Compiling the Process Functional Programs

Peter Václavík, Ján Kollár, Jaroslav Porubän, Miroslav Vidiščák

Department of Computers and Informatics, Technical University of Košice, Slovakia
Peter.Vaclavik@tuke.sk, Jan.Kollar@tuke.sk, Jaroslav.Poruban@tuke.sk, Miroslav.Vidiscak@tuke.sk

Abstract: This paper briefly describes process functional language compiler architecture. Process functional language is an experimental functional language with imperative features like memory cell variables. The main parts of the process functional language compiler are presented with short description and examples. Current implementation of PFL compiler comprises a algebraic types, primitive functions, operators, pattern matching, variable environment, abstract data types and process functional program profilation.

Keywords: functional programming, translation, process functional paradigm, program transformations1

1 Introduction

The Process functional language *PFL* was developed at Department of Computers and Informatics during the past few years [5], [7]. The main idea of the language design is to decrease a gap between imperative and functional languages with respect to positive characteristics of both programming paradigms. New constructions were defined during the language design process: environment variable, loop comprehension and spatial types. They differ *PFL* from other special functional languages.

The programming approach is functional without assignments and other state change statements. All imperative programs can be transformed to process functional programs [6].

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

Main constructs in process functional language are unit type, environment variables and processes.

Unit type - is marked by symbol (). The same symbol is used for the only one unit type value - control value (). Semantics of this control value is different than semantics of a data value. Control value represents control flow in program.

Environment variable - is a memory cell like in an imperative language. The main difference between imperative program variable, represented as a memory cell, and an environment variable in *PFL* are in the concept of variable application with respect to accessing and updating the variable. In *PFL*, the application is

- static (visible to programmer) binding of environment variable - memory cell to process argument defined by process signature (process type definition),
- implicit (invisible to programmer) environment variables application - operation to process argument (type of operation is depended on argument type).

The programmer does not affect the state directly using special operations but it is done implicitly by process application. The environment variable *v* is defined as follows:

$$v :: \tilde{T} \to T, \text{ where } \tilde{T} \to T \mid ()$$

The environment variable access operation is defined as

$$v :: () \to T$$

$$v\ () = \text{v}$$

The v is a value of type *T* stored in environment variable *v*. The environment variable update operation is defined as

$$v :: T \to T$$

$$v\ x = \text{x}$$

Next assertions about environment variable are true:

- Value type of environment variable application is always data value, not a spatial or control one.
- Environment variable is accessed or updated indirectly by process application.
- Application of control value on environment variable is equivalent to variable access operation in imperative languages.

- Application of data value on environment variable is equivalent to assignment of value to variable in imperative languages.

More detailed information about environment variable can be found in papers [5] [8].

Process - definition differs from function definition in pure functional language only by its type definition. Function is called process in process functional language if its type signature comprises one of the following forms.

- Process signature comprises unit type as its argument or value type, for example:

  $$f :: () \to T \to ()$$

  where *f* is a process with two arguments (type of first one is unit type, the second one is a data type *T*) and return value is type of unit type.

- Process type definition comprises at least one argument type in a form *v T*, for example:

  $$f :: v\ T \to a \to T$$

It is possible to combine both cases in type definition of a process.

$$f :: v\ T \to () \to ()$$

Next example presents definition of *PFL* process *pf*. Process type definition comprises first argument in the form *a T*, where *a* is name of an environment variable and *T* is its type.

$$pf :: a\ \ T \to T \to T$$

$$pf \qquad x\ \ y = \exp$$

The value of environment variable *a* can be update by application of the process *pf* to values *val1* and *val2*. This case is shown in the Figure 1 (a). In the next case, an environment variable is accessed as it is shown in the Figure 1 (b). Here, first argument is a unit value and the variable environment value is accessed.

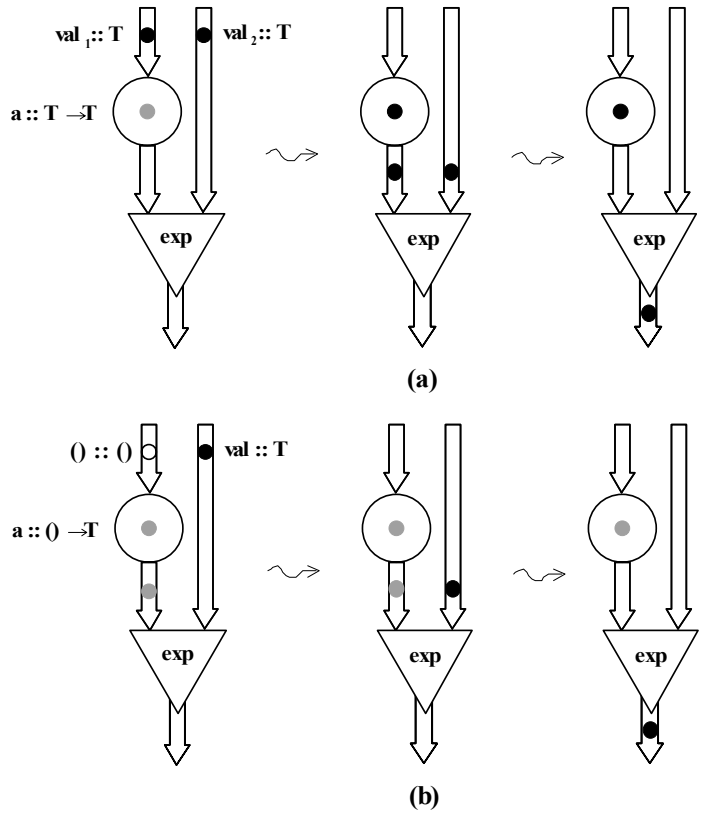

Figure 1
Instances of environment variable access and update.

2 Compiler architecture

The compiler architecture is inspired by the Haskell and Gopher compiler [3] [4]. The modular compiler architecture is shown in the Figure 2. This architecture is defined by phases of the process functional program compiling. Actual version of process functional compiler has already implemented these features:

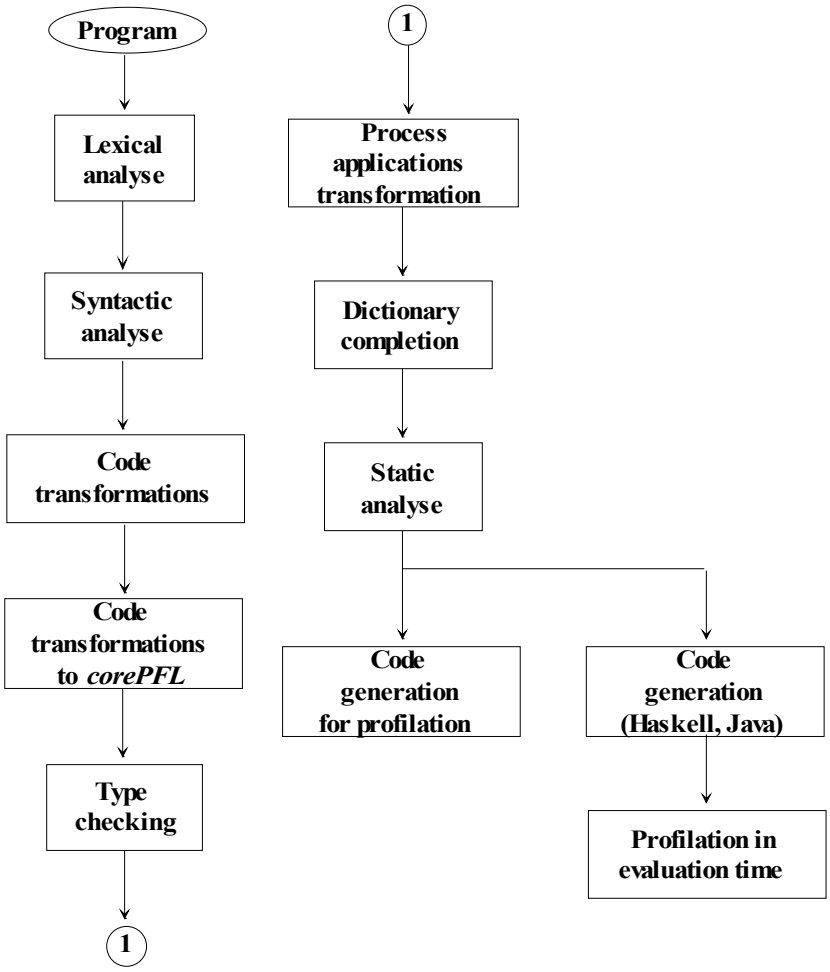

Figure 2
PFL compiler architecture.

- **Algebraic types**. Monomorphic and polymorphic algebraic type definitions are supported. Syntax and semantic is the same like in Haskell.

  data Tree a b = Leaf a | Node b (Tree a b) (Tree a b)

- **Primitive functions**. It is possible to define the primitive function explicitly. Primitive functions extend the usability of process functional language and make *PFL* general-purpose language for solving wide area of problems. Primitive function type definition can be placed in the source code of *PFL* program and implementation can be done in other platform

specific programming language, depending on the programmer requirements.

```
primitive plusInteger :: Integer -> Integer -> Integer
```

- **Operator** definition is the same like in other functional languages.
- **Global function definition**.
- **Pattern matching**.
- **Global variable environment** is defined by process definitions at a global level. If the same environment variable name is used in different processes, the environment variable is shared between these processes.
- **Abstract data type** definition with **object-oriented approach.** It is possible to define abstract types (type classes and its instances) like in Haskell. In contrast to Haskell multiparameter type class definition are supported. The object variable environment bounded to concrete type class is created from process type definitions defined in the class. More information about abstract types see [11].
- Process functional **program profilation** support is implemented within the compiler for identifying execution bottlenecks of the program - parts of a program where much of time and space is used. Syntactic construction *label* provides feedback to the programmer about resource utilization, relating information gathered from program runtime back to the source code in well-defined manner. More information about *PFL* program profilation can be found in [10].

These constructions are fully supported with respect to process functional paradigm definition. Next subsections describe basic parts of *PFL* compiler.

2.1 Lexical Analysis

In the lexical analysis phase, lexical units of process functional program are identified (keywords, identifiers, numbers, etc.). We can divide lexical units into two groups:

- Basic lexical units - basic lexical units defined by core *PFL*.
- Extensions of core PFL for profiling and object oriented *PFL* constructions (abstract type - **class**, **instance** and ⇒; etc.)

2.2 Syntactic Analysis

This phase covers the source program syntax checking. Furthermore, the program translation tree (see Figure 3) is generated from the input source program. Output from syntax analysis is:

- list of type constructors,
- list of constructors,
- list of type classes,
- list of instances,
- syntactic tree for all expressions within the program.

2.3 Code Transformations

This phase solves next basic code transformation:

- expression transformation according to operator associativity,
- expression transformation consideting operator priority,
- abstract types and object usage transformations.

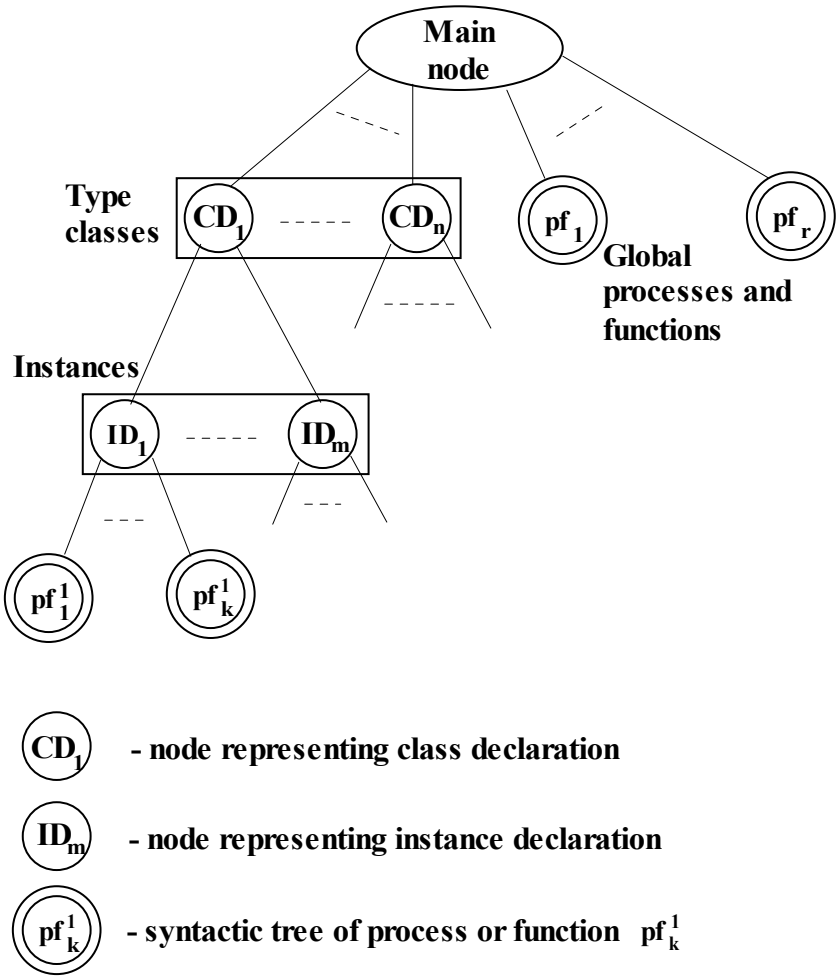

Figure 3
Instances of environment variable access and update.

2.4 Code Transformation to CorePFL

In this phase, all *PFL* syntactic constructions are transformed to *corePFL* constructions. The *corePFL* is the minimal subset of *PFL* constructions for expressing all other *PFL* constructions. Abstract syntax of the *corePFL* is presented on Figure 4. It comes out the Haskell natural semantics [9]. There are already defined transformations schemes from *PFL* to *corePFL*. As an example of transformation from *PFL* to *corePFL* is **if-then-else** expression transformation provided.

$$T_{if}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = \textbf{case } e_1 \textbf{ of } \{True \to e_2; False \to e_3\}$$

$$
\begin{array}{lll}
P & ::= Def \ \textbf{main} = e & \\
Def & ::= f = e \ Def & \\
 & \mid \ \varepsilon & \\
e & ::= x & \textbf{Variable} \\
 & \mid f & \textbf{Function} \\
 & \mid e_1 \oplus e_2 & \textbf{Primitive} \\
 & \mid y\,() & \textbf{Access} \\
 & \mid y\ e & \textbf{Update} \\
 & \mid C\ e_1 \ldots e_n & \textbf{Constructor} \\
 & \mid e_1\ \ e_2 & \textbf{Application} \\
 & \mid \textbf{case } e \textbf{ of } \{C_i\ \ x_1 \ldots x_{i_m} \to e_i\}_{i=1}^{n} & \textbf{Case} \\
 & \mid \lambda x.e & \textbf{Primitive}
\end{array}
$$

Figure 4
Abstract syntax of *corePFL*.

2.5 Type Checking, Context Checking

This is a one of key phases of *PFL* program translation. A unification type rules are applied on every expression, checking for type correctness of a program. Type checking is divided into few steps:

- Process or function signature derivation - using unification rules the signature for the process or function is derived from syntactic tree.
- Explicit signature definition - derived signature must by unified with user one.
- Specific checking rules - for some construction the specific type checking rules are defined. For example, the abstract types need to be checked for superclass declaration existence, type class hierarchy, instance declaration overlaying and instance declaration of superclass overlaying).

In context checking compiler looks for:

- If all functions/processes specified with type definitions have their definitions.

- If the context of a variable can be derived.
- If the context of an environment variable can be derived.
- If all algebraic types have their definitions.

2.6 Process Application Transformation

Process applications are transformed to a form, in which the environment variable update or access operation are implicitly embedded in expression. On the Figure 5 the transformation of process *p* application is shown. In the first case (a), process *p* is applied to control value (), which is similar to imperative access operation. Second case (b) presents application of a process *p* on value of *Int* data type, which is similar to imperative assignment operation.

2.7 Dictionary Completion

In this phase the rules of accessing the process or function definition from available dictionary (the result of instance declaration translation) are included to node of process or function identifier. This rules comes out the Haskell language [2]. Accessible scopes of dictionaries depends on a scope of process or function definition:

- Type class - context of instance declaration, actual instance declaration (and superclasses dictionary too) and context of function or process type definition.
- Global process or function - the context of function or process type definition.

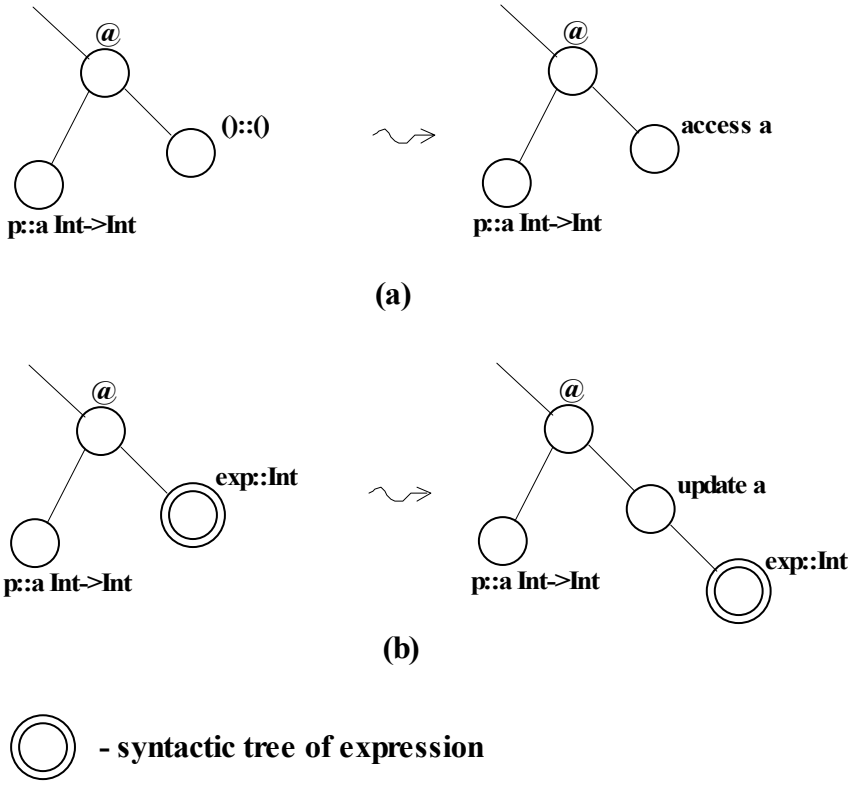

Figure 5
Update and access operation embedding.

2.8 Static Analysis

This step is essential for program optimization. Static analysis is used for gathering information from source program about program runtime behavior without need for program execution. Static analysis is done in compiler by control flow analysis and side effects analysis. Information acquired from the static analysis are used in code generator.

2.9 Code Generation

The last phase of a program compiling is code generation. Nowadays it is possible to generate the code to Haskell or Java. However, it is possible to expand the collection of target languages due to modular structure of a compiler. Implementation of generator to C language with MPI support is now in progress.

2.10 Profilation

This phase consists of two steps. First one is *PFL* program code generation with profiling support. The second step is gathering the information about resource utilization during the program runtime. If the compiler is adjusted to program profiling mode then the special code generator is selected. Profiling generator marks generated code for profiling.

Implemented process functional program profiler nowadays supports five types of profiles:

- frequency count profile,
- time profile,
- heap profile,
- maximum requirements heap profile,
- variable access/update profile.

Program profile is created during the execution using the sampling method. Execution is interrupted in specified time intervals (predefined value is 10 milliseconds) and information about used resources are collected and attributed to the current labeled center. Program profiling increases execution time approximately from 5 to 10% depending on the concrete program and labeling.

Conclusions

This paper is not concerned on particular problem solving. It brings the general overview of *PFL* programming language and its features. Main program compilation phases are described in the paper. Presented compiler architecture is already implemented. However, the presented architecture and *PFL* constructions may not be final one. For example, the optimization phase is absent in the compiler. Our current work is aimed to solving problems of parallel execution of process functional programs with respect to underlying architecture and implementaition of visual tools for modelling program behaviour. Our future work will be oriented to integration of aspect-oriented programming [1] and process functional programming extending the compiler architecture.

References

[1] Avdicausevic, E., Lenic, M., Mernik, M., Zumer, V.: *AspectCOOL: An experiment in design and implementation of aspect-oriented language*, ACM SIGPLAN not., December 2001, Vol. 36, No.12, pp. 84-94

[2] Hall, C., Hammond, K., Jones, S. P., Wadler, P.: *Type classes in Haskell*, European Symposium On Programming, LNCS 788, Springer Verlag, pp. 241-256, 1994

[3] Jones, M. P.: *The implementation of the Gopher functional programming system*, Research Report YALEU/DCS/RR-1030, May 1994

[4] Jones, S. L. P., Hughes, J.: *Report on the Programming Language Haskell 98 A Non-strict, Purely Functional Language*, pp. 163, February 1999

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

[6] Kollár J.: *PFL Expressions for Imperative Control Structures*, Proc. of Computer Engineering and Informatics Scientific conference with International participation, Oct. 14-15, Herľany, Slovakia, ISBN 80-88922-05-4, 1999, pp.23-28

[7] Kollár J.: *Object Modelling using Process Functional Paradigm*, Proc. 34th Spring International Conference MOSIS 2000 - ISM 2000 Information Systems Modelling, Rožnov pod Radhoštěm, Czech Republic, Máj 2-4, 2000, ACTA MOSIS No. 80, ISBN 80-85988-45-3, pp. 203-208

[8] Kollár J., Porubän J., Václavík P., Vidiščak M.: *Lazy State Evaluation of Process Functional Programs*, Proceedings of the Conference "Information Systems Modelling" ISM'02, Rožnov pod Radhoštěm, ISBN 80-85988-70-4, 2002, pp. 165-172

[9] Launchbury, J.: *A natural semantics for lazy evaluation*, In Proceedings of the 20th ACM Conference on Principles of Programming Languages, pp. 144-154, 1993

[10] Porubän J.: *Time and space profiling for process functional language*, Proceeding of the 7th Scientific Conference with International Participation Engineering of Modern Electric Systems '03, Felix Spa, Oradea, 1223-2106, 2003, pp. 167-172

[11] Václavík P.: *The Fundamentals of a Process Functional Abstract Type Translation*, Proceeding of the 7th Scientific Conference with International Participation: Engineering of Modern Electric '03 Systems, Felix Spa, Oradea, ISSN-1223-2106, 2003, pp. 193-198