UDC 51:681.3

Ján Kollár

THE CONCEPTION AND APPLICATION OF *PFL* : A PROCESS FUNCTIONAL PROGRAMMING LANGUAGE

A new process functional programming paradigm and its application in PFL — a process functional programming language is introduced in the paper. This paradigm is based on affecting the state represented by the values of memory cells strictly by evaluating expressions. Process functional conception prevents the use of assignments, which is a good basis for reasoning about the programs and systems by direct PFL program profiling and transformation. This paper is oriented to essential constructs of PFL language and its relation to object and parallel programming paradigm.

Introduction

It is well known that more complex tasks can be solved just increasing the level of the abstraction. Programming the complex and high parallel systems belongs to this category since the requirements for the correct functionality as well as for the sufficient time responses are of high importance [2].

The declarative nature of a purely functional approach may help to reason about the correctness and run-time performance [28]. Unfortunately, a purely functional specification deals with the referentially transparent expressions that are not powerful enough to express the state, which is crucial in the systems [17]. Therefore the application of purely functional languages to programs is elegant and conforming with mathematical methods of programming, but their application to the systems is cumbersome.

Modelling and analysis of the systems are essentially based on Petri nets and their derivatives [5, 6]. These methods are built up on the mathematical basis of Petri net automata and they are applied at different level of the modelled and analysed systems. However, from the viewpoint of software engineering and the specification of the systems, the language of Petri nets still remains on the assembler level. Therefore, it seems that the modelling and analysis of the systems on the basis of Petri nets would have be integrated into a high-level programming language, the best purely functional, taking the advantages of both approaches.

Unfortunately, Petri nets deal with the stateful systems [27] while pure functional languages do not consider the state at all. Seemingly Petri nets conform well just with imperative languages. On the other hand using mutable abstract types and monads [4, 31] it is possible to manipulate the state in a purely functional manner. These approaches can be found in Haskell [26] or Clean [1]. The essential idea is to update the memory cells by the application of the functions to the arguments of mutable abstract types. Hence, the mapping from values to computation allows consider a memory cell like an abstract type that changes its definition during the run time. Compare with SML [3], the abstraction level has increased rapidly, since there are no assignments used in Haskell or Clean programs and the memory cells are hidden to a programmer.

On the other hand, the work with just values and computations has two disadvantages. First, the role of the control in computation is suppressed, since it is not manipulated explicitly like in an imperative language [9, 16]. Second, and perhaps yet more important is, that memory cells are hidden to a programmer. Using a pure functional language, all the activity is well defined not however the subjects of this activity. The situation is almost the same like repairing the car knowing just how to do it, but omitting what and where are the component parts. Of course, one may argue that it is better to manipulate them by some automation tool. However, the programming is still the game that is the matter of the people, not robots. A programming language would not have to restrict the human creativity, excluding time and the spatial character of real world.

The process functional approach [11—15] is somewhere in the middle of imperative and functional approaches to software construction. Omitting the assignments, separating the concerns of the control and the data, considering the 'natural' environment of current von Neumann computers we are streaming about transparent and simple concept of systems specification.

In this paper we present the essential conception of PFL — an experimental process functional language and the opportunities that it provides to a programmer. From the viewpoint of software engineering, using PFL, neither rigorous mathematical approach, nor ad-hock combinatorial approach is preferred. A programmer may use any of them. The PFL script will be still in a mathematical form that allows, as we hope the reasoning about the correct functionality [23, 24, 25] and the run time characteristics via profiling support based on the Petri nets [7, 8].

1. PFL Types and Values

The values in PFL — a process functional programming language are defined in terms of concrete types of three kinds:

- **Data types** are domains of the data values, including the functions and the processes
- **Unit type** consists of just one control value, and
- **Spatial types** define immaterial positions in the space.

1.1. Data Types. The types char, int, and float, are primitive types, and they correspond to characters, integer and floating point numbers, respectively. For example, the value 'A' is of the type char, the value 320 is of the type int and 2.45 is of the type float. Primitive types are built-in types in PFL.

Algebraic types are new types defined by a programmer. The concept of algebraic data types is the same like in Haskell or Gofer [10], except that PFL types are designated with identifiers starting by lowercase letters. Algebraic data types are powerful enough to define enumerated types, variant types, the structures of items of the different types (similar to Pascal records) and the recursive types, such as lists and trees. The weakness of the algebraic types of purely functional languages is that they are not powerful enough to specify the efficiently implemented arrays. The algebraic types are defined using PFL data definitions.

Since PFL functions and processes are higher order, they may be included in data types category, although they are defined using PFL type definitions.

1.2. Unit Type. The unit type () consists of just one control value (), designated in the same way like the type. The control value has quite different meaning in computation like the data value. It means that control value does not belong to any data type. The role of the control is as much important as the role of data, hence the control is expressed and manipulated in PFL like a value, which has no significant representation but it has significant meaning.

Notice the analogous approach for the undefined value, which however belongs to each data type. The undefined value, usually designated by symbol $\perp$ has no significant representation, and it is still well-defined value. For example, $(1/0) = \perp$, or $(\text{False} \wedge \perp) = \text{False}$.

The control value is an argument or the value of PFL processes.

1.3. Spatial Types. The spatial type $\{R_1, ..., R_d\}$, where R_k is an enumerated algebraic type or a sub-range of integers, defines the d-dimensional space of immaterial positions — the spatial values. For example, the values of the spatial type {1..3, 0..1} are the spatial values {1, 0}, {1, 1}, {2, 0}, {2, 1}, {3, 0}, and {3, 1}. A spatial type comprises an ordered set of spatial values, which is mapped to a subset of cardinal numbers, starting with 0. Therefore each spatial value may be expressed in terms of a data value.

However, from the viewpoint of specification, it is better to separate the concerns of immaterial positions and material data values. Spatial types in PFL are proposed for an efficient implementation of arrays via spatial processes that are referentially non-transparent and extensionally defined mappings from spatial types to data types. The implementation of arrays in PFL in this way is as efficient like in any imperative language, but with considerably increased abstraction level.

1.4. Abstract Types. Opposite to concrete types that describe values, PFL abstract types describe overloaded functions and/or processes. User-defined abstract types are defined using type classes and their instances. Built-in abstract types are mutable abstract types allowing the update and the access of memory cells, instead of assignments.

2. PFL Data Definitions

Algebraic types are new types defined by a programmer using PFL **data** definition. Like examples of algebraic types, let us define the monomorphic type bool of boolean values False and True and three polymorphic types, tuple2, list and btree, as follows:

```
data bool      = False | True
data tuple2 a b = Tuple2 a b
data list a    = Nil | Cons a (list a)
data btree a   = Tip a
               | Bin (btree a) (btree a)
```

The types are defined by the sum of applications of constructors to type expressions. In our example above, the constructors are False, True, Tuple2, Nil, Cons, Tip, and Bin. If a type is defined by single constructor, (such as it is in the type tuple2), then it is called a product type, otherwise it is called a sum type. The constructors are applied in expressions to construct the values of algebraic types, since they are canonical functions, of the types as follows:

```
False  :: bool
True   :: bool
Tuple2 a b   :: tuple2 a b
Nil    :: list a
Cons   :: a -> list a -> list a

Tip    :: a -> btree a
Bin    :: btree a -> btree a -> btree a
```

The type definitions of constructors are never introduced in PFL script, since they are derived in the compile time from the **data** definitions.

The value (Tuple2 'a' 4.5) is of the type (tuple2 char float). The value (Tuple2 5 50) is of the type (tuple int int). The polymorphic type (tuple2 a b) is the principal (the most general) type of all more specific types of pairs, such as the types (tuple2 char float) and (tuple2 int int). Since the types tuple2, list, and btree have type variables such as a and b, they are polymorphic types [18]. On the other hand, bool is the monomorphic type.

The more appropriate syntactic forms for tuples and lists are available; The type (tuple2 a b) of pairs is written in the form (a, b), the type of triples (tuple3 a b c) in the form (a, b, c), etc. The type (list a) of lists is written in the form [a]. The more appropriate form for tuple (Tuple2 'a' 4.5) is ('a', 4.5). The constructor Nil is written as [ ]. Instead of Cons, the infix operator (:) is used. Hence, instead of the list of characters (Cons 'A' (Cons 'L' (Cons 'F' Nil))) we can write ('A' : 'L' : 'F' :[ ]). Yet more concise form for this list is as follows ['A', 'L', 'F'], or even "ALF", since the string is a list of characters.

PFL data definitions are allowed just at the global level of PFL script.

3. PFL Type Definitions

Both functions and processes in PFL may be built-in or user defined. In general, a type of function or process in PFL is defined by the function or process type definition in the form:

f :: t_1 -> ... -> t_n -> t

Depending on the form of types of arguments $t_1, ..., t_n$, and the type t of value, the f is either a function or a process.

The type definitions for built-in functions and processes (the operations) are obligatory, as well as the type definitions for user-defined processes. The type definitions for user defined functions are optional.

3.1. Function Types. If all the types of arguments as well as the type of value are data types or function types, then the type definition defines the type of a function, for example:

```
sum :: int -> int -> int
map :: (a -> b) -> [a] -> [b]
item :: ({1..50} -> a) -> int -> a
```

The arguments and value of the function sum are of the type int. The first argument of the function map of the type (a -> b) is a single argument function, the second is a list of the type [a] and the value is a list of the type [b]. The first argument of the function item is of the type ({1..50} -> a) of the spatial process — a mapping from space of 50 points of array positions to values of any type. The second argument is of the type int. The value of a function item is of any type, but the same like the type of items of the array.

The unit type () cannot be the type of arguments and/or the value of a function.

Notice also that the spatial type itself is never used like the type of an argument or the value of user defined functions and processes, since they never work with immaterial space positions. For example, the types ({1..50} -> a), ({1..50} -> -> float), ({1..50} -> [a]), ({1..50} -> (a->b)), are possible types for arguments or the value, not however the type {1..50}.

3.2. Process Types. The processes differ from functions just by their type definitions, not however by their definitions. The type definition is the definition of a process, not a function for two reasons:

1. The type definition of a process comprises the unit type () for an argument and/or the value.

2. The type definition comprises an argument or the value of the type in the form V t, where V is an environment variable and t is any data type.

The type in the form V () cannot occur in a type definition of a process. Intuitively, an environment variable cannot hold the control value.

Let us introduce some examples of process type definitions.

```
seq2 :: () -> () -> ()
par2 :: () -> () -> ()
```

Both processes above have the arguments and value of unit type. Therefore the processes seq2 and par2 (we omit their definitions for a while) can be applied to two control values (or expressions producing the control values) producing the control value.

The second example illustrates mixing the types in the type definition.

```
mixed :: () -> a -> ({1..40} -> a) -> ()
```

The first argument of process mixed is the control value, the second is a value of an expression of any type and the third is a spatial process of the type ({1..40} -> a). The value of the process mixed is the control value.

Preceding an argument and/or the value type of a function by an environment variable, designated by an identifier starting with uppercase letter, such a mapping is a process. In PFL the environment variables are not declared separately, just specified in the type definitions.

Let us introduce the next type definitions of the process p and the process q.

```
p :: A int -> B ({1..10, 1..10} -> (a -> b)) ->
     b -> C [a] -> float
q :: A a -> b -> A b
```

The environment variable A is shared by the first argument of process p, by the first argument of process q and by the value of process q. On the other hand, B and C are quite different variables (which however may be shared by another processes). The environment variables never occur in process definitions.

Provided that A t is an argument or value type, the value residing in A is of the type t.

Since the types int, a, and b can be unified producing the type int, the type definitions are correct, since they are specialised to:

```
p :: A int -> B ({1..10, 1..10} -> (a -> b)) ->
     b -> C [a] -> float
q :: A int -> int -> A int
```

On the other hand the type definitions, as follows:

```
p :: A int -> B a -> C a
q :: A char -> A b
```

are not correct, since char does not unifies with int.

In addition, in contrast to purely functional languages, we allow type definitions for both local functions and local processes.

4. PFL Type Synonyms

Each type expression not comprising an environment variable in PFL can be substituted by a type synonym, which is defined using **type** synonym definition. For example, let us have the type definition, as follows:

```
mixed :: () -> a -> ({1..40} -> a) -> ()
```

Let us define a new name for the type expression () -> a -> ({1..40} -> a) -> (), as follows:

```
type mixedtype a = () -> a ->
                   ({1..40} -> a) -> ()
```

Then the type definition above may be written in the form as follows:

```
mixed :: mixedtype a
```

We may define

```
type array a = {1..40} -> a
```

and to write

```
mixed :: () -> a -> array a -> ()
```

Using the **type** synonym in contrast to **data** definition, no new type is defined, just a new name for yet defined type is introduced to make the script more transparent. Hence, the type mixedtype a as well as the type () -> a -> -> array a -> () are the same types like the type () -> a -> ({1..40} -> a) -> ().

It is good praxis to predefine some algebraic types and type synonyms in the prelude file, which is prefixed to each user script, if compiled. Such the type synonym is string, defined as follows:

```
type string = [char]
```

Clearly, strings are lists of characters.

PFL type synonyms are allowed just on global level and they cannot be defined for the type expressions comprising the environment variables.

5. PFL Definitions

The form of the definitions in PFL is purely functional, like in Haskell or Gofer. It is impossible to distinct functions from processes looking just on the definitions. The information about their types is necessary. For example, the definition

```
f x y = x+y
```

is the function f, provided that the type definition omits, or it comprises neither environment variable, nor unit type (). For example, such a type definition could be as follows:

```
f :: int -> int -> int
```

It is good praxis to precede the definition by the type definition, hence the next two equations

```
f :: int -> int -> int
f x y = x + y
```

are the type definition and the definition of the function f, which summarises the values of its arguments. For example, the value of expression (f 3 2) is 5.

On the other hand, the equations, as follows:

```
f :: Alpha int -> Alpha int -> Alpha int
f x y = x+y
```

define the process f. The value of expression (f 3 2) is again 5, provided that the program is evaluated sequentially, i.e. in a single-threaded manner. Except that, the additional side effects on the variable Alpha are performed. First, applying (f 3), the value 3 is assigned to Alpha. Then, applying ((f 3) 2) the value 2 is assigned to Alpha, and finally, evaluating 3 + 2, the value 5 is assigned to Alpha. Hence, evaluating an expression, the state has been changed. Moreover, it is possible to apply the process f to control values, i.e. the applications (f () 2), (f 3 ()), and (f () ()) are legal applications of the process f (not however of the function f) in PFL. We discuss the semantics of such applications below, here we will concentrate to the forms of the function and process definitions.

5.1. Simple Definition. A function and/or process is defined by a simple expression on right hand side of the equation, for example, as follows:

```
h x y = x + y
```

5.2. Pattern Matching Definition. A function is defined by one or more equations. Then the constant patterns on the left-hand side of each equation are compared with the argument values. In case of matching, the corresponding right hand side is selected for evaluation.

An example of pattern matching definition is as follows:

```
map f [ ]     = [ ]
map f (x:xs)= f x : map f xs
```

Since h is the function of two arguments, hence the (h 3) is the function of single argument, adding the value 3 to this argument. The value of the application (map (h 3) [ ]) is [ ], applying the function (h 3) and matching the [ ] (Nil) pattern. The value of (map (f 3) [10, 20, 30]) is [13, 23, 33], because the map is applied recursively until the [ ] is matched.

Patterns are either variables or constants.

5.3. The Definition by Guarded Expressions. Instead of simple expression on right hand side guarded expression may be used, either with sequential or with parallel semantics.

The definition by sequential guarded expression is as follows:

```
seqselect x  = 10,   if x<5
             = 20,   if x>0
             = 30,   otherwise
```

The definition by parallel guarded expression is as follows:

```
parselect x  | x<5        = 10
             | x>0        = 20
             | otherwise  = 30
```

The application (seqselect 3) evaluates to 10, the application (parselect 3) may evaluate either to 10 or to 20, in an non-deterministic way. Of course, instead of constants, expressions on the right-hand side may be used.

In addition, in case of parallel guarded definition, these expressions may be sequentially guarded again. If the default True guard designated by **otherwise** keyword omits, in both cases the evaluation may fail.

5.4. Local Definitions. Local functions and/or processes may be defined behind **where** clause on the right hand side of the definition. In PFL local type definitions are allowed and they are obligatory for all user-defined local processes.

An example of the definition of local process h in a function f is as follows:

```
f x xs = h x xs
where
   h :: A a -> [a] -> [a]
   h x xs = x:xs
```

The value of application (f 'c' "omputer") is "computer". In addition, the character 'c' resides in local environment variable A until the value "computer" is evaluated. This has no significant meaning in this case, since if the application is evaluated, the local variable environment consisting of just one variable is de-allocated immediately.

6. PFL Variables are Mutable Abstract Types

A PFL variable is a memory cell, like in an imperative language. A value can be stored to and retrieved from this variable. On the other hand, there is much more important that the variables may be accessed and updated than that they are just containers for values.

An abstract type is a set of operations, which instances are defined using different definitions. Therefore a PFL variable V may be expressed in terms of abstract type, like a mapping

V :: $\grave{t}^{\sim}$ -> t

where $\grave{t}^{\sim}$ is either the unit type () or it is a data type t. Hence, V may be applied to an expression of both control and data type. The access instance of the mapping V is defined as follows:

```
V :: () -> t
V () = v
```

The update instance is defined as follows:

```
V :: t -> t
V x = x
```

The access instance is a 'constant'. The value of the expression V () is the value v, residing in the memory cell V. This definition mutates whenever the update instance is applied. The update is the identity, which however, assigns the applied value to the cell V. Applying V to v_1, the definition of the access is changed (it mutates) to:

V () = v_1

By the application (V v_2), the definition of the access mutates to:

V () = v_2

etc.

We may conclude that:

1. The type of the application of an instance of V is a data type, because the cell contains a data value, by no means the control value or a spatial value.

2. V is a built-in overloaded process. A programmer never defines it. The concept of memory cells like overloaded processes conform with the PFL concept. No memory cell is accessed and updated, referencing it directly, but rather by the application of processes.

3. Applying a V to the control value corresponds to the access of the value in an imperative language.

4. Applying a V to a data expression in PFL produces the value of this expression, not the control value like the update by assignment statement in an imperative language.

A programmer needs not to be familiar with mutable abstract types at all. It is sufficient to consider the variables like shared memory cells, and to known how the access and the update operates.

The access operates as shown in Fig. 1.

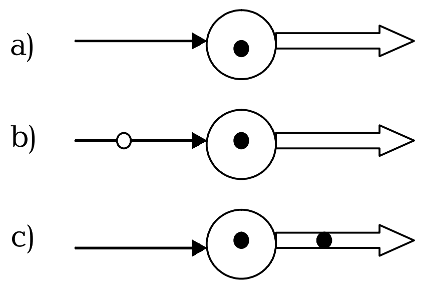

Fig. 1. The Access

Let the data value, marked by black circle resides in a variable V, according to Fig.1 a). Applying V (), the control value marked by white circle arrives through the input control arc, according to Fig.1 b). The value of the application is the data value, produced to the output data arc, the same like having been stored in V before.

The update is illustrated in the Fig. 2. In this case, both input and output arcs are the data arcs. Let the initial state in the Fig. 2 a) is the same like in the Fig. 1 a). Applying the variable V to a data value, marked by grey circle, according to Fig. 2 b), this value is assigned to V and produced to the output arc, according to the Fig. 2 c). Performing the update, the definition of the access has changed.

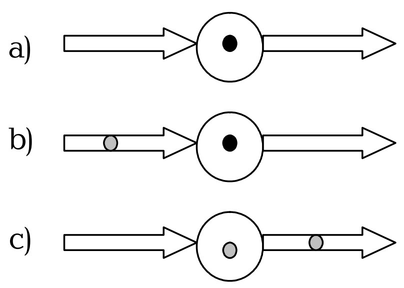

Fig. 2. The Update

Let us explain the semantics of user defined process more precisely, introducing an overloaded process f. Let the first instance of this process is as follows.

```
f :: a -> b -> ()
f x y = e'
```

where e'::(), i.e. the value of e' is the control value.

The second instance is a pure function, as follows:

```
f:: a -> b -> c
f x y = e
```

where e::t, i.e. e is of data type.

Now, let us define the PFL process p, as follows.

```
p :: A a -> B b -> C c
p x y = e
```

where e is the same expression like in the second (purely functional) instance of the process f. Provided that A, B, and C are the environment variables, the definition of p, may be expressed in terms of application of the overloaded process f, as follows:

```
p x y = C (f (A x) (B y))
```

The process p is implemented in the way of its definition above, but this form is not accessible to a user. The definition of p is more powerful than the definition of the second purely functional instance of f, since we allow the expression e not just of data type but also of unit type. The application of a process is however always the value of a data type. It may be noticed that this approach is inevitable to be able to apply the functional instance of f to the arguments (A x) and (B x), that must be of data type. Also this fact explains the reason why the update is not of the type t -> (), but of the type t -> t.

If there is a need for terminating the data flow, it is possible to define a process terminate, as follows:

```
terminate :: a -> ()
terminate x = ()
```

7. PFL Arrays are Spatial Processes

One of the benefits of spatial processes is that they express the arrays in a natural way allowing implementing them as efficiently like in an imperative language. The idea of spatial processes for arrays comes from the ability to define a function extensionally, i.e. by pattern matching. For example it is possible to define the function constArray, in the space of 2×3 points, as follows:

```
constArray   False  1  =  2.5
constArray   False  2  =  1.0
constArray   True   3  =  3.5
```

The value of the application (constArray False 1) is 2.5, the value of the application (constArray False 2) is 1.0, and the value of the application (constArray True 3) is 3.5. All the other items are undefined. The array constArray is a partial function of the type (bool -> int -> float), and it is the constant array. The items are not updatable, because the array was defined in the compile time. At the same time, there is no conceptual distinction between the spatial positions and the data.

PFL provides the opportunity to define and to redefine the arrays like partially defined processes in the run time, considering the spatial values and the spatial processes.

7.1. The Spatial Structure of a Value. The structure of the immaterial points in the space corresponding to the array positions defined above is expressed by the spatial type {bool, 1..3}. This spatial type, consisting of 6 immaterial positions is depicted in the Fig.3. The spatial process in the Fig.4, of the type {bool, 1..3} -> float, is extensionally defined mapping from spatial values to the values of the type float. At the same time the spatial process is a spatially structured value consisting of the spatially positioned values of the type float. The process in the Fig.4 is undefined, since all the positioned data are undefined.

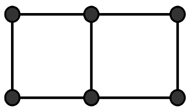

Fig. 3. The spatial values

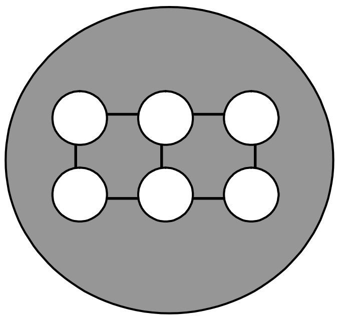

Fig. 4. The spatial process

The spatially structured value forming a spatial process is depicted in the Fig.4 by surrounding grey coloured circle including the spatially positioned data cells. It would not be confused with a spatial value of the spatial type according to the Fig. 3.

The spatial processes are retrieved from and stored to an environment variable V like data values, applying the built-in definitions for access and update as follows:

```
V :: () -> ({bool,1..3} -> float)
V () = v
```

and

```
V ::  ({bool, 1..3} -> float) ->
      ({bool, 1..3} -> float)
V x = x
```

So far, we are able to access the arrays and to update them. Below we will concentrate on how to access and update the array items.

7.2. Accessing the Argument Items. Provided that f is a PFL function, defined as follows:

```
f :: ({bool, 1..3} -> float) -> int -> float
f x i = x{True, i} + x{False, i+1}
```

the expression (f a 2) summarises the {True, 2}-th and the {False, 3}-th item values of the array given by an expression a.

Notice, that x :: {bool, 1..3} -> float, i.e. x is a value. The same holds however, if the process p is defined as follows:

```
p :: V ({bool, 1..3} -> float) -> int -> float
p x i = x{True, i} + x{False, i+1}
```

In this case (p () 2) is legal application and x is a current value of the environment variable V, of the type ({bool, 1..3} -> float). From the implementation point of view, the value x is a base address of an array, i.e. it is the input address of extensionally defined spatial process. The input address of a spatial process accesses the data area of the memory, while that of an ordinary function or process accesses the code area. The value of an updatable array is referentially non-transparent while the value of the function is referentially transparent. The reasons for referential non-transparency of spatial processes and ordinary processes are different. Spatial processes are non-transparent for their internal state changes and ordinary processes for the external state changes.

7.3. Incremental Update. Provided that $\{e_{bool}, e_{int}\}$::{bool, 1..3}, let us suppose the mapping as follows:

I^0 :: float -> ({bool, 1..3} -> float)

I^0 (x $\{e^0_{bool}, e^0_{int}\}$) = x

Clearly, this mapping may not be expressed in an ordinary lambda calculus by lambda abstraction, since instead of lambda variable there is an expression (x

$\{e^0_{bool}, e^0_{int}\}$) of the type float on the left hand side used. Furthermore, it would be poor effect if both e^0_{bool} and e^0_{int} are constant expressions.

On the other hand, provided that x is a spatially structured value, (on the basis of right hand side) the mapping l^0 is well defined in terms of lambda proposition. Informally, if x is a lambda variable — a hole containing the value of the type ({bool, 1..3} -> float), then (x $\{e^0_{bool}, e^0_{int}\}$) is a sub-hole positioned in the space point $\{e^0_{bool}, e^0_{int}\}$. Then of course, the value of x is defined by the application of a mapping l^0 to an expression e^0 of the type float, in the form as follows:

$l^0\ e^0$

or, in terms of the application of lambda proposition as follows:

$(\lambda\ (x\ \{e^0_{bool}, e^0_{int}\}).x)\ e^0$

Since, according to the lambda expression above, the value of the array has been partially defined (just for the $\{e^0_{bool},e^0_{int}\}$-th item by the value e^0), but the array like a value remains unchanged, we may use it again to define the next item as follows:

$(\lambda((\lambda\ (x\ \{e^0_{bool}, e^0_{int}\}).x)\ e^0)\{e^1_{bool}, e^1_{int}\}.x)\ e^1$

The PFL expression corresponding to the incremental definition of partially defined array is as follows:

$x\ \{e^0_{bool}, e^0_{int}\} \to e^0\ \{e^1_{bool}, e^1_{int}\} \to e^1$

of the type as follows:

({bool, 1..3} -> float)

7.4. Updating the Argument Items. Clearly, since x as well as the variables used in all expressions must be lambda variables, this expression may be used on the right hand side of a PFL function, such as:

```
f ::({bool, 1..3} -> float) -> ... ->
     {bool, 1..3} -> float
f x ... = x   {e^0_bool, e^0_int} -> e^0
              {e^1_bool, e^1_int} -> e^1
```

or a process, such as:

```
p :: V ({bool, 1..3} -> float) -> ... ->
        {bool, 1..3} -> float
p x ... = x  {e^0_bool, e^0_int} -> e^0
             {e^1_bool, e^1_int} -> e^1
```

The dots in the definitions above designate further patterns used in expressions on the right hand sides and the dots in the type definitions designate the types of these patterns.

Applying the function f or the process p, the array like a value remains unchanged, not however to the value of this array. It has changed defining the two items. The array arguments are updated, after they are applied. In the second case, the application (p () ...) updates the current array retrieved from V.

The same holds if the terminate process is applied to the right hand side expressions, according to the definitions, as follows:

```
f :: ({bool, 1..3} -> float) -> ... -> ()
f x ... = terminate (x {e^0_bool, e^0_int} -> e^0
                       {e^1_bool,e^1_int} -> e^1)
p :: V ({bool, 1..3} -> float) -> ... -> ()
p x ... = terminate (x{e^0_bool, e^0_int} -> e^0
                       {e^1_bool, e^1_int} -> e^1)
```

In this case the value of both f and p is the control value, not the array.

7.5. Updating the Value Items. Let us suppose the process p, such that x is not a lambda variable of the process p, defined as follows:

```
p :: ... -> {bool, 1..3} -> float
p ... = x {e^0_bool, e^0_int} -> e^0
          {e^1_bool, e^1_int} -> e^1
```

Clearly, since x is free variable, the definition above is wrong. A possible solution to this problem is to construct the new array and to develop a mechanism how to update it in place. Or, like it is in PFL, not to construct the arrays at all, just to define them. The correct PFL definition for updating the array by the value is as follows:

```
p :: ... -> V ({bool, 1..3} -> float)
```

p ... = $\{e^0_{bool}, e^0_{int}\}$ -> e^0 $\{e^1_{bool}, e^1_{int}\}$ -> e^1

Considering the fact that the array like a value in contrast to data value is defined by initialisation and it may be retrieved by the application (V ()), the right hand side expression is just syntactic sugaring for the application of lambda proposition in the form, as follows:

$(\lambda ((\lambda (V() \{e^0_{bool}, e^0_{int}\}).V()) e^0)$
$\{e^1_{bool}, e^1_{int}\}.V()) e^1$

Notice however, that the value of the application of the process p, defined as follows:

```
p :: ... -> V ({bool, 1..3} -> float)
p ... = terminate (x {e⁰bool, e⁰int} -> e⁰
                     {e¹bool, e¹int} -> e¹)
```

is not of the unit type, but it is of the type ({bool, 1..3} -> float).

Concluding, we may define the values of the items of our array accessed via an argument of a process, as follows:

```
array :: Array ({bool, 1..3} -> float) -> ()
array  x = x  {False, 1} -> 2.5
              {False, 2} -> 1.0
              {True, 3}  -> 3.5
```

Evaluating (array ()) we obtain ().

Or, we may define the values of the items of an array accessed by a value of the process array, as follows:

```
array :: Array ({bool, 1..3} -> float)
array   =  {False,1}  -> 2.5
           {False, 2} -> 1.0
           {True, 3}  -> 3.5
```

Then, evaluating the 'constant' array the (partially defined) array is obtained.

7.6. Accessing the Value Items. Considering the lambda proposition application for updating the value items, we have

V () $\{e^0_{bool}, e^0_{int}\}$ :: float

Using the same syntactic sugaring like for the update, the PFL syntactic construct for accessing the value (the target array) item is derived, in the form as follows:

$\{e^0_{bool}, e^0_{int}\}$

Like an example, let us define

```
incr ::  ({1..5} -> int) -> int ->
         A ({1..5} -> int)
incr x i = {i} -> {i} + x{i}
```

Then the application (incr a 3) increments the 3-rd item of the array A by the value of the array given by an expression a.

On the other hand, defining

```
incr ::  ({1..5} -> int) -> int ->
         A ({1..5} -> int)
incr x i = x{i} -> {i} + x{i}
```

the 3-rd item of the array a is incremented leaving the array A unchanged.

We may conclude that:

1. A programmer must decide which (argument or value) array is updated, mixing is not allowed in an incremental update.
2. The items of an array may accumulate the values easily.
3. The use of the arrays is symmetrical; the items of both argument and value arrays are accessible and updatable.
4. The side effects on arrays are local to the processes of the array arguments an/or values.
5. The arrays are not constructed, but rather defined like the spatial processes with the internal state.

8. Comprehensive Expressions

8.1. List Iterations. Provided that e_1, e_2, e_3:t, where t is int, char, float or an enumerated type, list iterations of the type [t] are in the following forms:

```
[e1 .. e3]
[e1, e2 .. e3]
[e1 ..]
[e1, e2 ..]
```

The first two list iterations produce finite lists. For example, the value of [2 .. 5] is the list [2, 3, 4, 5], the value of [5 .. 2] is the list [ ], the value of [5, 3 .. 2] is the list [5, 3] , the value of ['a', 'c' .. 'h'] is the list "aceg". The second two list iterations produce the infinite lists. For example, the value of [2.0 ..] is the list [2.0, 3.0, 4.0, etc. and the value of [2.0, 1.9 ..] is the list [2.0, 1.9, 1.8, , etc.

Clearly, PFL recursive data values, are constructed lazily.

8.2. List Comprehensions. Provided that e::t, where t is any data type, list comprehension is an expression of the type [t] as follows

[e | Q_1, ... , Q_n]

where Q_k is either the filter — a boolean expression or the list generator in the form, as follows:

p_k <- e_k

Provided that e_k::[t_k], the pattern p_k::t_k.

The expression [e |] with the empty list of qualifiers evaluates to [e |].

For example, the list comprehension, as follows:

[(x, y) | x <- [3..5], y <- [1,2]] evaluates to

[(3, 1), (3, 2), (4, 1), (4, 2), (5, 1), (5, 2)].

The list comprehension

[(x, y) | x <- [3..5], y <- [1, 2], even (x+y)] evaluates to [(3, 1), (4, 2), (5, 1)], and the list comprehension [x+y | (x, y) <- [(1, 2), (10, 20)]]

evaluates to [11, 22].

8.3. Loop Comprehensions. Provided that e::(), loop comprehension is an expression of the type (), as follows:

(e | Q_1, ..., Q_n)

where Q_k is either the filter — a boolean expression or the loop range generator.

Provided that v, e_1, e_2, e_3::t, where t is int, char, float or an enumerated type, loop range generator is in one of the next forms:

v <- {e_1 .. e_3}

v <- {e_1, e_2 .. e_3}

where v is a variable and e_1, e_2, and e_3 are expressions.

The expression (e |) with the empty list of loop qualifiers evaluates to (e)., of the value ().

The variable v is local to a loop comprehension and may be implemented using local variable environment. Since the value of the expression e must be the control value, the expression e may be evaluated in an iterative way using the generated value v for each iteration step. The next value is assigned to v safely, since the expression e is a process, hence it is evaluated eagerly.

8.4. Array Aggregates. Provided that the argument x of a PFL function/process is an array, it is updated using the construct, in the forms:

x{$e^0{}_1$, ..., $e^0{}_n$} -> e^0

x{$e^0{}_1$, ..., $e^0{}_n$} -> e^0{$e^1{}_1$, $e^1{}_n$} -> e^1

x{$e^0{}_1$, ..., $e^0{}_n$} -> e^0{$e^1{}_1$, $e^1{}_n$} -> e^1{$e^2{}_1$, $e^2{}_n$} -> e^2

etc. The value array of a PFL function or process is updated using the construct, as follows:

{$e^0{}_1$, ..., $e^0{}_n$} -> e^0

{$e^0{}_1$, ..., $e^0{}_n$} -> e^0 {$e^1{}_1$, $e^1{}_n$} -> e^1

{$e^0{}_1$, ..., $e^0{}_n$} -> e^0 {$e^1{}_1$, $e^1{}_n$} -> e^1 {$e^2{}_1$, $e^2{}_n$} -> e^2

etc. As it has been shown already above, these expressions are lambda proposition applications and may be used in PFL definitions producing the arrays of the type

{R_1, ..., R_n} -> t

provided that {$e^k{}_1$, ..., $e^k{}_n$}::{R_1, ..., R_n} and e^k::t.

On the other hand we have defined the process terminate, which terminates the data flow, mapping a data value to the control value.

Therefore the expressions, such as

terminate(x {$e^0{}_1$, ..., $e^0{}_n$} -> e^0)

terminate({$e^0{}_1$, ..., $e^0{}_n$} -> e^0)

updates the argument or value arrays producing the control value and would be used in loop comprehensions in the form, for example, as follows:

(terminate(x {$e^0{}_1$, ..., $e^0{}_n$} -> e^0) | Q_1, ..., Q_n)

(terminate({$e^0{}_1$, ..., $e^0{}_n$} -> e^0) | Q_1, ..., Q_n)

If the terminate is applied to lambda proposition application in the loop comprehension implicitly, we obtain the form for array aggregates, such as follows:

$(x\{e^0_1, ..., e^0_n\} \text{->} e^0 \mid Q_1, ..., Q_n)$

$(x\{e^0_1, ..., e^0_n\} \text{->} e^0\{e^1_1, e^1_n\} \text{->} e^1 \mid Q_1, ..., Q_n)$

....

$(\{e^0_1, ..., e^0_n\} \text{->} e^0 \mid Q_1, ..., Q_n)$

$(\{e^0_1, ..., e^0_n\} \text{->} e^0 \{e^1_1,e^1_n\} \text{->} e^1 \mid Q_1, ..., Q_n)$

....

The value of array aggregate is the control value. Hence, if a process is defined by array aggregate which updates the argument array, the type of the value of this process is either () or V t, where t is any data type. However, if the array aggregate updates the value array, the type of the value of the process in the type definition must be V ($\{R_1, ..., R_n\}$ -> t, where t is any data type.

9 . Abstract Types and Objects

The PFL abstract type is a set of functions and/or processes called member functions and/or processes, that are defined by a class definition and at least one instance definition.

A general rule is that the class definition consists of just type definitions, called type signatures and an instance definition consists of the definitions. However, if all the definitions of a member function are the same, instead of including them into each instance, it is possible to introduce just one definition in the class. For example, we may define the class Stack and its two instances, as follows:

```
class Stack a where
   push  :: a -> [a] -> [a]
   pop   :: [a] -> [a]
   top   :: [a] -> a
   empty :: [a] -> bool
   empty [ ]      = True
   empty (x:xs)   = False
instance Stack int where
   push  x xs   = x : xs
   pop   (x:xs) = x
   top   (x:xs) = True
instance Stack [a] where
   push  x xs  = xs ++ [x]
   pop   xs    = init xs
   top   xs    = last xs
```

We have defined three pure overloaded functions, namely push, pop and top. Defining two instances for push (the same holds for pop and top) the same name designates two different definitions.

Applying push to an integer, this integer is pushed like the first element, which applying push to a list this list is appended. For example, the value of (push 3 [4, 2]) is [3, 4, 2], whilst the value of (push "alpha" ["delta", "gamma"]) is ["delta", "gamma", "alpha"].

On the other hand, the predicate empty is defined for all defined instances, since it is defined in the class definition. The member functions of a class, such as push, pop, top and empty are accessible to global functions and vice versa. Since the functions last and init such that

```
init (xs++[x]) = xs
last (xs++[x]) = x
```

are defined on the global level, they are automatically inherited by a member functions.

The function empty must be defined for all instances that are defined, otherwise it is impossible to recognise which its instance is applied in the expression (empty [ ]). On the other hand, if we decided for the same definition of a member function for all instances, such as for Stack int for example, this member function may be defined in class definition. If all instances of member functions have the same definitions, then we may define a class:

```
class Stack a where
   push   :: a -> [a] -> [a]
   push   x xs  = x : xs

   pop    :: [a] -> [a]
pop   (x:xs)    = x

top        :: [a] -> a
top   (x:xs)    = True

empty   :: [a] -> bool
empty [ ]       = True
empty (x:xs)  = False
```

followed by

instance Stack int
instance Stack [a]

Then the value of (push 3 [4, 2]) is [3, 4, 2], and the value of (push "alpha" ["delta", "gamma"]) is ["alpha", "delta", "gamma"]. It is easy to see, that this approach may be used for all monomorphic classes, since each monomorphic class has just one instance.

9.1. The Inheritance. If a member function of a class is applied in another instance definition (or a default definition introduced in another class definition, it is inherited using context in correspond-ding instance or class definition.

The context is given by a list of type expressions in the form

$(C_1\ t^1{}_1 \dots t^{n1}{}_1, \dots , C_m\ t^m{}_m \dots t^{nm}{}_m)$ =>

where $C_k\ t^1{}_k \dots t^{nk}{}_k$ is an instance of the k-th class C_k. It is possible to specify one or more instances of one or more classes. It is also possible to specify all the instances of one class. Then, the type expressions $t^1{}_k \dots t^{nk}{}_k$ are in the form of type variables $a^1{}_k \dots a^{nk}{}_k$.

For inherited monomorphic classes C_k the type expressions omit, since $n_k = 0$.

Like an example, provided that any instances push, pop, etc. are required to be applied in any instance of the class C we may write a class header for C as follows:

class (Stack a) => C a **where**

If we require the overloaded functions defined in an instance of Stack class, for all member defined in the class C, we may write

class (Stack int) => C a **where**

or

class (Stack [a]) => C a **where**

If all the members of the class Stack are inherited by just one instance of a class C then we may write the instance header for an instance of class C, such as follows:

instance (Stack a) => C char **where**

Finally, it is possible to inherit the instance of the class Stack by an instance of a class C, by the headers, such as:

instance (Stack int) => C [a] **where**
instance (Stack [a]) => C char **where**

In PFL the context just specify the accessibility of inherited instances of classes. The concrete overloaded definition is recognized on the basis of the types of arguments. That is why we may require to inherit the polymorphic classes or instances in a monomorphic classes one, such as:

class (Stack a) => D **where**
class (Stack [a]) => D **where**

The purely functional approach above may be extended to member processes of the unit types in type signatures. The approach becomes different when there is an effort to define the processes working with the variable environments.

9.2. Objects. The ability to access and update the variable environment for global and local processes has been shown above. Here we will concentrate to the dynamically allocated variable environments for the member processes forming the objects. PFL provides opportunity for object oriented programming.

Let us define the polymorphic class Buffer, as follows:

class Buffer a **where**
```
init :: N int -> H int -> T int -> ()
init n h t = ()

sendItem :: N int -> T int ->
            B({0..99}->a)-> a -> ()

sendItem n t b v
```

```
        = b{newNT (n+1)
            ((t+1) `mod` 100)} -> v

    receiveItem :: N int -> H int ->
                        B({0..99} -> a) -> a

    receiveItem n h b
        = newVN (b{newH
                ((h + 1) `mod` 100)}) (n – 1)

    empty :: N int -> bool
    empty n = n = 0

    full :: N int -> bool
    full n = n = 100

    newNT :: N int -> T int -> int
    newNT n t = t

    newH :: H int -> int
    newH h = h

    newVN :: a ->N int -> a
    newVN v n = v
```

To be able to use buffer for integers and and floating point numbers, let us define the instances, as follows:

```
instance Buffer int
instance Buffer float
```

Class members init, empty, full, newNT and newH are monomorphic processes. Therefore the same code is generated for them. On the other hand, for each polymorphic process, such as sendItem, receiveItem and newVN it is necessary to generate two different target codes, one for integer instance and one for floating point instance. However, this is still not sufficient enough, since the processes work with the variable environment which consists of the variables N, H, T and B. For example, if there is a need to work with two integer buffers at the same time, then two copies of variable environment are needed.

A new copy of variable environment for an instance of the PFL class is allocated using type expression in an expression. A PFL object is a copy of this variable environment, together with the instances of class members which access it. Provided that b1 is an object variable environment allocated by a type expression (Buffer int), we may initialize the integer buffer by the application

```
b1=>init 0 0 0
```

If b2 is an object variable environment allocated by the application (Buffer int) again, then

```
b2=>init 0 0 0
```

initialises the different integer buffer, using the same code for init. The two objects differ just by their environments, not by the processes that access them. On the other hand, allocating the object variable environment b3 by (Buffer float), a new buffer of floating point numbers is initialized using

```
b3=>init 0 0 0
```

by the same init process like for integer buffers, but the item is sent by the different code of sendItem.

The life-time of an object variable environment is determined just by its use, like the life time of data cells constructed by constructors of algebraic types. The only difference is that the values of data cells are evaluated immediately, but the values of variable environment are undefined, when the variable environment is allocated.

Hence, the update of an object variable environment sharing it by two or more processes may be performed extending the types of arguments and/or values to abstract types. An example of processes with the arguments of abstract type Buffer a is introduced below.

10. Parallelism

In this section we will concentrate more on the PFL ability to express parallel programs, than on the exhaustive description of potential power of parallel process functional approach. At the same time, the following example explains how the objects are used.

Let us consider simple producer consumer problem that is to be solved in the time-sharing system. Let we have three producers and three consumers; the first producer produces the values for the first consumer, the second producer for the second consumer and the third producer for the third consumer. Therefore we need three buffers. Let us suppose the first two buffers contain integers and the third one contains floating point numbers.

Let the values, which will be produced, are evaluated by applying the next processes:

```
data1 :: () -> int
data2 :: () -> int
data3 :: () -> float
```

We omit the definitions, which may be quite different, according to the purposed application.

In general, the potential definitions of data1, data2, and data3 are not referentially transparent. It means that repeated application of data1 (), (as well as data2 (), and data3 ()) may produce the different values, although no free variables occur in the definitions.

Let the consumed values are used by the next processes:

```
use1 :: int -> ()
use2 :: int -> ()
use3 :: float -> ()
```

Again, the omitting definitions are highly dependent on the application.

A synchronous message passing by one item may be expressed by the definition of main program, as follows:

```
#main = (  use1(data1()) ||
           use2(data2()) ||
           use3(data3()) )
```

i.e. the three expressions are executed in parallel.

However, we require each use(data()) be evaluated repeatedly and both the use and the data be evaluated asynchronously, using a buffer for message passing.

First, let us produce and consume the items in an infinite loop, as follows:

```
producer :: Buffer a -> (() -> a) -> ()
producer b d = loop (prodItem b d)
where
  loop::()->()
  loop () = loop (proditem b d)
consumer :: Buffer a -> (a -> ()) -> ()
consumer b u = loop (consItem b u)
where
  loop::() -> ()
  loop () = loop (consItem b u)
```

The definitions above use the object of an abstract type Buffer, and the process, which either evaluates or uses the single item data.

The processes that produce a single item and consume the single item are as follows:

```
prodItem :: Buffer a -> (()->a) -> ()
prodItem b d = ( b => sendItem () () () (d())
                 | not (b => full() ) )

consItem :: Buffer a -> (a -> ()) -> ()
consItem b u = (u(b => receiveItem () () ())
                 | if not (b => empty() ) )
```

The aim of these processes is either to send just evaluated item to the buffer, if the buffer is not full, or to use just received value from the buffer.

Let the representation of the buffer is a ring queue of items, accessed by head and tail respectively. Since of its finite representation the actual number of the items must be checked. Hence, the buffer is initialized by process initbuffer, which is defined as follows:

```
initbuffer :: Buffer a -> ()
initbuffer b = b => init 0 0 0
```

The parallel work with three buffers is expressed by the process parWith, as follows:

```
parWith :: Buffer a -> Buffer a -> Buffer a
             -> ()
parWith b1 b2 b3
= ((initbuffer b1;
        (producer b1 data1 ||
                consumer b1 use1) ) ||
 (initbuffer b2;
        (producer b2 data2 ||
                consumer b2 use2) ) ||
 (initbuffer b3;
        (producer b3 data3 ||
                consumer b3 use3) ) )
```

Finally, the main is defined by the application of parWith to three buffers.

```
#main = parWith (Buffer int) (Buffer int)
                (Buffer float)
```

In the example above we have used the built-in parallel application operation, defined as follows:

```
(par2) :: () -> () -> ()
(par3) :: () -> () -> () -> ()
(par4) :: () -> () -> () -> () -> ()
```

Instead of (par3 e1 e2 e3) we may write (e1 || e2 || e3).

Analogously, the sequential application is defined as follows:

```
(seq2) :: () -> () -> ()
(seq3) :: () -> () -> () -> ()
(seq4) :: () -> () -> () -> () -> ()
```

Again, instead of (seq3 e1 e2 e3) we may write (e1; e2; e3).

Conclusion

In this paper we have presented:

- The essential conception of PFL — a process functional programming language
- The style in which process functional programs are constructed
- The principle of updating the memory cells using mutable abstract types
- The application of spatial types in spatial processes.
- PFL ability for object oriented and parallel programming.

PFL is a general-purpose experimental language aimed for reliable specification and efficient implementation of complex software systems. Many of ideas come from purely functional languages as well as from the concepts of mutable abstract types and monads that were implemented in Haskell or Gofer.

Like pure functional languages, PFL prevents the use of assignments in programming, however, it does not suppress the role of the control. Instead of that, the control is expressed in a rigorous way via the control value. It means that the specification of the control is not based upon an ad-hock sequencing of the statements, but rather on the application of processes. Actually, the PFL functions view the data values directly. The processes are eager functions such that they view either the control values directly, or they view the data values indirectly — via the environment variables. The concept of spatial types allows us to manipulate arrays in a natural way and implement them efficiently.

From the viewpoint of programming methodology, PFL is not restricted to strictly mathematical specification in a purely functional manner, since a programmer deals with the state manipulation. In this sense, a programmer has to take into account the importance of the timing and its role to updating the state. From one point of view, this task may be performed in an intuitive way, but once a PFL script is specified, it expresses the software design process in the mathematical form, which may be reasoned about and profiled. For example, we may reason about the order, in which the contents of variables may change, about what grains of the specified script are purely functional, about the response times when a process or function is applied, etc. At the same time, this mathematical form is strongly related to the implementation. Hence, the profiling is not the deal of some post-mortem analysis, but the activity, which may help to specify a system satisfying the requirements in the design phase already.

Seemingly the PFL specification is non-transparent, especially considering "too much" control values like arguments. However, if an imperative program is expressed in PFL, we would see the enormous amount of control values arising, that are manipulated using an imperative language in an undisciplined and a very dangerous manner. In fact, the number of control values decreases rapidly, when the same problem is specified in PFL. Moreover, at this stage of the research, we are using just textual form of PFL language, like a basis for further development. A PFL compiler is under the construction, and we think about it like a basis for further research experiments.

Although we have illustrated the PFL ability for parallelism just for time sharing system, a heterogeneous network of computers, in role of the supercomputer seems to be good environment for proving the strength of a process functional approach in praxis. At the first stage, the aim is to bind a sequential version of PFL language to MPI — a message passing interface standard [19, 20]. At the second stage, we will support MPI in PFL directly, similarly like it is in mpC [21, 22].

The further development of spatial types is necessary, not to be restricted just to arrays, allowing an efficient specification of the problems, such as having been dealt in computer graphics and virtual reality [29, 30], like an interesting application in our department.

Since each imperative program may be expressed in terms of PFL expressions, PFL may contribute to software standardisation. Once the methods of program transformation on the basis of PFL are available, and a cross-compiler from an imperative language to PFL is developed, there is an opportunity to maintain an existing software systems, having been developed on an ad-hock combinatorial imperative basis in an exact mathematical manner. Therefore, like a starting point, the development of profiling tool for PFL language in the future is crucial.

Currently the work goes on the detailed specification and implementation of the concepts having been presented in this paper.

This work was supported by the VEGA Grant No.1/5265/98.

1. *Achten P., Plasmeijer R.* Interactive Functional Objects in Clean // IFL'97, LNCS 1467, 1998. — P. 304—321.
2. *Axford T.* Concurrent Programming. — J. Willey and sons, 1988. — 250 p.
3. *Harper R., MacQueen D., Milner R.* // Standard ML. ECS-LFCS-86-2, LFCS Report Series, University of Edinburgh, Department of Computer Science, 1986. — 35 p.
4. *Hudak P.* Mutable abstract datatypes — or — How to have your state and munge it too, Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-914, December 1992, revised May 1993.
5. *Hudák Š., Teliopoulos K.* Formal Specification and Time Analysis of Systems. Proc. of the International Scientific Conf. ECI'98, Košice-Herľany, Slovakia, October 8—9. — P. 7—12.
6. *Hudák Š., Teliopoulos K.* TB nets: Properties of Time Interval Profiles. In: Proceedings of International Conference RSEE'97, (T. Maghiar Ed.), Oradea, Romania, May 30—31, 1997. — P. 25—30.
7. *Hudák Š., Teliopoulos K.* Merging Firing Sequences of P-decomposed Petri Nets. Proceedings of International Conference RSEE'97, Oradea, Romania, May 30—31, 1997. — P. 31—36.
8. *Hudák Š., Teliopoulos K.* Loop spectral analysis of time reachability problem. In: Proceedings of the 2-nd International Conference RSEE'98, Oradea, Romania, May 27—30, 1998. — P. 51—61.
9. *Jones M. P., Hudak P.* Implicit and Explicit Parallel Programming in Haskell, Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-982, August 19, 1993.
10. *Jones M.P.* An Introduction to Gofer — Functional programming environment, Version 2.20. draft, 1991.
11. *Kollár J.* Process Functional Programming. Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic, April 27—29, 1999. — P. 41—48.
12. *Kollár J.* Comprehending Loops in a Process Functional Programming Language. May 1999 // Computers and AI. — 15 p.
13. *Kollár J.* Control-driven Data Flow. May 1999 // J. of Electrical Engineering. — 10 p.
14. *Kollár J.* Process Functional Arrays // Proc. Intl. Conf. RMEES'99, Engineering of Modern Electric Systems, Felix-Spa, Romania, 1999.
15. *Kollár J.* PFL Expressions for Imperative Control Structures // Proc. of Computer Engineering and Informatics Scientific conference with International participation, Oct 14—15, 1999, Herľany, Slovakia. P.23—28.
16. *Launchbury J., Peyton Jones S.L.* Lazy Functional State Threads. Computing Science Department, Glasgow University, 1994. — 17 p.
17. *Meyer B.* Object-oriented Software Construction. Prentice Hall, 1985.

18. *Milner R.* A theory of type polymorphism in programming // J. of Computer and System Science. — Vol.17, 1978. — P. 348—375.
19. *MPI*: A Message-Passing Interface Standard. Message Passing Interface Forum June 12, 1995, University of Tennessee, Knoxville, Tennessee. — 231 p.
20. *MPI-2*: Extensions to the Message-Passing Interface. Message Passing Interface Forum, July 18, 1997, University of Tennessee, Knoxville, Tennessee. — 362 p.
21. *The mpC* Programming Language Specification. Russian Academy of Sciences Institute for System Programming, June 1997. — 66 p.
22. *A Programming* Environment for Heterogenous Distributed Memory Machines / Dmitry Arapov, Alexey Kalinov, Alexey Lastovetsky, Ilya Ledovskih, Ted Lewis // Proc. of 6th Heterogenous Computing Workshop (HCW'97), IEEE Computer Society, Geneva, Switzerland, April 1997. — P. 32—45.
23. *Novitzká V.* Proof systems by institutions // Proc. CEI'99 Scient. Conf., Košice-Herľany, Slovakia, October 8—9. — P.7—12.
24. *Novitzká V.* Systems for Deriving Correct Implementations // Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic, April 27—29, 1999. — P. 201—207.
25. *Novitzká V.* When is the program correct? // Analele Universitatii din Oradea, fascicola Electrotehnica, Oradea, Romania, 1998. P. 70—74.
26. *Peterson J., Hammond K.*, editors: Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. Version 1.3. Yale University, May 1996. — 164 p.
27. *Peyton Jones S.L., Wadler P.* Imperative functional programming // 20th Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993. — P. 71—84.
28. *Reade C.* Elements of Functional Programming. Prentice-Hall, 1989. — 600 p.
29. *Sobota B., Valigurský M.* Real Time Visualising Kernel for Virtual Reality Applications // Proceedings of International Conference Modelling and Simulation of Systems MOSIS'98, Bystřice p. Hostýnem, May 5—7, 1998. P. 117—122.
30. *Sobota B., Janošo R., Paralič M.* The distributed raytracing in the virtual reality systems // Proceeding of the Scientific Conference with International Participation "Informatics and Algorithms", Prešov, Slovakia, September 3—4, 1998. P. 64—68.
31. *Wadler P.* The essence of functional programming // 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992. — P. 23.

Date received 02.02.04

About author

Dr. Ján Kollár

Technical University of Košice
Department of Computers and Informatics,
Letná 9, 042 00 KOŠICE, Slovak Republic

Phone: +421 55 602 2577
Fax: +421 55 633 0115
E-mail: Jan.Kollar@tuke.sk