# Views, Program Transformations, and the Evolutivity Problem

Julien Cohen, Rémi Douence

## To cite this version:

Julien Cohen, Rémi Douence. Views, Program Transformations, and the Evolutivity Problem. 2010. hal-00481941v1

HAL Id: hal-00481941
https://hal.science/hal-00481941v1

Preprint submitted on 7 May 2010 (v1), last revised 28 Jan 2011 (v2)

# Views, Program Transformations, and the Evolutivity Problem

Julien Cohen[1] and Rémi Douence[1,2]

1: LINA (UMR 6241, CNRS, Université de Nantes, École des Mines de Nantes)
2: ASCOLA (INRIA, École des Mines de Nantes)

**Abstract**

In this article, we argue that a program transformation approach is a good way to solve the tyranny of the primary decomposition. We illustrate our transformation-based approach on a case study defined with the expression problem. We also propose the *evolutivity problem* based on the experience of the expression problem.

## 1 Introduction

Evolutivity has become a major criteria of quality for enterprise software. In most cases, evolutivity comes from design choices on the software architectures. But it is well know that it is difficult to find software architectures that are evolutive with respect to all concerns. This is illustrated by the so-called *expression problem* [6] (section 2) (also called the *extensibility problem*).

Many solutions have been proposed for the expression problem (for instance, see [7] for a survey). These solutions allow to extend a program in a modular way, by adding new compilation units (classes, modules, aspects, mixins...). So, they tackle extensibility, but they also require strong static type safety, separate compilation and no modification of the existing code. But evolution is not only extension, it is also maintenance. We do not always want to add new functionalities, we often need to modify the existing functionalities, to correct an error or to change a behavior for instance (corrective, adaptative or perfective maintenance [5]).

For this kind of evolution, solutions to the expression problem have a major drawback : changes become non modular. Indeed, the code to modify, which was initially contained in a single module (or another compilation unit), finds to be spread over several modules after extensions have been implemented (section 3). This is not surprising since the expression problem is focused on extension and forbids modifications of the existing code.

In this paper, we propose to solve the problem of modular evolutivity not with a programming language approach, but with a program transformation approach. First we show how a program transformation can allow to have modular extensions as well as modular changes (section 4). Second, we illustrate our idea with a (work in progress) program transformation between two given architectures that provide the same functionality (section 5). That particular transformation is based on a sequence of basic refactoring operations. We also discuss how that transformation can be subject to evolutions when needed (section 5.3).

## 2 The Expression Problem

The expression problem is a practical instance of conflict between two architectural choices.

Let us consider the expression evaluator in Figure 1. The data type `Expr` represents the expression language to be evaluated. It has a constructor for each literal (*e.g.*, `Const` for integers) and operator (*e.g.*, `Add` represents the addition). Functions for evaluating or printing expressions are defined by pattern matching: a case is defined by constructor.

```
data Expr =
    Const Int
  | Add (Expr,Expr)

eval (Const i)     = i
eval (Add (e1,e2)) = eval e1 + eval e2

stringOfExpr (Const i)     = show i
stringOfExpr (Add (e1,e2)) =
       stringOfExpr e1 ++ "+" ++ stringOfExpr e2
```

Figure 1: Expression Data Type in Haskell

Figure 2: Constructor decomposition (architecture of the program $P_{data}$)

This code is modular with respect to functions. Modularity is better seen in Figure 2 where modules are used to structure this program. The left-hand side of the figure shows the module dependencies: for instance, the module `EvalMod` depends on the data type and its constructors defined in the module `Expr`. The right-hand side of the figure shows a matrix of modules with respect to the constructor and function definitions: for instance, the module `EvalMod` uses both constructors `Const` and `Add` but it defines a single function `eval`. The corresponding code is detailed in the Figure 3. This program architecture makes

```
module Expr (Expr(Const,Add)) where

 data Expr =
     Const Int
   | Add (Expr,Expr)
```

```
module EvalMod (eval) where
 import Expr (Expr(Const,Add))

 eval (Const i) = i
 eval (Add (e1,e2)) = eval e1 + eval e2
```

```
module StringOfExprMod (stringOfExpr) where
 import Expr (Expr(Const,Add))

 stringOfExpr (Const i) = show i
 stringOfExpr (Add (e1,e2)) =
     stringOfExpr e1 ++ "+" ++ stringOfExpr e2
```

```
module Main (main) where
 import Expr (Expr(Const,Add))
 import EvalMod (eval)
 import StringOfExprMod (stringOfExpr)

 e1 = Add (Add (Const 1,Const 2),Const 3)
 main = print (stringOfExpr e1)
```

Figure 3: Functional decomposition in Haskell (program $P_{fun}$)

it easy to add a new function (*e.g.*, for simplifying expression). However, this code is not modular with respect to constructors. The code corresponding to a given constructor (*e.g.*, `Add`) is spread in all functions. So, when the data type is extended and a new constructor (*e.g.*, `Mult`) is introduced, all functions must be maintained in order to take into account the new constructor.

The alternative program structure in Figure 4 gathers all the pieces of code related to a given constructor in a corresponding module. For instance, the module `ConstMod` collects the `eval` equation and the `stringOfExpr` equation for `Const`. The right-hand side of the figure makes it explicit: in the matrix modules are not any more line but columns. The Figure 5 details the corresponding code. In particular, a function `dispatch` takes functions as parameters as well as and expression and it performs pattern matching for applying the right one. In this context, a function such as `eval` is defined as the (partial) application of `dispatch` to the functions for evaluation.

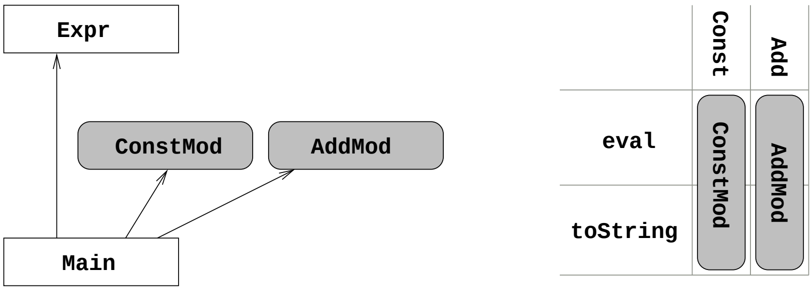


Figure 4: Constructor decomposition (architecture of the program $P_{data}$)

This alternative code is modular with respect to constructors. This program structure makes it easy to add an new constructor (*e.g.*, for product): the corresponding module is introduced and `dispatch` is extended with a new case. However, this code is not modular with respect to functions. The code corresponding to a given function (*e.g.*, `eval`) is spread in all modules. So, when a new function (*e.g.*, `simplify`) is introduced, every modules must be modified in order to take into account the new function.

This example shows the tyranny of the primary decomposition in action. Whatever primary program structure is chosen, it makes it non modular to extend the program in a way.

The expression problem has found various solutions in the literature. By the means of advanced features in the host language, is is possible to design program structures where it is modular to extend the data type, and modular to extend the functions. But, as we will see, these solutions share a same drawback : they do not allow modular maintenance.

# 3 The Evolutivity Problem

## 3.1 Extension is only part of the problem

In order to illustrate the problem with modular extension solutions, let us now make abstraction of the particular data type and functions and consider an

```
module Expr (Expr(Const,Add),e1, dispatch) where

data Expr =
    Const Int
  | Add (Expr,Expr)

e1 = Add (Add (Const 1,Const 2),Const 3)

dispatch f_c f_a (Const i) =
    f_c (dispatch f_c f_a ) i
dispatch f_c f_a (Add (e1,e2)) =
    f_a (dispatch f_c f_a) e1 e2
```

```
module ConstMod (eval,stringOfExpr) where

 eval = \ f i -> i

 stringOfExpr = \ f i -> show i
```

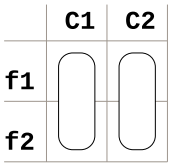
```
module AddMod (eval,stringOfExpr) where


 eval = \ f e1 e2 -> (f e1) + (f e2)

 stringOfExpr
     = \ f e1 e2 -> (f e1) ++ ("+" ++ (f e2))
```

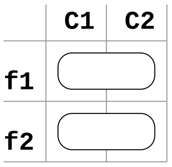
```
module Main (main) where

import Expr(Expr(Const,Add), dispatch, e1)
import ConstMod (stringOfExpr,eval)
import AddMod (stringOfExpr,eval)

eval = dispatch
       (ConstMod.eval)
       (AddMod.eval)


stringOfExpr =
    dispatch
    (ConstMod.stringOfExpr)
    (AddMod.stringOfExpr)


main = print (Main.stringOfExpr e1)
```

Figure 5: Constructor decomposition in Haskell (code of the program $P_{data}$)

incremental development scenario for an abstract application. We consider a data-type with two constructors `C1` and `C2` (`Const` and `Add` in the previous example), as well as two functions `f1` and `f2` (`eval` and `stringOfExpr` in the previous example).

The initial (view of the) program may focus on function extensibility or data extensibility. These two possible structures are pictured below.





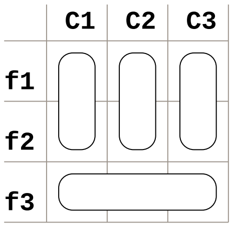
In these diagrams, the rounded boxes represent modules.[1] The left hand side diagram means that there is a module for each constructor of the data type and the code of the functions is spread over these modules. This illustrates the situation in the classical object approach (pattern composite) and in the architecture of figure 5. The right hand side diagram means that there is a module for each function and the code corresponding to a constructor of the data type is spread over these modules. This illustrates the situation of the classical functional approach (and also in the visitor pattern).

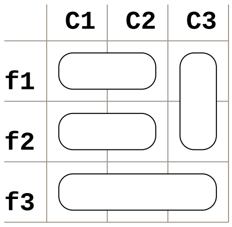
We now consider that we have chosen a particular solution to extend any axis by providing a new module (for instance, one of the solutions cited in [7]) and we will see what happens with the following scenario :

1. Extension : We introduce a new constructor `C3` (for instance `Mult`).
2. Extension : We introduce a new function `f3` (for instance `derivate`).
3. Extension : We introduce a new constructor `C4` (for instance `Div`).
4. Extension : We introduce a new function `f4` (for instance `check_div_by_zero`).
5. Maintenance : We modify the function `f1`.
6. Maintenance : We modify the data constructor `C1`.

The whole scenario is illustrated by Figure 6 and is detailed below.

**Two first extensions (steps 1 & 2).** After the first two steps, we will be in one of the following situations, depending on the initial program :





---

[1] We generalize the definition of module to: ”any modular entity of the programming language”. For instance, a function definition is a modular entity.

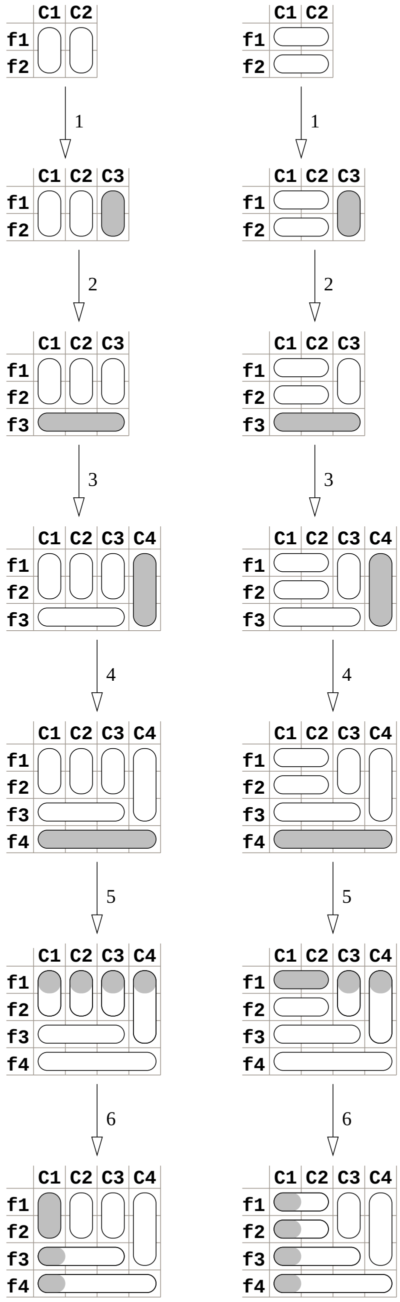


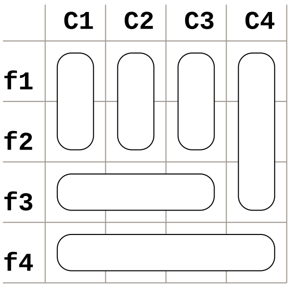
Figure 6: scenario for some solutions to the extension problem
Grey zones represent new or modified code.

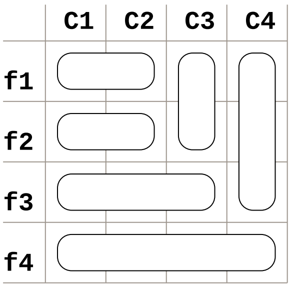
In the left hand side diagram, the extension of the data type with `C3` is natural, and adding the function `f3` can be done with the chosen specific mechanism (in this case, the module for `f3` has a different nature from the three other modules of the application).

In the right hand side diagram, we have extended `f1` and `f2` with the chosen specific modular feature to take `C3` into account and then we add `f3`. If we want the extension for `f3` to be fully modular, we have to define `f3` on `C1`, `C2` and `C3` in a single module. (An other solution would have been to make a module with `f3` defined on `C1` and `C2` and to complete the module of `C3`, but we do not consider this is modular.) Even if the modules for `f1`, `f2` and `f3` are of the same nature, they do not cover the same subset of constructors.

This means that one cannot fully rely on `f1` or `f2` as patterns to write `f3` (*problem 1*).

**Two following extensions (steps 3 & 4).** Now, let us take two more extensions into account.





In the left hand side diagram, `C4` is added naturally as a module, but we see that the corresponding module does not cover the same functions as the modules for `C1`, `C2` and `C4` (this boils downs to the problem 1). Then `f4` is added with the same technique as `f3` but, again, the module for `f4` does not cover the same cases as the module for `f3`.

We meet the same problems in the case of the right hand side diagram.

We can observe that the regular architecture of the initial programs rapidly becomes disordered with incremental extensions. This will reveal to be bad at maintenance time.

**Maintenance time (steps 5 & 6).** What if we have to modify `f1` (to correct an error or to cope with a change in its specification)? For instance, we could optimize the expression evaluation with simplifications such as `e+0=e` and `e*0=0`, or we could modify the conversion of expression to string in order from an infix notation to a prefix notation.

In the left hand side case we have nothing good to hope as the code for `f1` is already spread over several modules in the original program. In the (initially) operation-centered architecture, the code is finally spread also over several modules. This means that we have lost the benefits of the initial modularity: the maintenance is no more modular (*problem 2*).

This is the same for the maintenance of `C1`: in the data-centered architecture, the code has become spread over several modules.

Furthermore, as modules of the same nature cover different cases, the maintenance of a function (or of a constructor) is not the same as the maintenance of an other function. For instance, in the operation-centered architecture (right hand side diagram), modifying `f3` requires to modify (at most) 2 modules whereas modifying `f2` requires to check 3 modules. The maintenance task has lost the regularity it had in the initial programs (*problem 3*).

The example of this section shows that, in practice, the technical solutions for modular extensibility are not sufficient for modular maintainability. Let us now reformulate the problem.

### 3.2 The Evolutivity Problem and Modular Evolutions

As the expression problem focuses on the modularity of extensions, we formulate the *evolutivity problem* which focuses on the modularity of evolutions (extensions and maintenance). The evolutivity problem consists in providing means to implement extensions or changes in a modular way, regardless of whether the evolution is data-centered or function-centered (or, ultimately, regardless of the kind of concern the evolution addresses).

We now precise what we mean by modular, since we want to be less restrictive than in the expression problem.

**Modular evolutions.** Suppose we want to modify the constructor `Const` in our example program. For instance, in the definition of the type `Expr`, we want to change `Const Int` into `Const (Int,String)`. In the data-centered program, $P_{data}$, we have to modify the type `Expr`, but also the dispatcher and the type of its parameter functions (to take into account the string). As a consequence, the type of the functions in `ConstMod` changes. For instance, `eval =  f i -> i` is changed into `eval =  f i s -> i`. In this example, we consider that the change is modular because the "business logic change" is located into a single module, the other changes are "administrative" (mechanical consequences of the change in the business logic).

For this reason, we say that an evolution is *modular* when the business logic code to be changed or added is limited to a single module.

## 4 Programs, Views and Transformations

In this section, we describe a solution to the evolutivity problem which is based on program transformations.

### 4.1 Programs and Views

What is a program? Depending on the point of view, we can say that two programs are equals if they are syntactically, semantically or computationally equivalent for instance. So, for a given relation of equivalence, a program is a set elements of the language which are equivalent (*i.e.* a class of equivalence). For a given program, that is for a given class of equivalence, we call *views* of the program the elements of the class.

Depending on your needs, you will choose the appropriate relation of equivalence. The usual $\alpha$-equivalence is an example of relation that can be used to denote the computational equivalence. The relation of equivalence on terms of the $\lambda_v$-calculus defined in [4] is an example of relation denoting the semantic equivalence. The equality of the textual representations is the smallest relation of equivalence : any change in the textual representation is considered to change to program. For instance, when you obfuscate a program (by removing ends of lines (EOL), removing the comments and renaming all the variables for instance), you change the properties of the program (from a human point of view). This last relation is too restrictive to be useful in the problem addressed in this paper, but it fits our definition of programs and views : a program has a unique view in this case.

Of course, some of these relations are undecidable, but, in the following, we will focus on views for which the equivalence is decidable.

Following these definitions, $P_{data}$ and $P_{fun}$ are two views of the same program. The relation of equivalence of interest here is the functional behavior equivalence.

## 4.2 Refactoring tools to navigate between views

Code refactoring boils down to pass from one view of a program to another (for the functional behavior equivalence). Several popular languages have a refactoring tool. These tools provide elementary refactoring operations. The process of refactoring a program consists in applying successively such operations on a (view of a) program. If each of these operations is correct, that is if the input and the result are functionally equivalent, applying successively elementary refactoring operations is also correct since equivalence is transitive. For this reason, refactoring tools allow to navigate into classes of equivalence of programs. In the general case, the whole class cannot be accessed by this means, but it is not important in our case.

It is important to note that some of these tools are not correct. In particular, this is currently the case for the tool in Eclipse for Java. But some tools, such as the Haskell Refactorer (HaRe) [2] take a particular attention to provide correct operations [3].

## 4.3 A Solution to the Evolutivity Problem

We propose to use navigation between views to solve the evolutivity problem (and the more general problem of tyranny of the primary decomposition).

To illustrate this solution, let us consider that we have a tool such that if we have a data-centered view of a program, the tool can compute the operation-centered view of the same program, and vice-versa. For instance, in our example of section 2, the tool should be able to transform $P_{data}$ into $P_{fun}$ and vice-versa. With such a tool, the programmer can choose the view in which the evolution he has to implement is modular.

Figure 7 illustrates the scenario of the previous section with such an approach. For instance, when the programmer wants to add a new constructor (step 1), the program is first presented in the data-centered view ; when the programmer wants to add a new function (step 2), the program is first presented in the operation-centered view.

Since no evolution is made transversally to the considered axis of decomposition, the views of interest always keep a regular architecture.

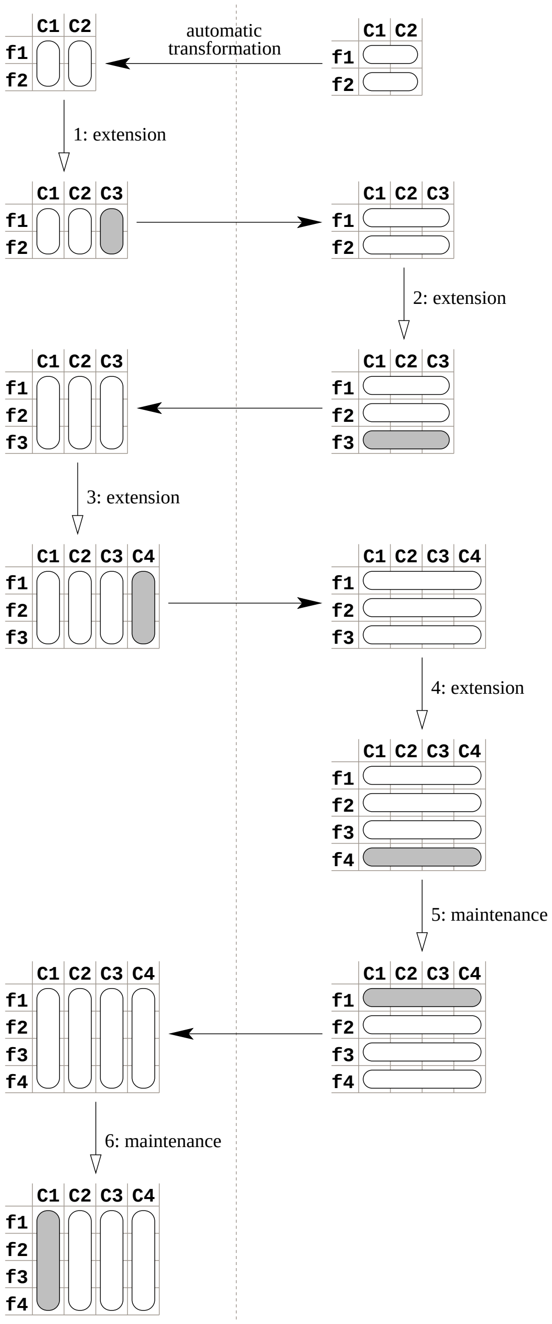


Figure 7: evolution scenario in our program transformation setting
Grey zones represent new or modified code.

This approach has several advantages:

- Virtually any programming language can be used. As soon as two alternative programming structures can be expressed in a language, the corresponding transformation can be set.
- As a consequence of the previous point, the programmer does not have to learn a new language or possibly complex language features.[2]
- The approach is not limited to two views.

Of course, this solution needs a tool to switch between the appropriate views of the program. We do not explore the different possibilities to build such a tool in this paper. Instead we provide in the next section a possible implementation, which is composed of two sequences of refactoring operations (one for each way). Since each step of refactoring is correct (we suppose it is guaranteed by the refactoring tool), the two transformations $P_{fun} \rightarrow P_{data}$ and $P_{data} \rightarrow P_{fun}$, which are inverse of each other, are also correct. This explains why we have claimed that $P_{fun}$ and $P_{data}$ where two views of a same program.

As we will see, the implementation we describe is a work in progress as it is not yet fully automatized.

# 5 Implementation of a Transformation

In this section we give a transformation between the two architectures of our example program. This transformation can be defined as a global correspondence between two patterns of programs (see appendix A), but we prefer to define it by a sequence of elementary transformations, following the divide and conquer strategy. Each of these elementary transformations can be seen as a refactoring operation, so that we can use refactoring tools instead of designing a specific tool from scratch. Such a refactoring tool is available for Haskell : the Haskell Refactorer (HaRe) [2].

Using a sequence of refactoring steps instead of a global transformation has the advantage that each elementary step is easy to understand, to prove correct (if necessary), to maintain and to reuse. Furthermore, many of these refactoring operations are already available in existing tools.

HaRe has the good property that it will check whether a transformation is correct in the considered context before applying it [3]. For this reason, we do not have to provide the proof of correctness for each elementary step of our transformation if it is handled by HaRe. However, one of the elementary steps is not supported by HaRe, so we will prove its correctness separately. At the moment, we cannot just rely on HaRe to handle the transformation. However, we present the steps of transformation as a case study, assuming that an engineering effort is still necessary to make the whole transformation automatic.

## 5.1 Decomposing the transformation

In order to explain the refactoring steps, we will use two intermediate programs, noted $P_1$ and $P_2$ which are given in figures 8 and 9. In the following we give

---

[2] There is the counterpart that he may have to cope with several views.

```
module Expr (Expr(Const,Add), dispatch) where

data Expr =
    Const Int
  | Add (Expr,Expr)

dispatch f_c f_a (Const i) =
    f_c (dispatch f_c f_a ) i
dispatch f_c f_a (Add (e1,e2)) =
    f_a (dispatch f_c f_a) e1 e2
```

```
module EvalMod (eval) where

import Expr (Expr(Const,Add), dispatch)

eval = dispatch
       (\ f i -> i)
       (\ f e1 e2 -> (f e1) + (f e2))
```

```
module StringOfExprMod (stringOfExpr) where

import Expr (Expr(Const,Add), dispatch)

stringOfExpr = dispatch
               (\ f i -> show i)
               (\ f e1 e2 -> (f e1) ++ ("+" ++ (f e2)))
```

```
module Main (main) where

import Expr(Expr, e1)
import StringOfExprMod(stringOfExpr)
import EvalMod(eval)

e1 = Add (Add (Const 1,Const 2),Const 3)

main = print (stringOfExpr e1)
```

Figure 8: The program $P_1$

```
module Expr (Expr(Const,Add), dispatch) where

data Expr =
    Const Int
  | Add (Expr,Expr)

dispatch f_c f_a (Const i) =
    f_c (dispatch f_c f_a ) i
dispatch f_c f_a (Add (e1,e2)) =
    f_a (dispatch f_c f_a) e1 e2
```

```
module EvalMod (eval) where

import Expr (Expr(Const,Add), dispatch)

eval = dispatch evalConst evalAdd

evalAdd = \ f e1 e2 -> (f e1) + (f e2)

evalConst = \ f i -> i
```

```
module StringOfExprMod (stringOfExpr) where

import Expr (Expr(Const,Add), dispatch)

stringOfExpr =
    dispatch stringOfExprConst stringOfExprAdd

stringOfExprAdd
    = \ f e1 e2 -> (f e1) ++ ("+" ++ (f e2))

stringOfExprConst = \ f i -> show i
```

```
module Main (main) where

import Expr(Expr, e1)
import StringOfExprMod(stringOfExpr)
import EvalMod(eval)

e1 = Add (Add (Const 1,Const 2),Const 3)

main = print (stringOfExpr e1)
```

Figure 9: The program $P_2$

an overview of the intermediate transformations. Detailed instructions to make HaRe do the transformations are given in appendix C.

$$P_{fun} \longleftrightarrow P_1 \longleftrightarrow P_2 \longleftrightarrow P_{data}$$

**Step $P_{fun} \longleftrightarrow P_1$ : transform a recursive definition into an use of an iterator.** The functions `eval` and `stringOfExpr` of $P_{fun}$ can be seen as simple traversals of trees. It is well known that this kind of function can be expressed with iterators such as the classical `fold` operator. The program $P_1$ corresponds to the replacement of recursive definitions of `eval` and `stringOfExpr` into definitions using such an iterator.

The reason that made us choose the iterator `dispatch` instead of `fold` is that it is more general.[3]

This is the transformation that is not supported by HaRe. We have proven with Coq [1] that the programs $P_{fun}$ and $P_1$ are equivalent (in the functional programming language provided with Coq), see appendix B.

**Step $P_1 \longleftrightarrow P_2$ : make global definitions for anonymous functions.** In $P_2$, some functions that were used as argument of the `dispatch` iterator in $P_2$ are now defined globally. The programs are equivalent as long as the new definitions are not exported and there are no other definitions with the same name in the corresponding module. This step in handled by HaRe in the two directions $P_1 \longrightarrow P_2$ and $P_1 \longleftarrow P_2$, as well as the verifications that are necessary to ensure the correctness.

**Step $P_2 \longleftrightarrow P_{data}$ : move definitions into the right modules.** Transforming $P_2$ into $P_{data}$ consists in moving definitions from "function modules" to "constructor modules". Additionally, some minor syntax cleaning has to be done, such as renaming of the functions and creating/removing empty modules.

## 5.2 Automation of the process.

Currently, to have a fully automatic transformation, the following tasks have to be automatized:

1. The step $P_{fun} \longleftrightarrow P_1$ to transform a recursive function into an use of an iterator.
2. The orchestration of the successive HaRe refactorings.

As this is a work in progress, it is unclear at the moment how hard these tasks are. Since HaRe operations are available as a library (in addition to the human-computer interface), we can hope to simply use this library. However, it might be necessary to extend HaRe itself.

It would be also useful to provide an high level interface to specify the orchestration of the elementary refactorings, for instance by the means of a domain specific language.

---

[3] For instance, `dispatch` allows to define left-to-right evaluators and right-to-left evaluators whereas the use of `fold` does not allow to choose the order of evaluation of the subtrees.

## 5.3 Case study on the maintenance of the transformation

Now that we have discussed a possible implementation of a transformation operation, let us discuss the possibility to make it evolve at we meet “severe” evolutions in the considered program.

A transformation must be sufficiently general to cope with a family of changes that will occur in one of the views of the program. For instance, adding a function or a case in the data type should not make us to change the transformation. However, during the project’s life, we may have to implement evolutions that break the transformations. In this section, we illustrate that case. For a given example, we will identify the steps of the transformation that change. Let us now focus again on our initial example program. We consider we want to add an environment as a parameter to the function `eval` to be able to take variables into account later.

The additional parameter to `eval` makes that the iterator `dispatch` is not convenient anymore in the data-centered architecture. We will use an other iterator.

### 5.3.1 Severe modification of a function

In order to extend `eval`, we use the operation-centered view. We make the following modification :

```
eval (Const i) = i
eval (Add (e1,e2)) = eval e1 + eval e2
```

becomes

```
eval env (Const i) = i
eval env (Add (e1,e2)) = eval env e1 + eval env e2
```

This modification is modular in this view. Of course, you have to add an environment in the existing calls to eval, in particular in the module `Main`.

### 5.3.2 Adapting the transformation

The difference with the initial transformation is that we have to use a different iterator. The fact that functions passed as argument to the iterator, which are defined in the “constructor modules”, have an additional parameter is transparent for our transformation since HaRe handles the functions without respect to their parameters.

The code to appear in $P_1$ is the following :

```
dispatch_env f_c f_a env (Const i) =
    f_c (dispatch_env f_c f_a) env i
dispatch_env f_c f_a env (Add (e1,e2)) =
    f_a (dispatch_env f_c f_a) env e1 e2

eval env = dispatch_env
       (\ f env i -> i)
       (\ f env e1 e2 -> (f env e1) + (f env e2))
       env
```

It should be noticed that `eval` and `stringOfExpr` are not treated uniformly during the first transformation step since they do not use the same dispatcher. It does not mean that each function should have its own dispatcher since `dispatch_env` can be used for any number of additional parameter as long as you group them in a tuple. The iterator `dispatch_env` could also be used to define `stringOfExpr`, but then you would have a strange dummy parameter in the definitions of `stringOfExpr` in the constructor modules.

Since the following steps do not change, we can account on a good reuse of the first transformation to build the second one. We finally get the data-centered view :

```
module ConstMod (eval, StringOfExpr) where
 stringOfExpr = \ f i -> show i
 eval = \ f env i -> i
```

```
module AddMod (eval, StringOfExpr) where
 stringOfExpr s = \ f e1 e2 -> (f e1) ++ (s ++ (f e2))
 eval = \ f env e1 e2 -> (f env e1) + (f env e2)
```

## 6 Conclusion

In this paper, the need to consider not only the extensions but also maintenance when dealing with modularity has led us to formulate the evolutivity problem. We have seen that solutions to the expression problem do not fit in the evolutivity problem.

We have identified a notion of views of programs in order to propose a solution to the evolutivity problem. This solution is based on program transformation techniques. In particular, refactoring tools may be used to build a transformation that is needed in our solution. Building such a tool is not done yet but promising experiments have been done.

The next big steps to make this views approach usable are :

1. to provide tools that help programmer to build easily (and correctly) their own transformations ;
2. to study how transformations are impacted by major changes in a view.

Our approach has the following pros and cons with respect to the solutions to the expression problem :

- It does not depend on particular language features,
- it is not limited to two views,
- the transformation has to be built,
- the programmer may have to cope with several views, which may be disturbing (but each view may be kept simple since specific language features are not needed).

# References

[1] The Coq Proof Assistant. Web site : `http://coq.inria.fr`.

[2] Hare – the Haskell Refactorer. Web page, 2003-2009. `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`.

[3] H. Li and S. Thompson. Formalisation of Haskell Refactorings. In M. van Eekelen and K. Hammond, editors, *Trends in Functional Programming*, September 2005.

[4] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[5] E. B. Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[6] P. Wadler. The expression problem. Message to java-genericity electronic mailing list, November 1998.

[7] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

# A Global transformation

In this appendix, we define a global transformation between the data-centered architecture and the operation-centered architecture by expressing these architectures as patterns where with exhibit which part of the code are common. Figures 10 and 11 give these two patterns.

In addition to the set of constructors $\{C_1, \cdots, C_n\}$ and to the set of functions $\{f_1, \cdots, f_p\}$, the common parts in the two architectures are the expressions noted $E_{ij}$, from which some parameters have been abstracted. We note $E[x_1, \cdots, x_k]$ an expression $E$ where the identifiers $x_1$ to $x_k$ have be abstracted, which means that we can write $E[a_1, \cdots, a_k]$ to express the substitution of the $x$'s for the $a$'s in $E$.

One one hand, this transformation is easy to specify (as done here). But on the other hand, it does not provide any hints to decompose it into small parts to make it easier to implement or prove correct.

```
data $t$ = $C_1$ $t_1$ | ...  | $C_n$ $t_n$

dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ($C_1$ ($x_1, x_2, \cdots, x_{a_1}$))
   = $g_1$ (dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ) $x_1$ $x_2$ $\cdots$ $x_{a_1}$
dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ($C_2$ ($x_1, x_2, \cdots, x_{a_2}$))
   = $g_2$ (dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ) $x_1$ $x_2$ $\cdots$ $x_{a_2}$
...
dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ($C_n$ ($x_1, x_2, \cdots, x_{a_n}$))
   = $g_n$ (dispatch $g_1$ $g_2$ $\cdots$ $g_n$ ) $x_1$ $x_2$ $\cdots$ $x_{a_n}$

module $C_1$Mod ($f_1, \cdots, f_p$) where
  $f_1$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_1}$ = $E_{1\,1}[h, x_1, x_2, \cdots, x_{a_1}]$
  $f_2$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_1}$ = $E_{1\,2}[h, x_1, x_2, \cdots, x_{a_1}]$
  ...
  $f_p$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_1}$ = $E_{1\,p}[h, x_1, x_2, \cdots, x_{a_1}]$

...
module $C_n$Mod ($f_1, \cdots, f_p$) where
  $f_1$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_n}$ = $E_{n\,1}[h, x_1, x_2, \cdots, x_{a_n}]$
  $f_2$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_n}$ = $E_{n\,2}[h, x_1, x_2, \cdots, x_{a_n}]$
  ...
  $f_p$ $h$ $x_1$ $x_2$ $\cdots$ $x_{a_n}$ = $E_{n\,p}[h, x_1, x_2, \cdots, x_{a_n}]$

module Main (main) where
import qualified ...
$f_1$ = dispatch $C_1$Mod.$f_1$ $C_2$Mod.$f_1$ ...  $C_n$Mod.$f_1$
$f_2$ = dispatch $C_1$Mod.$f_2$ $C_2$Mod.$f_2$ ...  $C_n$Mod.$f_2$
...
$f_p$ = dispatch $C_1$Mod.$f_p$ $C_2$Mod.$f_p$ ...  $C_n$Mod.$f_p$

main = ...
```

$n$ : number of constructors
$a_i$ : arity of the constructor $C_i$
$p$ : number of traversal functions

Figure 10: data-centered Program Scheme

```
data $t$ = $C_1$ $t_1$ | ...  | $C_n$ $t_n$

module $f_1$Mod ($f_1$) where
  import ($t$($C_1, C_2, \cdots, C_n$))
  $f_1(C_1$ ($x_1, x_2, \cdots, x_{a_1}$)) = $E_{1\,1}[f_1\ x_1, x_2, \cdots, x_{a_1}]$
  $f_1(C_2$ ($x_1, x_2, \cdots, x_{a_2}$)) = $E_{2\,1}[f_1\ x_1, x_2, \cdots, x_{a_2}]$
  ...
  $f_1(C_n$ ($x_1, x_2, \cdots, x_{a_n}$)) = $E_{n\,1}[f_1\ x_1, x_2, \cdots, x_{a_n}]$

module $f_2$Mod ($f_2$) where
  import ($t$($C_1, C_2, \cdots, C_n$))
  $f_2(C_1$ ($x_1, x_2, \cdots, x_{a_1}$)) = $E_{1\,2}[f_2\ x_1, x_2, \cdots, x_{a_1}]$
  $f_2(C_2$ ($x_1, x_2, \cdots, x_{a_2}$)) = $E_{2\,2}[f_2\ x_1, x_2, \cdots, x_{a_2}]$
  ...
  $f_2(C_n$ ($x_1, x_2, \cdots, x_{a_n}$)) = $E_{n\,2}[f_2\ x_1, x_2, \cdots, x_{a_n}]$

...

module $f_n$Mod ($f_n$) where
  import ($t$($C_1, C_2, \cdots, C_n$))
  $f_n(C_1$ ($x_1, x_2, \cdots, x_{a_1}$)) = $E_{1\,n}[f_n\ x_1, x_2, \cdots, x_{a_1}]$
  $f_n(C_2$ ($x_1, x_2, \cdots, x_{a_2}$)) = $E_{2\,n}[f_n\ x_1, x_2, \cdots, x_{a_2}]$
  ...
  $f_n(C_n$ ($x_1, x_2, \cdots, x_{a_n}$)) = $E_{n\,n}[f_n\ x_1, x_2, \cdots, x_{a_n}]$

module Main (main) where
  import ...
  main = ...
```

$n$ : number of constructors
$a_i$ : arity of the constructor $C_i$
$p$ : number of traversal functions

Figure 11: operation-centered Program Scheme

# B Proof of correctness of the first step of transformation

In this appendix we first give a proof that the definition of `eval` and `stringOfExpr` with an iterator `fold` are equivalent to the recursive definition with pattern matching. Then we discuss the proof for the case of the `dispatch` iterator, which correspond to the step $P_{fun} \longleftrightarrow P_1$.

## B.1 Proof for the use of `fold`

We give here the Coq definitions, theorems and their proofs stating that using `fold` to define `eval` and `stringOfExpr` is correct.

```
(* Proof that the use of an iterator preserves the   *)
(* semantics. We use fold as instance of iterator.   *)
(* Proofs for dispatch are more complex and more
   restrictive, they are given separately.           *)

(* ----- the inductive data type and ---------------
   ----- the corresponding iterator ---------------- *)
Inductive expr : Set :=
  | Const : nat -> expr
  | Add : expr -> expr -> expr.


Fixpoint fold (t:Type) f_c f_a (e:expr) : t :=
match e with
   | Const i => f_c i
   | Add e1 e2 =>
           f_a (fold t f_c f_a e1) (fold t f_c f_a e2)
end.
(* remark : (t:Type) expresses the polymorphism *)

(* ---------- FUNCTION EVAL ----------------------- *)

Fixpoint eval (e:expr) : nat := match e with
  | Const n    => n
  | Add e1 e2 => eval e1 + eval e2

end.

(* eval with fold *)
Definition eval2 (e:expr) :=
       fold nat (fun i => i) (fun n1 n2 => n1 + n2) e.

(* theorem and proof that these two definitions
                                      are equivalent *)
Theorem eval_equal_fold_eval :
   forall e, eval e = eval2 e.
```

```
Proof.
 induction e.

 compute ; auto.

 simpl.
 unfold eval2.
 simpl.
 fold (eval2 e1) ;  fold (eval2 e2).
 auto.
Qed.

(* -------- FUNCTION STRINGOFEXPR -----------------*)

(* We keep the string data type abstract
             to show the generality of the proof. *)

Parameter abstractstring : Set.
Parameter string_of_nat : nat -> abstractstring.
Parameter concat :
   abstractstring -> abstractstring -> abstractstring.

Fixpoint stringOfExpr (e:expr) : abstractstring :=
  match e with
  | Const n     => string_of_nat n
  | Add e1 e2 =>
          concat (stringOfExpr e1) (stringOfExpr e2)
end.

Definition stringOfExpr2 (e:expr) :=
          fold abstractstring string_of_nat concat e.

Theorem stringOfExpr_equal_stringOfExpr2 :
          forall e, stringOfExpr e = stringOfExpr2 e.

Proof.
 induction e.

 compute ; auto.

 simpl.
 unfold stringOfExpr2.
 simpl.
 fold (stringOfExpr2 e1) ;  fold (stringOfExpr2 e2).
 rewrite IHe1 ; rewrite IHe2 ; auto.
Qed.
```

### B.2 Proof for the use of `dispatch`

The `dispatch` iterator is more general than `fold`, it leaves more control to the programmer. In particular, one can use non terminating functions using `dispatch`, which is not the case with `fold` (as long as each argument function always terminate). For instance, `dispatch` can be used to write an evaluator for terms expressing recursion.

For this reason, the proof is more difficult to set in Coq for the use of `dispatch` and we provide it in the file `eval_dispatch.v` instead of inserting it into this report.

## C Detailed use of HaRe

We detail in this appendix the procedure to follow to make HaRe perform the transformation $P_1 \longleftrightarrow P_2 \longleftrightarrow P_{data}$. At the time this report is written, there are some bugs left in HaRe, we will explain how to pass over them.

In the following, $\mathcal{F}$ is the set of function names to consider and $\mathcal{C}$ is the list of constructors to consider. There is no ordering in $\mathcal{F}$ but the order in $\mathcal{C}$ is important because of the order of the arguments of the `dispatch` operator. In our example program, we have $\mathcal{F} = \{eval, stringOfExpr\}$ and $\mathcal{C} = [Const, Add]$.

For any $c \in \mathcal{C}$, $pos(c)$ denotes the position of $c$ in $\mathcal{C}$. It also corresponds to the position of the argument for the constructor $c$ in the parameters of `dispatch`.

We give the algorithms as sequences of commands corresponding to actions in Emacs in the HaRe mode.

### C.1 Step $P_1 \rightarrow P_2$ : extract local definitions

In this step, new names have to be given for new definitions. We will create these new names by concatenating existing names : we note $f.c$ the identifier corresponding to the concatenation of the identifier $f$ and the name of a constructor $c$. For instance, `eval.Add` stands for `evalAdd`.

We call the *definition of* $f$ the point of the program where the identifier $f$ is locally or globally bound to an expression, if there is only one such definition.

The Algorithm 1 gives the sequence of commands to run to make the transformation $P_1 \rightarrow P_2$.

### C.2 Step $P_2 \rightarrow P_{data}$ : move definitions into coherent modules

The algorithm for the step $P_2 \rightarrow P_{data}$ is given in two parts, that is $P_2 \rightarrow P_3 \rightarrow P_{data}$, with an intermediate state $P_3$. The first part, $P_2 \rightarrow P_3$ in Algorithm 2, gives the instructions to reach a data-centered architecture. The second part, $P_3 \rightarrow P_{data}$ in Algorithm 3, gives some final syntactic changes to reach $P_{data}$ which are not currently supported by HaRe.

In these algorithms, $n$`Mod` stands for a name (constructor name or function name) resulting from the concatenating of the name $n$ and the suffix `"Mod"`. For instance, `AddMod` is build on this model.

During this step, the compilation sequence changes as the program is transformed. Note also that the program $P_{data}$ is compiled with the `ghc` option `-fno-monomorphism-restriction`.

**Algorithm 1** $P_1 \rightarrow P_2$

```
for all $f \in \mathcal{F}$ do
  for all $c \in \mathcal{C}$ do
    In the definition of $f$, select the argument of dispatch in position $pos(c)$.
    In the HaRe menu, run "introduce new def".
    Answer $f.c$ when prompted for a name for the new definition (a new local definition for $f.c$ is created and $f.c$ is used as argument of dispatch).
    Place the cursor at the beginning of the identifier $f.c$.
    In the HaRe menu, run "lift def to top level" (the local definition is transformed into a global definition).
  end for
end for
```

**Algorithm 2** $P_2 \rightarrow P_3$

```
for all $c \in \mathcal{C}$ do
  Create an empty module named $c$Mod.
  Use the "Add file" menu to add the module in the HaRe project.
end for
for all $c \in \mathcal{C}$ do
  for all $f \in \mathcal{F}$ do
    Place the cursor on the identifier $f.c$.
    Run the HaRe command "move def to another module" from the menu.
    When prompted for a module name, answer $c$Mod (the definition is moved and export/import statements are added).
  end for
end for
for all $c \in \mathcal{C}$ do
  for all $f \in \mathcal{F}$ do
    Rename $f.c$ into $f$.
  end for
end for
```

**Algorithm 3** $P_3 \rightarrow P_{data}$

```
for all $f \in \mathcal{F}$ do
  Move $f$ from $f$Mod to the module Main (and update the import/exports).
  Remove the module named $f$Mod.
end for
```

**Bugs in HaRe.** When applying Algorithm 2, the following bugs in the current version of HaRe (0.5) are encountered when you move a definition from a module to another :

- You may have to add “ends of line” between two definitions in the target module.
- You may need to correct the indentation of some definitions in the target module.
- The literal `"+"` is not handled by the refactoring “move def to another module”. *That bug can be corrected in HaRe source code by adding a line in* `refactorer/RefacUtils.hs`*. The* `diff` *patch is the following :*

```
2534a2535
>          lit (HsString s)= return (HsString s)
```

Furthermore, HaRe should be able to handle the moves in Algorithm 3 but we have encountered blocking problems when using it. Finally, some steps which have been done by hand, such as creating/removing modules, are features that should be easy to add in refactoring tools like HaRe.

## C.3 From $P_{data}$ to $P_2$ : expand functions defined in other modules

**Algorithm 4** $P_{data} \rightarrow P_1$

```
for all $f \in \mathcal{F}$ do
  for all $c \in \mathcal{C}$ do
    Place the cursor on the qualified name $m_c.f$ in the definition of $f$ (module Main).
    In the HaRe menu, run “unfold def” (the name is replaced by the corresponding function).
    et remove def
  end for
end for
In the HaRe menu, run “clean imports” (module Main).
for all $c \in \mathcal{C}$ do
  Remove the empty module $m_c$.
end for
for all $f \in \mathcal{F}$ do
  Create a new empty module named $m_f$ and add it to the HaRe project.
  Add “import Expr(Expr(Const,Add), dispatch)” into that module.
  Place the cursor on the identifier $f$ (module Main).
  In the HaRe menu, run “Move def to another module”.
  When prompted for a module name, answer $m_f$.
end for
```

Note : in the general case, if intermediate functions are used in the functions to be moved, they should also be taken into account.