TYPE ENVIRONMENTS IN OBJECT ORIENTED PROCESS FUNCTIONAL LANGUAGE

Peter VÁCLAVÍK, Jaroslav PORUBÄN *
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, tel. 095/602 3175,
E-mail: Peter.Vaclavik@tuke.sk, Jaroslav.Poruban@tuke.sk

SUMMARY

The state of a system is expressed using PFL, a process functional language, in an easily understandable manner. The paper presents PFL environment variable - our basic concept of the state manipulation in the process functional language. The concept of the process functional language has been extended with an object-oriented paradigm features. The paper presents an abstract syntax of the object-oriented PFL and describes the syntax of types and type environments. In the paper main ideas of object-oriented PFL type checking and program translation in a compiler are explained. Finally the paper presents an example where type environments are inferred as a result of PFL program translation. Presented results described in this paper were implemented in PFL compiler.

Keywords: *process functional language, object-oriented functional programming, type environment, type unification*

1. INDRODUCTION

Object-oriented concepts are ubiquitous in programming. Objects may model real life entities, or may represent system artifacts like stacks. Objects provide a way to structure a system and to control the computation. One of the characteristics of object-oriented programming is inheritance, which allows new classes to be defined as increments of existing ones. Inheritance comes with dynamic binding, inclusion polymorphism (subtyping) and method overriding.

In a functional language, a programmer declares referential transparent functions. Whenever a function is called, it returns the same result for the same arguments. Thus, functional programming languages allow for elegant reasoning. Furthermore, functional languages support specification and programming at a rather high level using type inference, parametric polymorphisms, high-order functions, lazy evaluation and algebraic data type.

PFL, an experimental process functional language [5], is somewhere between imperative and pure functional languages, since the variable environment is visible to a user, and the source definitions of processes are purely functional, i.e. without assignments and without environment variables. This supports, as we believe, the simplification of the systems design and, at the same time, the simpler reasoning about the systems. From the viewpoint of implementation, *PFL* is somewhere between an impure eager functional language and a monadic lazy pure functional language. It is because the different binding of variable environment to *PFL* processes than that can be found in impure languages. So, the state of a system is expressed using *PFL* in an easily understandable manner. Coming out from monad and state transformers theory we define our concept of *PFL* variable environment.

The research of languages that combines functional and object-oriented features has a long history. The aim is to integrate the formal methods benefits of functional programming, comfort to a programmer given by higher level of abstraction with the software engineering benefits of both paradigms.

The approaches to integrate both paradigms can be divided into two categories. The first category comprises effort to extend functional language with object-oriented features [2,4,7]. The second one covers efforts to extend imperative object-oriented language with functional features such as higher-order functions, algebraic data types and parametric polymorphism [8,9].

In the section 0 we present basic approach to object-oriented programming in a process functional language. Nowadays the process functional language combines imperative, functional and object-oriented paradigms of programming. Process functional language has been extended to object-oriented language in a past few years and two compilers have been already implemented that compiles process functional code to Java and Haskell language.

In the section 0 the types and type environments of a *PFL* program translation are described. Even though there is solution for PFL presented the one is applicable in other programming languages as well. This approach was inspired with Haskell. Short example concludes the section.

* This work was supported by VEGA Grant No. 1/1065/04 - Specification and implementation of aspects in programming.

2. THE CURRENT STATE IN OBJECT ORIENTED *PFL* RESEARCH

PFL is an extended subset of Haskell programming language. It is strongly typed functional language supporting pattern matching, local and global function definitions, algebraic and abstract data types, primitive function definitions, operator definitions, global and local variable environments.

PFL type system comprises unit type `()` as it is in Haskell. This type comprises just control value, representing the control. It means for example, that it is impossible to mix data and control arguments for constructors of algebraic data types.
Let T be a data type. Then the type

$$\widetilde{T} = T \cup ()$$

ranges over a data type and unit type. The *PFL* process is similar to a function. Definition of a *PFL* process is same as a definition of a pure function. The only difference is in its type definition. Type definitions for processes are obligatory. The type definition of a process extends a syntax and semantics of a pure function type definition. The type definition of a process comprises either `()` for an argument or the value type, or an argument type in the form $v\ T$, where v is an environment variable and T is a data type. Examples of processes type definitions are provided later. The idea of incorporating of some aspects in function type definitions is also presented in Clean with its uniqueness attributes [10].

Functions are first-class values in *PFL*. On the other hand processes are not. Process cannot be passed as an argument to a function or returned as the result of a function. There is no partial process application. This approach has been chosen to simplify identification of program parts manipulating the state. The *PFL* static analyzer finds parts of a program affected by the state - processes.

The well-known and commonly accepted concept of the variable environment in both imperative and impure functional languages is as follows. The variable environment **Env** is a mapping from variables to their values. If $\rho :: \alpha \to \beta$ is environment and $a \in \alpha$, $b \in \beta$, then the update expression is as follows.

$$(\rho\,[a \mapsto b])\,x = \begin{cases} b & ,\text{if } x = a \\ \rho\,x & ,\text{if } x \neq a \text{ and } \rho \neq \varnothing \\ \bot & ,\text{if } x \neq a \text{ and } \rho = \varnothing \end{cases}$$

Symbol $\varnothing$ is used to define empty environment.

The variable environment **Env** is defined using the update expression.

```
Env = Var → Value
access :: Var → Env → Value
access x e = e x
update :: Var → Value → Env → Env
update x v e = e [ x ↦ v ]
```

In the type definition of **Env**, **Var** is a domain of environment variables and **Value** denotes a disjunctive unification of all *PFL* data types values.

A syntactic form of a variable attributed type `v T` as an argument type of a process allows a user to consider the visible variable environment in role of input memory gate of process bodies, consisting of a subset of environment variables - memory cells that are possibly shared by multiple definitions of processes in the same scope.

The processes may be applied either to control values, and computed using values accessed from the environment variables, or to data values and computed using them, updating the environment variables by this value before.

Concluding, the state is defined by the environment that internally conforms to that used in imperative and impure functional languages, but for the reasons of its binding to process bodies, the *PFL* semantics is the same as the semantics of monadic approach [6]. In *PFL*, the access and the update of environment are uniform in each scope, by processes defined in the same scope as the environment - global, local or object one.

Let us suppose a simple *PFL* process `sum` defined in a main scope, which has two environment variables `a` and `b` defined in a process type definition, as follows.

```
sum :: a Int -> b Int -> Int
sum x y = x + y
```

Suppose an application `sum 3 4` exists somewhere in an expression of a *PFL* program, such that `sum` is accessible (for example in a definition of a process in the main scope). Then the result of the application will be updating the environment variables `a` by the value 3, updating the environment variables `b` by the value 4, as an additional side effect to the evaluation of pure function. It means the value of the application will be 7.

This is so because in the first stage of the translation the definition above is transformed to the form of pure function, as follows

```
sum :: Int -> Int -> Int
sum x y = x + y
```

while each application of `sum` is transformed to the form, in which environment variable is applied to corresponding argument. For example, `sum 3 4` is transformed to `(sum (a 3) (b 4))`.

Prog	→	*CDecls IDecls Def* **main** = exp	Program
CDecls	→	*CDecl* CDecls	
	|	ε	
CDecl	→	**class** $\theta \Rightarrow \kappa\, \alpha_1 \ldots \alpha_n$ **where** $\{f_i :: \varphi_i\}_{i=1}^{m}$	Type class declaration (n ≥ 0)
IDecls	→	*IDecl* IDecls	
	|	ε	
IDecl	→	**instance** $\phi \Rightarrow \kappa\, \tau_1 \ldots \tau_n$ **where** $\{f_i = \exp_i\}_{i=1}^{m}$	Instance declaration (n ≥ 0)
Def	→	f *exp Def*	
	|	ε	
exp	→	x	Identifier
	|	()	Control value
	|	χ exp_1 ... exp_n	Type constructor
	|	κ exp_1 ... exp_n	Type class constructor
	|	exp_1 exp_n	Application
	|	$\lambda x.exp$	Function abstraction
	|	$x \Rightarrow exp$	Object application

Fig. 1 Abstract syntaxt of object-oriented *PFL*.

On the other hand, if original argument is the control value `()`, then the transformed application may be for example `(sum (a ()) (b 5))`, provided that the source form is `(sum () 5)`. Then the value of `y` will be 5 (updating `b` by 5), and the value of `x` will be the value accessed from environment variable `a` by the application `(a ())`. Provided that the value in `a` is 6, the value of `(sum (a ()) (b 5))` will be 11. If no value has been assigned to `a` before, then the value of the application is undefined.

Since the access and update instances are applied implicitly, i.e. they never occur in the process definitions, the state change strongly depends on the order in which the arguments of a process are evaluated.

Object-oriented approach is based on extension of abstract types known from pure functional languages [11]. The *PFL* oriented-oriented extensions are presented in the next example.

```
data Color = Red | Green | Blue
class Point where
    pointX :: x Int -> Int
    pointY :: y Int -> Int
    moveX :: Int -> Int
    moveY :: Int -> Int

class (Point) => CPoint where
  setColor :: c Color -> ()
  color :: c Color -> Color

instance Point where
   pointX px = px
   pointY py = py
   moveX px = px + (pointX ())
   moveX py = py + (pointY ())

instance CPoint where
   setColor pc = ()
   color pc = pc
```

There are defined two classes `Point` and `CPoint` in the example. Classes are defined as monomorphic type classes - it is possible to define only one instance declaration. The class `Point` defines two processes for direct manipulation of environment variables (coordinates) `x` and `y`. Class `CPoint` extends class `Point`. A function `setNewX` using `CPoint` objects can be defined as follows.

```
setNewX :: CPoint -> CPoint -> Int
setNewX p1 p2 = p1 => moveX (
   p2 => pointX ())
```

Arguments `p1` and `p2` represent objects. Function `setNewX` moves coordinate `x` of an object `p1` relatively to new position defined by value of coordinate `x` of an object `p2`.

3. TYPES AND TYPE ENVIRONMNETS

Type checking is one of the most important parts of a program compiling. Usually the type checking is done during compile time. It is important to identify and infer types and create type environments during early stages of type checking for the later use when transformation rules are selected and applied. PFL has defined set of types (see section 0) and set of type environments (see section 0). The result of a PFL the program translation is presented on example focusing on inferred type environments (see section 0).

$$
\begin{array}{lcll}
\bar{\tau} & \rightarrow & \nu\, T & \\
 & | & \tau & \\
 & | & () & \\
 & | & \kappa\, \tau_1 \ldots \tau_n & (n \geq 0) \\
 & | & \bar{\tau}' \rightarrow \bar{\tau} & \\
\tau & \rightarrow & \alpha & \\
 & | & \chi\, \tau_1 \ldots \tau_k & (k \geq 0,\ k = arity(\chi)) \\
 & | & \tau' \rightarrow \tau & \\
T & \rightarrow & \kappa\, \tau_1 \ldots \tau_n & (n \geq 0) \\
 & | & \chi\, \tau_1 \ldots \tau_n & (n \geq 0) \\
\rho & \rightarrow & \left\langle \kappa_1\ \tau_1^1 \ldots \tau_{r_1}^1, \ldots, \kappa_1\ \tau_1^m \ldots \tau_{r_m}^m \right\rangle \Rightarrow \bar{\tau} & (m \geq 0) \\
\sigma & \rightarrow & \forall \alpha_1 \ldots \alpha_l . \theta \Rightarrow \bar{\tau} & (l \geq 0) \\
\varphi & \rightarrow & \forall \alpha_1 \ldots \alpha_l . \phi \Rightarrow \bar{\tau} & (l \geq 0) \\
\theta & \rightarrow & \left\langle \kappa_1\ \alpha_1^1 \ldots \alpha_{r_1}^1, \ldots, \kappa_1\ \alpha_1^m \ldots \alpha_{r_m}^m \right\rangle & (m \geq 0) \\
\phi & \rightarrow & \left\langle \kappa_1\ \tau_1^1 \ldots \tau_{r_1}^1, \ldots, \kappa_1\ \tau_1^m \ldots \tau_{r_m}^m \right\rangle & (m \geq 0) \\
\gamma & \rightarrow & \left\langle \gamma_1, \ldots, \gamma_m, \nu_1 : \widetilde{T}_1 \rightarrow T_1, \ldots, \nu_1 : \widetilde{T}_l \rightarrow T_l \right\rangle & (m \geq 0, l \geq 0) \\
\widetilde{T} & \rightarrow & T & \\
 & | & () &
\end{array}
$$

Fig. 2 Syntax of types.

3.1. *PFL* types

The abstract syntax of *PFL* program is defined on the Fig. 1. Basic type definitions are denoted with symbols $\theta, \kappa, \alpha, \varphi, \chi$ and ϕ. The abstract syntax contains rules:

- Abstract types declaration rules
  - Type class declaration
  - Instance declaration
- Object manipulation rules
  - Object creation – application of a type class constructor
  - Object application – application of a process or function in the context of object
- Basic processing of expression rules

In the phase of source *PFL* program translation the rules of static semantics are used. Types and type environments are important parts of the translation rules. The *PFL* type system is inspired with the type system defined by Wadler and Blott [12]. Types used in static semantic of object-oriented *PFL* are as follows.

α type variable
κ type class name
χ type constructor
() unit type
T concrete type (algebraic type, class type)
$\widetilde{T}$ argument type
ν environment variable
$\bar{\tau}$ basic type
τ pure basic type
θ type class context
ϕ instance context
ρ overloaded type
φ extended polymorphic type
σ polymorphic type
γ variable environment

Environment variable ν is used only in type expressions. It is main reason of comprising environment variable in a set of types. Syntax of types is shown on the Fig. 2. Application of type rules is explained on next function type definition.

$$f :: (Ord\ a) \Rightarrow a \rightarrow [a] \rightarrow Boolean$$

This type definition is expressed with types defined on the Fig. 2 as follows.

$$f :: \forall \alpha . (Ord\ \alpha) \Rightarrow \alpha \rightarrow List\ \alpha \rightarrow Boolean$$

It is polymorphic type definition. It is possible to write it in next form.

$$f :: \sigma$$

Environment name	Abbreviation	Type
Type Variable Environment	*TVE*	$\{\alpha\}$
Type Constructor Environment	*TCE*	$\{\chi : k\}$
Type Class Environment	*CE*	$\{\kappa : \forall \alpha_1 \ldots \alpha_n . \theta \Rightarrow PFE\}$
Class Variable Environment	*CVE*	$\{\kappa : \gamma\}$
Local Class Variable Environment	*LCVE*	$\{\nu : \widetilde{T} \rightarrow T\}$
Instance Environment	*IE*	$\{\mathbf{dpf} : \forall \alpha_1 \ldots \alpha_l . \phi \Rightarrow \kappa\ \tau_1 \ldots \tau_n\}$
Identifier Environment	*IDE*	$\{\mathbf{var} : \varphi\}$
Selection of Position Environment	*SPE*	$\{\mathbf{sel} : i\}$
Process and Function Environment	*PFE*	$\{\mathbf{pf} : \varphi\}$
Context of Process and Function Environment	*CPFE*	$\{\mathbf{dvar} : \kappa\ \tau_1 \ldots \tau_n\}$
Current Instance Type	*CIT*	$\{\mathbf{dvar} : \kappa\ \tau_1 \ldots \tau_n\}$
Context of Instance Environment	*CIE*	$\{\mathbf{dvar} : \kappa\ \tau_1 \ldots \tau_n\}$
Object Environment	*OE*	$\{var : \langle \mathbf{dvar}, \mathbf{env} \rangle\}$
Environment of Variable Environment	*EVE*	$\{name : \mathbf{exp}\}$
Names Environment	*NE*	$\{name\}$
Types Environment	*TE*	$\{name\}$
Environment	*E*	(*TVE*,*TCE*,*CE*,*CVE*,*IE*,*IDE*,*SPE*,*CPFE*,*PFE*)
Declarations Environment	*DE*	(*CE*,*IE*,*IDE*,*CVE*,*SPE*)

Fig. 3 Type Environments.

In this form of type definition the polymorphic type $\sigma = \forall \alpha . \theta \Rightarrow \tau$ where context $\theta = Ord\ \alpha$. Name *Ord* is a class name and type variable α is a type class parameter. Type τ consists of two type constructors:

- *List* – type constructor of arity 1.
- *Boolean* – type constructor of arity 0.

Also, we could write polymorphic type in the form $\sigma = \forall \alpha . \rho$. In this example type $\sigma = \varphi$. Main difference between them is in context type. Context of extended polymorphic type φ comprises types in the form $\kappa\ \tau_1 \ldots \tau_n$. It means that polymorphic type σ is more general than type φ due to comprising only type environment ($\sigma \subseteq \varphi$).

3.2. *PFL* type environments

In the phase of *PFL* program transformation the type environments are created and applied. These type environments comprise sufficient information to verifying validity of type variables, type constructors, class names and variable names in types and expressions. Type environments are shown on the Fig. 3. Names written in bold represent names produced in the compile time. Type environments are divided into next categories.

- Simple environment – does not comprise other environment.
- Compound environment – consists of other environments.

Environment *ENV* is defined as a mapping of *name* into information (*info*) in next form.

$$\{name : info\}$$

This form describes environments except of *TVE*, *NE* and *TE* where only *name* without mapping is defined.
We have defined operators within type environments. They are applied as a part of semantic rules. These operators are divided into two categories.

- Operator for selection of environment from compound environment - *of*.
- Operators for combination of environments - $\oplus, \vec{\oplus}, \oplus_{ENV}, \vec{\oplus}_{ENV}$ and $\overset{\mapsto}{\oplus}_{ENV}$.

Semantics of operator *of* is possible to explain on next application *IDE of E*. Operator *of* selects environment *IDE* from environment *E*. More complex semantics is defined for operators of environments combination. In first case the operators over simple environments are considered - $\oplus, \vec{\oplus}$. They are defined as follows.

$$
\begin{aligned}
\alpha \cup \kappa\ \tau_1 \ldots \tau_n &= \kappa\ \tau_1 \ldots \tau_n \\
\alpha \cup \chi\ \tau_1 \ldots \tau_n &= \chi\ \tau_1 \ldots \tau_n \\
\alpha_1 \cup \alpha_2 &= \alpha_3, \text{where } \alpha_3 = \alpha_1 \vee \alpha_2 \\
\alpha \cup () &= () \\
\bar{\tau}_1^1 \to \bar{\tau}_2^1 \cup \bar{\tau}_1^2 \to \bar{\tau}_2^2 &= (\bar{\tau}_1^1 \cup \bar{\tau}_1^2) \to (\bar{\tau}_2^1 \cup \bar{\tau}_2^2) \\
\kappa_1 \tau_1^1 \ldots \tau_n^1 \cup \kappa_2 \tau_1^2 \ldots \tau_n^2 &= \kappa_1 (\tau_1^1 \cup \tau_1^2) \ldots (\tau_n^1 \cup \tau_n^2), \text{if } \kappa_1 = \kappa_2 \\
\chi_1 \tau_1^1 \ldots \tau_n^1 \cup \chi_2 \tau_1^2 \ldots \tau_n^2 &= \chi_1 (\tau_1^1 \cup \tau_1^2) \ldots (\tau_n^1 \cup \tau_n^2), \text{if } \chi_1 = \chi_2 \\
\nu T_1 \cup \nu T_2 &= T_1 \cup T_2
\end{aligned}
$$

Fig. 4 Types unification rules.

$$\oplus :: simpleEnv \times simpleEnv \to simpleEnv$$
$$\vec{\oplus} :: simpleEnv \times simpleEnv \to simpleEnv$$

Type *simpleEnv* represents simple environment. Combination of two environments is written in next form $IDE_1 \oplus IDE_2$. Operator $\oplus$ combines items of environment IDE_1 and IDE_2. The semantics is as follows.

$$(ENV_1 \oplus ENV_2)var = \begin{cases} ENV_1 var, \text{if } var \in dom(ENV_1) \\ \qquad \wedge\ var \notin dom(ENV_2) \\ ENV_2 var, \text{if } var \in dom(ENV_2) \\ \qquad \wedge\ var \notin dom(ENV_1) \end{cases}$$

$$(ENV_1 \vec{\oplus} ENV_2)var = \begin{cases} ENV_1 var, \text{if } var \in dom(ENV_1) \\ \qquad \wedge\ var \notin dom(ENV_2) \\ ENV_2 var, \text{if } var \in dom(ENV_2) \end{cases}$$

Expression $dom(ENV_1)$ represents domain of environment ENV_1. Operators $\oplus_{ENV}, \vec{\oplus}_{ENV}$ and $\overset{\mapsto}{\oplus}_{ENV}$ combine simple environment *simpleEnv* and compound environment *compEnv* as follows.

$$\oplus_{ENV} :: compEnv \times (simpleEnv \vee compEnv) \to compEnv$$
$$\vec{\oplus}_{ENV} :: compEnv \times simpleEnv \to compEnv$$
$$\overset{\mapsto}{\oplus}_{ENV} :: compEnv \times simpleEnv \to compEnv$$

The semantics of the operators is defined as follows.

$$(ENV_1 \oplus_{ENV} ENV_2)var = \begin{cases} E \oplus_{CE} (CE\ of\ ENV') \oplus_{IE} (IE\ of\ ENV') \\ \quad \oplus_{IDE} (IDE\ of\ ENV') \oplus_{CVE} (CVE\ of\ ENV') \\ \quad \oplus_{SPE} (SPE\ of\ ENV'), \text{if } ENV' = DE \\ E[((ENV\ of\ E) \oplus ENV')/ENV], otherwise \end{cases}$$

$$E \vec{\oplus}_{ENV} ENV' = E[((ENV\ of\ E) \vec{\oplus} ENV')/ENV]$$
$$E \overset{\mapsto}{\oplus}_{ENV} ENV' = E[ENV'/ENV]$$

The expression $E[((ENV\ of\ E) \oplus ENV')/ENV]$ define substitution of environment ENV as a combination of environment ENV and ENV'.
Semantics these operators are explained on short examples as follows.

$$E \oplus_{IDE} IDE' = (TVE, \ldots, IE, IDE \oplus IDE', SPE, \cdots)$$

The definition of compound environment E is shown on the Fig. 3. As a result the environments IDE and IDE' are applied using operator $\oplus$ with semantics defined above. It is similar in the case of operator $\vec{\oplus}_{ENV}$.

$$E \vec{\oplus}_{IDE} IDE' = (TVE, \ldots, IE, IDE \vec{\oplus} IDE', SPE, \cdots)$$

The items of environment IDE' with the same name like in environment IDE are favoured. The semantics of operator $\overset{\mapsto}{\oplus}_{ENV}$ is a little bit different.

$$E \overset{\mapsto}{\oplus}_{IDE} IDE' = (TVE, \ldots, IE, IDE', SPE, \cdots)$$

In the compound environment E the environment IDE is replaced with environment IDE'.

There are three implicit conditions defined over the environments.

- The same variable *var* cannot be comprised in two environments ENV_1 and ENV_2 if the operator $\oplus$ is applied.

$$dom(ENV_1) \cap dom(ENV_2) = \varnothing$$

- The variable *var* must be comprised in the environment *ENV* if the expression *ENV var* is occurred.

$$var \in dom(ENV)$$

- Only one instance of the same type has to be declared for type class. If the expression $IE_1 \oplus IE_2$ is occurred, next condition is true.

$$\forall \kappa_i\ \tau_1^i \ldots \tau_{r_i}^i \in IE_1 \wedge \forall \kappa_j\ \tau_1^j \ldots \tau_{r_j}^j \in IE_2 :$$
$$\kappa_i \neq \kappa_j \vee \tau_1^i \bigcup \tau_1^j = \bot \vee \ldots \vee \tau_{r_i}^i \bigcup \tau_{r_j}^j = \bot$$

The operator $\bigcup$ unifies two types (see Fig. 4). For other cases the result of unification is equal $\bot$.

Example

Now we will consider example from section 0. The type environments after the *PFL* program translation are inferred as follows.

$TCE_0 = \{\text{Color} : 0\}$
$CE_0 = \{$
  Point : $\langle\rangle \Rightarrow \{$
    pointX : x Int $\rightarrow$ Int
    pointY : y Int $\rightarrow$ Int
    moveX : Int $\rightarrow$ Int
    moveY : Int $\rightarrow$ Int $\}$
  CPoint : $\langle$Point$\rangle \Rightarrow \{$
    setColor : c Color $\rightarrow$ ()
    color : c Color $\rightarrow$ Color $\}$
$\}$
$IE_0 = \{$**selPointFromCPoint** : $\langle$CPoint$\rangle \Rightarrow$ Point$\}$
$IDE = \{$
  pointX : $\langle$Point$\rangle \Rightarrow$ x Int $\rightarrow$ Int
  pointY : $\langle$Point$\rangle \Rightarrow$ y Int $\rightarrow$ Int
  moveX : $\langle$Point$\rangle \Rightarrow$ Int $\rightarrow$ Int
  moveY : $\langle$Point$\rangle \Rightarrow$ Int $\rightarrow$ Int $\}$
  setColor : $\langle$CPoint$\rangle \Rightarrow$ c Color $\rightarrow$ ()
  color : $\langle$CPoint$\rangle \Rightarrow$ c Color $\rightarrow$ Color $\}$
$\}$
$CVE = \{$
  Point : $\langle$x, y$\rangle$
  CPoint : $\langle\langle$x, y$\rangle$, c$\rangle$
$\}$
$SPE = \{$**selPointFromCPoint** : 0$\}$

These environments have been obtained by application of static semantic rules. These environments are helpful during program translation.

CONCLUSION

In this paper the essence of object-oriented *PFL* - experimental functional language has been presented. The subject of our current research is to exploit the process functional paradigm for integrating functional, imperative and aspect-oriented methodology, using simple, uniform and still practical basis, appropriate for source-to-source transformations, reasoning on the behavior and verification experiments. Currently we have implemented the compiler from *PFL* to both Java and Haskell languages. Using *PFL*, the level of abstraction has increased, preserving all abilities of imperative languages, including the visibility of all environments, providing a single tool for affecting the state in the form of the application of processes.

As it was mentioned the type and type environments have been inspired with Haskell. In the paper the extensions of type environments and manipulations over them are presented. Types and type environments have been incorporated into *PFL* compiler. It is also possible to use presented design of types and type environments in similar problems in other programming languages with minority changes.

We are trying to give programmers simple and understandable tool integrating functional language and imperative style of programming. It is an experimental language not only focused on functional programming but also it is a platform for object-oriented and aspect-oriented programming. One of our aims is to extend *PFL* to be a parallel and aspect-oriented language. Especially aspect-oriented paradigm gives us the possibility to investigate the area of software evolution based on aspect-oriented approach. Research in extending *PFL* to be parallel [1] is now in the phase of design and implementation where the work was also inspired by the work in the area of distributed functional languages such as DClean [3].

REFERENCES

[1] Běhálek, M., Šaloun, P.: Paralelization of Process Functional Language. Proceedings of 7-th International Scientific Conf. on Electronic Computers and Informatics ECI 2006, Košice-Herľany, Slovakia, September 20-22, 2006, pp. 168-173, ISBN 80-8073-598-0

[2] Bobrow, D., G., Kahn, K., Kiczales, G., Masiner, L., Stefik, M., and Zdybel, F.: CommonLoops: Merging Lisp and object-oriented programming. Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1986, pp. 17-29.

[3] Horváth, Z., Hernyák, Z, Zsók, V.: Implementing Distributed Skeletons using D-Clean and D-Box. 17th International Workshop on Implementation and Application of Functional Languages IFL 2005 Dublin, Ireland, 2005, pp. 1-16.

[4] Hughes, J., and Sparud, J.: Haskell++: An Object-Oriented Extension of Haskell. In Proceedings of the Haskell Workshop, 1995.

[5] Kollár, J.: Process Functional Programming, Proc. ISM'99, Rožňov pod Radhoštěm, Czech Republic, April 27-29, 1999, 41-48.

[6] Kollár, J., Porubän, J., Václavík, P.: From Eager PFL to Lazy Haskell. In: Computing and Informatics, 2006, pp. 61--80, ISSN 1335-9150.

[7] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J.: The Objective Caml system, documentation and user's manual - release 3.08. INRIA, July 2004.
[8] Lüfer, K.: A framework for higher-order functions in C++. In Proc. Conf. Object-Oriented Technologies (COOTS), Monterey, CA, 1995, USENIX.
[9] Odersky, M., Wadler, P.: Pizza into Java: Translating Theory into Practice. Proc. 1997 ACM Symp. on Principles of Prog. Langs. (POPL '97), Paris, France, pp. 146-159.
[10] Plasmeijer, R., Eekelen, M.: Concurrent Clean Language Report version 2.1. University of Nijmegen, November 2002.
[11] Václavík, P., Kollár, J., Porubän, J.: Object-oriented Programming with Functional Language. 8'th International Conference ISIM'05, Hradec nad Moravicí, Czech Republic, April 19 - 20, 2005, pp. 167-174, ISBN 80-86840-09-3.
[12] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: ACM Symposium on Principles of Programming Languages, Austin, Texas, 1989, pp. 60-76.

BIOGRAPHIES

Peter Václavík was born in 1977. He received his MSc. summa cum laude in 2000 and his PhD. in Computing Science in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was involved in the research projects dealing with implementation of functional programming languages and parallel programming. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming and program profiling systems.

Jaroslav Porubän was born in 1977. He received his MSc. summa cum laude in 2000 and his PhD. in Computing Science in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was involved in the research projects dealing with implementation of functional programming languages and parallel programming. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming and program profiling systems.