed into its normal form "NENV"
         by NENV=N(ENV). }
        { A DPS in the TDPS with the value-name "&r" is returned. }
        end}
   end}
```

The algorithm for obtaining normalized DPSs described in this section has been implemented using TAO-Lisp [14] under VAX/VMS. The program consists of about 600 lines, and the execution results of the program for several examples are shown in the next section.

5. Examples of the DPS Computation

This section provides examples of DPS computation using the algorithm presented in Section 4. Examples are ordered in complexity.

5.1 Simple Expressions

The first example is a block expression without function definitions.

[Program 1]

```
        { a=x*y; b=x+z; x=g*h; y=x*g; z=r+s; return a }
Processed Block ::
        { a=x*y; b=x+z; x=g*h; y=x*g; z=r+s; return a }
   Dependency Property Set of the Block ::
        { {g,h} }
```

Thus, the result of this block expression is defined by the free variables "g" and "h", although "r" and "s" are also free.

5.2 Function Application

[Program 2]

```
     f=^[[k,u] if u==0 then 0 else f(k+1,u-1) fi]
     Notice that this program is equivalent to
     f=^[[k,u] if u≥0 then 0 else w fi] .
Processed Block ::
        if u==0 then 0 else f(k+1,u-1) fi
   Dependency Property Set of the Block ::
        { {u}, {u,(f {{k}} {{u}})} }
Initial DPSs of functions in the surrounding block
   FUNC f(k,u) : Initial DPS = { {u}, {u,(f {{k}} {{u}})} }
<< Iterative Procedure >>
```

```
STEP : 0   TEMPORARY status : {f}

                FUNC f(k,u) : DPS = { {u} }

STEP : 1   CONVERGE  status : {f}

                FUNC f(k,u) : DPS = { {u} }

        --- End of Iteration ---
```

Dependency Property Set of the Block ::

```
        FUNC f(k,u) : DPS = { {u} }
```

Therefore, the result of "f" in Program 2 is determined by the second parameter only.

5.3 Mutual Recursion

[Program 3]

```
    { f=^[[x,y] if x>0 then f(x-1,y+1) else g(x,y) fi];
      g=^[[x,y] if x==0 then y else f(-x,-y) fi] }
```

This program is equivalent to f=^[[x,y]x+y].

Processed Block ::

```
        if x>0 then f(x-1,y+1) else g(x,y) fi
```

Dependency Property Set of the Block ::

```
        { {x,(f {{x}} {{y}})}, {x,(g {{x}} {{y}})} }
```

Processed Block ::

```
        if x==0 then y else f(-x,-y) fi
```

Dependency Property Set of the Block ::

```
        { {x,y}, {x,(f {{x}} {{y}})} }
```

Initial DPSs of functions in the surrounding block

```
   FUNC f(x,y) : Initial DPS = { {x,(f {{x}} {{y}})},
                                 {x,(g {{x}} {{y}})} }
   FUNC g(x,y) : Initial DPS = { {x,y}, {x,(f {{x}} {{y}})} }
```

Initial DPSs are assumed to be $.

<< Iterative Procedure >>

```
STEP : 0   TEMPORARY status : {f,g}

                FUNC f(x,y) : DPS = { {$,x} }
```

Examples of the DPS Computation

```
                  FUNC g(x,y) : DPS = { {$,x}, {x,y} }
STEP : 1    TEMPORARY status : {f}
                  FUNC f(x,y) : DPS = { {$,x}, {x,y} }
            CONVERGE  status : {g}
                  FUNC g(x,y) : DPS = { {$,x}, {x,y} }
STEP : 2    CONVERGE  status : {f,g}
                  FUNC f(x,y) : DPS = { {$,x}, {x,y} }
                  FUNC g(x,y) : DPS = { {$,x}, {x,y} }
          --- End of Iteration ---
Dependency Property Set of the Block ::
          FUNC f(x,y) : DPS = { {x,y} }
          FUNC g(x,y) : DPS = { {x,y} }
```

Program 3 is an example in which the conventional maximal fixed point solution approach [7] yields an incorrect solution. The computation based on this approach is shown.

Initial DPSs are assumed to be null.

```
<< Iterative Procedure >>
STEP : 0    TEMPORARY status : {f,g}
                  FUNC f(x,y) : DPS = { {x} }
                  FUNC g(x,y) : DPS = { {x}, {x,y} }
STEP : 1    TEMPORARY status : {f}
                  FUNC f(x,y) : DPS = { {x}, {x,y} }
            CONVERGE  status : {g}
                  FUNC g(x,y) : DPS = { {x}, {x,y} }
STEP : 2    CONVERGE  status : {f,g}
                  FUNC f(x,y) : DPS = { {x}, {x,y} }
                  FUNC g(x,y) : DPS = { {x}, {x,y} }
          --- End of Iteration ---
Dependency Property Set of the Block ::
```

Examples of the DPS Computation

```
        FUNC f(x,y) : DPS = { {x}, {x,y} }

        FUNC g(x,y) : DPS = { {x}, {x,y} }
```

Since "x+y" cannot be evaluated without "y", it is incorrect to include a parameter set {x} in the DPS of "f".

5.4 Nested Function Applications

```
[Program 4]
        { f = ^[[k,u] if u==0 then 0 else f(k,f(u,k-1)) fi }
Processed Block ::
        if u==0 then 0 else f(k,f(u,k-1)) fi
   Dependency Property Set of the Block ::
        { {u}, {u,(f {{k}} {{ (f {{u}} {{k}}) }})} }
Initial DPSs of functions in the surrounding block
   FUNC f(k,u) : Initial DPS
               = { {u}, {u,(f {{k}} {{ (f {{u}} {{k}}) }})} }
<< Iterative Procedure >>
STEP : 0   TEMPORARY status : {f}
                FUNC f(k,u) : DPS = { {u} }
STEP : 1   TEMPORARY status : {f}
                FUNC f(k,u) : DPS = { {u}, {k,u} }
STEP : 2   CONVERGE  status : {f}
                FUNC f(k,u) : DPS = { {u}, {k,u} }
        --- End of Iteration ---
Dependency Property Set of the Block ::
        FUNC f(k,u) : DPS = { {u}, {k,u} }
```

5.5 Nested Block Expressions

```
[Program 5]
   { f = ^[[x,y]
        { g = ^[[a,b] a+a]; k = g(x,y)+x; return k+x } ] }
```

```
Examples of the DPS Computation

Processed Block ::
         { g = ^[[a,b] a+a]; k = g(x,y)+x; return k+x }

Processed Block ::
        a+a
   Dependency Property Set of the Block ::
        { {a} }
Initial DPSs of functions in the surrounding block
   FUNC g(a,b) : Initial DPS = { {a} }
<< Iterative Procedure >>
STEP : 0   TEMPORARY status : {g}
                FUNC g(a,b) : DPS = { {a} }
STEP : 1   CONVERGE  status : {g}
                FUNC g(a,b) : DPS = { {a} }
        --- End of Iteration ---
Dependency Property Set of the Block ::
        FUNC g(a,b) : DPS = { {a} }

   Dependency Property Set of the Block ::
        { {x} }
Initial DPSs of functions in the surrounding block
   FUNC f(x,y) : Initial DPS = { {x} }
<< Iterative Procedure >>
STEP : 0   TEMPORARY status : {f,g}
                FUNC f(x,y) : DPS = { {x} }
                FUNC g(a,b) : DPS = { {a} }
STEP : 1   CONVERGE  status : {g}
                FUNC f(x,y) : DPS = { {x} }
                FUNC g(a,b) : DPS = { {a} }
        --- End of Iteration ---
Dependency Property Set of the Block ::
        FUNC f(x,y) : DPS = { {x} }
        FUNC g(a,b) : DPS = { {a} }
```

As shown above, DPSs of nested block expressions are evaluated from the inner to the outer one.

6. Conclusion

This paper has proposed a new partial computation method for functional programming languages, called the projected function method. This method makes it possible to execute general partial computation without the pre-binding capability that is essential to the partial computation of non-strict functions in the conventional method.

This paper has also presented a new concept called Dependency Property Set (DPS). The DPS indicates the dependency relation between function parameters and resultant values. The DPS concept plays an important role in the projected function method. An algorithm for computing DPSs based on the data flow analysis method is also shown.

In spite of its clearness, the most serious problem in functional programming has been computational inefficiency compared with side-effect based programming. Since partial computation enables intensive program optimization and computation sharing in fully automatic way, significant computation reduction is possible. The computational power required for the projected function method is same as the computational power of the DCSB model. Therefore, this method in conjunction with the DCSB model offers a way to realize highly-parallel and highly-effective functional programming machines.

Acknowledgements

The authors would like to thank Dr. Noriyoshi Kuroyanagi, Director of the Communication Principles Research Division at Musashino Electrical Laboratory for his constant guidance and encouragement. They also wish to thank the members of the dataflow system research group in the Communication Principles Research Section 2 for their thoughtful discussions and comments.

References

1. Futamura,Y. Partial Computation of Programs, Proc. 4th RIMS Symposia on Software Science and Engineering, Lecture Notes in Computer Science No.147, Springer-Verlag, (1983), 1-35
2. Henderson,P. Functional Programming / Application and Implementation, Prentice-Hall, (1980)
3. Ershov,A.P. Mixed Computation in the Class of Recursive Program Schemata, Acta Cyberneticca, Tom.4, Fosc.1, Szeged, (1978), 19-23
4. Beckman,L., et al. A Partial Evaluator and its Use as a Programming Tool, Artificial Intelligence 7, (1976), 319-357

References

5. Kahn,K. A partial evaluator of Lisp written in a Prolog written in Lisp intended to be applied to the Prolog and itself which in turn is intended to be given to itself together with the Prolog to produce a Prolog compiler, UPMAIL, Dept. Computer Science, Uppsala Univ., (1982)

6. Ono,S., Takahashi,N. and Amamiya,M. Partial Computation with a Dataflow Machine, Proc. 5th RIMS Symposia on Mathematical Methods in Software Science and Engineering, RIMS Kyoto Univ. (1984), 169-203

7. Kam,J.B. and Ullman,J.D. Global Data Flow Analysis and Iterative Algorithms, JACM 23, (1978), 158-171

8. Amamiya,M., Hasegawa,R. and Ono,S. Valid, A High-Level Functional Programming Language for Data Flow Machines, Rev. ECL 32, NTT, (To appear)

9. Manna,Z. Mathematical Theory of Computation, McGRAW-HILL, (1974)

10. Bird,R.S. Tabulation Techniques for Recursive Programs, ACM Computing Surveys 12, (1980), 403-417

11. Turner,D.A. A New Implementation Technique for Applicative Language, Software - Practice and Experience 9, (1979), 31-49

12. Keller,R.M. and Sleep,M.R. Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages, Proc. Conference on functional programming and computer architecture, ACM, (1981), 131-140

13. Amamiya,M. and Hasegawa,R. Dataflow Computing and Eager and Lazy Evaluations, New Generation Computing 2, (1984), 105-129

14. Umemura,K. TAO-Lisp: Portable Lisp System written in Pascal, Proc. 27th Annual Conference of IPSJ, (in Japanese), (1983), 409-410