Embedded Process Functional Language

Marek Běhálek[1] and Petr Šaloun[2]

[1] Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VŠB Technical University of Ostrava, 17. listopadu 15, Ostrava, Czech Republic
`marek.behalek@vsb.cz`

[2] Department of Informatics and Computers, Faculty of Science, University of Ostrava, 30. dubna 22, Ostrava, Czech Republic
`petr.saloun@osu.cz`

Abstract. Embedded systems represent an important area of computer engineering. Demands on embedded applications are increasing. To address these issues, different agile methodologies are used in traditional desktop applications today. These agile methodologies often try to eliminate development risks in early design phases. Possible solution is to create a working model or a prototype of critical system parts. Then we can use this prototype in negotiation with customer and also to prove technological aspects of our solution. From this perspective functional languages are very attractive. They have excellent abstraction mechanism and they can be used as a tool producing a kind of executable design. In this paper we present our work on a domain specific functional language targeted to embedded systems — *Embedded process functional language* (e-$\mathcal{PFL}$). Created language works on a high level of abstraction and it uses other technologies (even other functional languages) created for embedded systems development on lower levels. It can be used like a modeling or a prototyping language in early development phases.

1 Introduction

Embedded systems represent an important area of computer engineering. Most of these systems are programmed in low level languages due to strict performance or memory constrains. On the other hand, demands on embedded applications are increasing. For example we want to decrease time to market, improve maintenance or make development process cheaper.

Different approaches are used to solve such problems. For example different agile methodologies [1] are more and more popular in the area of traditional desktop applications today. These agile methodologies try to eliminate development risks. Development risks are mainly related to: *business risks* (confusion in communication with customer, created product cannot be used in practice) and *technological risks* (inability to use developed application in practice due to technological issues). Possible solution that eliminates these risks is to develop working model or prototype of critical system parts in early development phases.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2353

We can use this prototype for communication with customer and also to prove technological aspects of the solution.

To address these issues we need a better tool (or tools). From this point of view, functional languages are very attractive. They have several interesting properties [2]. Among others they have excellent abstraction mechanism (represented by functions composition and high-order functions) and that is why they can be used as a tool producing a kind of executable design. To conclude functional languages definitely have a place in specification, prototyping and simulation in early design phases.

In our work we are developing domain specific functional language targeted to embedded systems development. Developed language can be used like a modeling or a prototyping language in early development phases. Created language is called *Embedded process functional language* (e-$\mathcal{PFL}$).

2 Related Works

There are other tools able to model embedded systems. For example we can use *Unified Modeling Language* – UML. Or we can use simulation tools like *Simulink*[3]. We can address business risks using such tools. We are able to create concrete model and use it during negotiations with customer. On the other hand we may not be able to solve technological issues.

Embedded systems differ in many ways from common desktop applications [3]. For example a development platform is separated from a target platform, debugging is possible only with emulator, or there is no operating system present. Really try some application on a real device can be the only possibility to eliminate technological risks. Create language is still programming language. It can be straightforwardly transformed into a target code and used on concrete embedded systems.

Usage of functional paradigm of programming for development of embedded applications was suggested in [4]. Also there exist several different functional languages for implementation of embedded systems. Those languages cover wide range of abstraction levels (from hardware description to high-level application logic).

Erlang [5] is probably the most widely known language in this area. It has its origin in Ericsson, so unlike most declarative languages it did not come from academic development. It is a combination of logic and functional programming. Typical Erlang program consists of many light-weighted processes communicating trough asynchronous messages. It is primarily used in heavily concurrent and distributed applications. Other programming languages that can be used in this area are in fact languages designed mainly for reactive systems like: *Concurrent Haskell* [6] or *Eden* [7]. Other examples of languages for embedded systems development are: *Embedded Gofer* [4] (strongly typed purely functional language), *Lava* (Haskell library with ability to generate VHDL code) or *Lustre*

[3] Description available at: `http://www.mathworks.com/products/simulink/`

(synchronous data-flow language used for reactive systems and hardware description). *Hume* [8] represents different approach. Unlike presented languages (they often come from some general purpose language) it was specially developed for (especially real-time) embedded systems implementation. It tries to address performance issues, time and space constrains and controllability. The compiler can calculate for instance how much heap and stack each part of program will ever require at most.

Also different case studies comparing usage of functional languages for embedded systems development with traditional approaches were performed [9, 10].

3 Coordination Layer

Embedded systems are often described as a set of communicating functional units, no matter if multiple processing units are present on target machine or not [4]. Also our model is a set of communicating devices on the highest level.

Most of functional languages used for implementation of embedded applications take two-level approach to language design. Purely functional expression layer is often embedded into a coordination layer. Coordination layer describes communicating processes (or communicating functional units in context of embedded systems). Presented languages are often extended with side-effecting constructs to address issues on coordination layer. These constructs often enable creation of functional units and maintain their synchronization. There are two extremes. First can be represented by language *Embedded Gofer* [4]. It uses monads to encapsulate processes and language is extended with message passing primitives to maintain their communication. Such side-effecting constructs make reasoning about program properties hard or impossible. Another extreme is *Hume* [8]. Coordination layer in Hume must be strictly defined on a static level inside a source code. Communication is then implicit and we are able to compute necessary system properties at a compile time.

When creating new language, we must decide about language aspects on coordination layer. We use dynamic coordination mechanism in e-$\mathcal{PFL}$. There are language constructions that allow functional units creation and define functional units' connections. On the other hand, static model on coordination layer have certain advantages.

Primary purpose of e-$\mathcal{PFL}$ is to create working model of an embedded system in early design phases. We want to be able to use created language on real devices to eliminate technological risks. In our approach, we want to use other technologies even other functional languages on lower levels. Static model of the coordination simplifies usage of other technologies on lower levels.

As a compromise we use following model.

– Coordination layer is dynamic on language level in EPFL. This feature simplifies embedded systems modeling.
– Then we use partial evaluation and concrete system *configuration* is produced as a result of computation. This configuration represents static model

of future system on coordination layer. Created configuration is stored in XML file.

This approach has several advantages. For example to eliminate technological risks we can produce several concrete models of future system using the same program logic on higher level. Then we can produce different target codes for different architectures using these configurations.

4 Embedded Process Functional Language

In our work we are developing domain specific functional programming language (Embedded process functional language – e-$\mathcal{PFL}$) targeted to embedded systems development. Developed language can be used like a modeling or a prototyping language in early development phases. Presented language come from *Process Functional Language* – $\mathcal{PFL}$ [11] on language level. On implementation level it extends *Parallel Process Functional Language* [12] (that we have created before). Created language was introduced in [13].

Process Functional Language improves state representation by introducing variables while trying not to compromise its declarative nature. Usage of variables is bound to processes and is maintained by a compiler. That is why we are able to determine program parts manipulating with state at a compile time. Process application is the only place where we can access or update variables. The scope of variables is defined by the scope of processes that use them. Access and update of variable environment is uniform. We must use processes from the same scope. For instance programmer cannot directly access or update variables. Similar language constructions were used for e-$\mathcal{PFL}$.

Created language is shortly described in this section. It uses eager evaluation. Syntax and semantics come from $\mathcal{PFL}$ (it is close to pure functional subset of Haskell). This set of constructions was extended to support embedded systems development.

Embedded systems are often described as a set of functional units. Also in e-$\mathcal{PFL}$ embedded system is a set of communicating devices. These devices are modeled using data type `Device` for default module `Prelude`.

```
data Device = Process EmbProcess
            | Fair    [Device]
            | Unfair  [Device]
```

Devices are strictly built from embedded processes. Embedded processes cannot be used directly. Embedded processes are described by data type `EmbProcess` (this data type was used in `Device` definition). Embedded processes encapsulate issues related to communication on coordination layer. Syntax of embedded processes is close to common functions. Embedded process definition is extended by variables (like in processes in $\mathcal{PFL}$). Variables are bounded to parameters and also to return value (it can be a tupple and then variables are bounded to every tupple element). Following example shows embedded process definition.

```
work :: a Integer -> b Integer ->(c Integer, d Integer)
work x y = (x, x+y)
```

Used variables represent communication channels. Embedded processes define operations with known input (variables bounded to parameters: `a` and `b` in our example) and output (variables bounded to return value: `c` and `d` in our example). Input or output of created devices is defined by used embedded processes. Each of variables can be used like an input maximally by one device and like an output also maximally by one device.

Devices can be started using native function `startDevice`.

```
startDevice :: Device -> EmbSystem -> [Annotation] -> ()
```

Each `Device` is an autonomous system working independently on other devices. Devices are working asynchronously[4]. When a device is started, it tries to execute embedded processes that it encapsulates. Processes compete for execution time within a single device. Only one process can be running at given time. When there is no process running new candidate for execution is selected according to available input and *fairness* (according to data constructors `Fair` and `Unfair`).

These devices can be divided into distant parts — embedded system components. Embedded system components are defined using data type `EmbSystem` (first argument is component name, the second is mediator). Data type `Mediator` defines concrete mediator. Both are from basic module `Prelude`.

```
data Mediator = Hume | MicroNET

data EmbSystem = EmbComponent [Character] Mediator | Emulator
```

In our approach we do not produce one target code. Programmer can divide embedded system into parts. Each of these parts is associated with exactly one mediator. Each component can use different mediator and thus can have different features and properties. For every component is generated target code with respect to used mediator. We have integrated two mediators into e-$\mathcal{PFL}$ now. Programmer can use *Hume* like an intermediate language or created run-time environment for *.NET Micro Framework*.

For example when Hume is used as a mediator then source codes in Hume are produced by e-$\mathcal{PFL}$ compiler. These codes can be then ported using tools created by Hume developers or we can use developed tool to compute runtime constrains.

Using this technique we are able to address technological risks of embedded systems development. Also we are able to implement even distributed embedded systems as a single application in e-$\mathcal{PFL}$ and divide it into distant parts during application porting. Using this technique we are able to integrate several different languages or platforms into one solution. This solution then composes different approaches to embedded systems development and benefits from their properties.

[4] A timer can be implemented using library functions if needed. Form this point of view the timer is a device periodically generating output necessary for other devices to continue.

Finally we are able to change certain device features when the device is started. We are able to add *annotations* to started devices. Annotations are related to created configuration. They are stored into generated configuration XML file and they affect produced target code. Using annotation we are able to change coordination, initial values or target code optimization level. Annotations are defined using data type `Annotation`. Programmer must not use annotations directly. He can use standard functions from module `Prelude`. For example function `rename`. Using this function programmer can *rename* inputs or outputs for a device to change devices connections.

```
data Attribute  = CAttribute  [Character] [Character]
data Annotation = CAnnotation [Character] [Attribute]

rename :: [Character] -> [Character] -> Annotation
rename x y = CAnnotation "rename"
             [(CAttribute "old" y),(CAttribute "new" x)]
```

Communication is implicit in e-$\mathcal{PFL}$. Related issues are solved during a configuration run. Each of variables used in started devices represents a communication channel. Type of this communication channel is known at a compile time and we are able to compute all necessary information related to usage of these channels (for example initial values or underlying device architecture). Potential communication issues simplify that there is maximally one input device and one output device to each of these channels and each of them can hold up to one value only. Main issues are thus related to data synchronization. A sort of default communication is computed during the configuration run and computed information is stored in a resulted configuration. Programmer can modify this configuration in the future and thus he can control communication.

5 Example

This section shows simple application written in e-$\mathcal{PFL}$.

```
produce :: a Integer -> (a Integer, b Integer, c Integer)
produce x = (x+1, x, x)
work :: Device
work = Process produce

showB ::b Integer -> ()
showB x = writeLine (show x)
printer :: Device
printer = Process showB

annotation :: Annotation
annotation = rename "b" "c"

component1 :: EmbSystem
component1 = EmbComponent "worker_component" Hume
```

```
component2 :: EmbSystem
component2 = EmbComponent "printer_component" MicroNET

main= (startDevice work    component1 []) `bl`
      (startDevice printer component2 []) `bl`
      (startDevice printer component2 [annotation])
```

Listing 1.1: Simple example in e-𝒫ℱℒ

Previous example use process `bl`. This process comes from 𝒫ℱℒ . It forms a sequence of process. Its functionality is similar to construction `do` from Haskell.

Example composes from two components (first is using Hume, second is using .NET Micro Framework). Fist component contains one device based on `Device` named `work`. Second component contains two devices based on `Device` named `printer`. Example also shows how default connections of devices can be changed by a programmer (using annotations).

6 Conclusion and Future Work

We are developing a domain specific language called Embedded Process Functional Language (e-𝒫ℱℒ) targeted to embedded systems development. Created language works on a high level of abstraction. It uses other technologies (even other functional languages) created for embedded systems development on lower levels. It can be used like a modeling or a prototyping language in early development phases.

The contribution is that we are able to eliminate development risks using e-𝒫ℱℒ . In e-𝒫ℱℒ we are able to create working prototype of future system (or its critical parts). Then we can use this prototype in negotiation with customer to eliminate business risks. Applications created in e-𝒫ℱℒ can be simulated using implemented simulator. Using partial evaluation we are able to extract static model (or models) of future system. This model can be for example visualized and we can use it in communication with customer. Still created e-𝒫ℱℒ is a programming language and we can straightforwardly produce target codes. We are using other technologies on lower level (now we are using Hume and .NET Micro Framework) and we can benefit from their features. Produced codes can be directly used on real devices. Using this technique we can eliminate technological risks during the development process.

For practical experiments we have implemented e-𝒫ℱℒ simulator using .NET platform and a distributing cross-platform compiler. This compiler use Hume and .NET Micro Framework on a lower levels. Also we have created GUI containing tool that simplifies configuration of the applications.

Proposed e-𝒫ℱℒ is under active development now. We are considering other language constructs that may improve its capabilities. For example we are considering different language constructions changing devices connections. We are also improving implemented tools. For example compiler implements only basic language constructions now. Constructions like list generators (common in

Haskell) are not supported yet. Also we want to extend basic libraries. Another area is practical applications of presented ideas. We want to use e-$\mathcal{PFL}$ for development of real embedded systems.

On the other hand presented approaches and principles are actively developed and used mainly in academic circles. There is still a long way ahead to see if presented usage of functional paradigm can truly compete with current methodologies.

Work is partially supported by Czech-Slovak fund KONTAKT MEB 080878: Cooperation in area of design and implementation of language systems.

References

1. Highsmith, J., Fowler, M.: The agile manifesto. Software Development Magazine **9**(8) (2001) 29–30
2. Hughes, R.: Why functional programming matter. The Computer Journal **32**(2) (1989) 98–107
3. Vahid, F., Givargis, T.: Embedded System Design: A Unified Hardware/Software Introduction. John Wiley & Sons, Inc., New York, NY, USA (2001)
4. Wallace, M., Runciman, C.: Extending a functional programming system for embedded applications. Softw. Pract. Exper. **25**(1) (1995) 73–96
5. Armstrong, J.: A history of erlang. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, New York, NY, USA, ACM (2007) 6–1–6–26
6. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent haskell. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1996) 295–308
7. Loogen, R., Ortega-mallén, Y., Peńamarí, R.: Parallel functional programming in eden. J. Funct. Program. **15**(3) (2005) 431–475
8. Hammond, K., Michaelson, G.: Hume: A domain-specific language for real-time embedded systems. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings. Volume 2830 of Lecture Notes in Computer Science., Springer (2003) 37–56
9. Nyström, J.H., Trinder, P.W., King, D.J.: Evaluating distributed functional languages for telecommunications software. In: ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang, New York, NY, USA, ACM (2003) 1–7
10. Specht, E., Redin, R.M., Carro, L., Lamb, L.d.C., Cota, E.F., Wagner, F.R.: Analysis of the use of declarative languages for enhanced embedded system software development. In: SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design, New York, NY, USA, ACM (2007) 324–329
11. Kollár, J., Porubän, J., Václavík, P.: From eager pfl to lazy haskell. Computers and Artificial Intelligence **25**(1) (2006)
12. Běhálek, M., Šaloun, P.: Parallel process functional language. In: SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, Proceedings Volume II. (2007) 1–12
13. Běhálek, M., Šaloun, P.: Simulation of embedded applications implemented in embedded process functional language. In: First International Conference on Computational Intelligence, Modelling, and Simulation, Brno, Czech Republic, IEEE Computer Society (7-9 September 2009) 253–258