Views, Program Transformations, and the Evolutivity Problem in a Functional Language

Julien Cohen, Rémi Douence

Views, Program Transformations, and the Evolutivity Problem in a Functional Language*

Julien Cohen[1] & Rémi Douence[2]

1: Université de Nantes – LINA (UMR 6241, CNRS, Univ. Nantes, ÉMN)
2: INRIA – ASCOLA team (ÉMN - INRIA - LINA)

We report on an experience to support multiple views of programs to solve the tyranny of the dominant decomposition in a functional setting. We consider two possible architectures in Haskell for the classical example of the expression problem. We show how the Haskell Refactorer can be used to transform one view into the other, and the other way back. That transformation is automated and we discuss how the Haskell Refactorer has been adapted to be able to support this automated transformation. Finally, we compare our implementation of views with some of the literature.

1 Introduction

Evolutivity is a major criteria of quality for enterprise software. Evolutivity is strongly related to the design choices on the software architectures. However, it is generally impossible to find software architectures that are evolutive with respect to all concerns. So, one of these concerns has to be privileged (section 2.1). As shown by the solutions to the *expression problem* [28], there are many ways, often based on specific language features, to provide modular extensions which are orthogonal to the main axis of decomposition of the architecture (section 2.2). However, these solutions focus on extensions and generally break the regularity of the initial architecture, leading to a decrease in the maintainability (section 2.3). This shows that the modular extensibility and maintainability on orthogonal concerns are difficultly supported at the language level.

Multiple views [7] tackle the problem of modular evolutivity with a program transformation approach instead of a programming language approach. For a given application, the source code of several equivalent architectures can be computed one from another, so that the programmer who has to implement an evolution can choose the architecture which his the most convenient for his task. With proper tools, the implemented evolutions are reflected in all the available architectures.

In this paper, we report on an experience of providing support for multiple views for a functional language. In the following, we consider the classical example of a simple evaluator coming from the expression problem [28] and we illustrate how multiple views can provide modular extensions as well as modular changes on several orthogonal axis (section 3). Then, we propose an implementation of a transformation from one view to another. That transformation is based on a refactoring tool (section 4). Last, we discuss the work to make this kind of tool usable for enterprise software (section 5) and we compare our experience to other tools for multiple views (section 6.3).

2 Modularity and Evolution

In this section, we illustrate the tyranny of the dominant decomposition in a functional language setting with a simple example (section 2.1), we recall the definition of the expression problem (section 2.2), which is closely related to our problem, and we focus on maintenance and show that it is not well covered by the expression problem as extensions tend to degrade the initial structure (section 2.3).

*This is the second version of the report initially entitled *Views, Program Transformations, and the Evolutivity Problem.*

2.1 Each Architecture privileges extensibility on a given axis

When choosing a module structure for a given program, one has to choose between several possibilities with different advantages and disadvantages [22]. We illustrate this with two possible module structures for a simple evaluator which have dual advantages and disadvantages. This program is the same that is often used to motivate the expression problem, here given in Haskell.

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

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

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

The data type `Expr` represents the expression language to be evaluated. This data type has a constructor for literals (`Const` for integers) and another for an operator (*e.g.*, `Add` represents the addition). Two functions, for evaluating or printing expressions, are defined by pattern matching: a case is provided for each constructor.

This code is modular with respect to functionalities (because of the scope introduced by function declarations). The modularity is better seen in Figs. 1 and 2(a) where modules are used to structure the program.

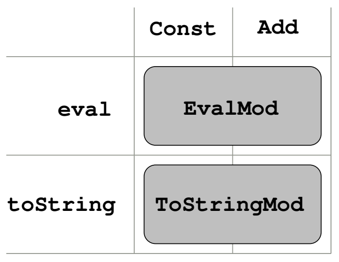
```
module Expr  where

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

e1 = Add (Add (Const 1,Const 2),Const 3)
e2 = Add (Add (Const 0,Const 1),Const 2)
```

```
module EvalMod where
import Expr

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

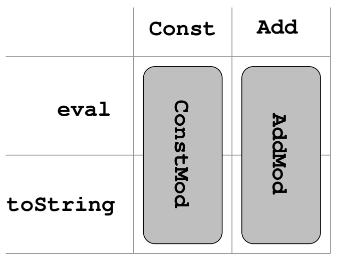
```
module ToStringMod where
import Expr

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

```
module Client where
import Expr
import ToStringMod
import EvalMod

r1 = print (toString e1)
r2 = print (show (eval e1))
r3 = print (toString e2)
r4 = print (show (eval e2))
```

Figure 1: Functional decomposition in Haskell – program P_{fun}



(a) Functional decomposition (program P_{fun})



(b) Constructor decomposition (program P_{data})

Figure 2: Coverage of modules with respect to functions and data constructors

Fig. 2(a) shows a matrix indexed on constructors and functions where the modules of interest have been pictured. For instance, the module `EvalMod` deals with the two constructors but with only one function.

This program architecture makes it easy to modify an existing function since the code to deal with is localized by syntactic module boundaries. It is also easy to add a new function by adding a new module. However, this code is not modular with respect to data constructors. The code corresponding to a given constructor (*e.g.*, `Add`) crosses module boundaries. So, when the data type is extended and a new constructor (*e.g.*, `Mult`) is introduced, each function module must be modified in order to take into account the new constructor.

Figs. 3 and 2(b) describe an alternate code architecture. That structure gathers all the pieces of code related to a given constructor into a single module. For instance, the module `ConstMod` collects the parts of the definition of `eval` and `toString` for the `Const` case. Fig. 2(b) pictures this architecture: modules in the matrix do not cover functions anymore but constructors.

This alternative code is modular with respect to data constructors. Indeed, this program structure makes it easy (modular) to add a new constructor (*e.g.*, `Mult` for a product): the corresponding module is introduced (and `fold` is extended with a new case). However, this code is not modular with respect to functionalities: the code corresponding to a given function (*e.g.*, `eval`) is spread in all constructor modules. So, when a new function is introduced, each module must be modified in order to take the implement the new function.

This illustrates the tyranny of the dominant decomposition in action. Whatever primary program structure is chosen, some extensions will not be modular.

2.2 The Expression Problem

The problem we have described has been subject to many proposals in the context of the so-called Expression Problem (see [29] for a review of some solutions). The expression problem tries to tackle modular extensibility from a language point of view and imposes constraints that are coherent for this point of view. These constraints are the following [28]:

- The extension should come as a separate file/module and should not require to modify existing files/modules.
- The files/modules that were already in the program before the extension should not be recompiled.
- The type system should be able to ensure that the extension is safe.

Several works (for instance those listed in [29]) show that specific features of the host language makes it possible to design a program structure where it is modular to extend the data type, and also modular to extend the functionalities. However, as we will see, these solutions share a drawback: maintenance is not modular. Indeed, successive evolutions tend to break the initial structure [6]. This is not taken into account by the expression problem.

2.3 Extension is only part of the problem

In order to illustrate the deviation from structural regularity with referenced solutions to the expression problem, we consider an incremental development scenario for our example of application. However, we abstract the code of our example and consider a data-type with two constructors `C1` and `C2` (instead of `Const` and `Add`), as well as two functions `f1` and `f2` (instead of `eval` and `toString`).

```
module Expr  where

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

fold a c (Const i)     = c i
fold a c (Add (e1,e2)) = a (fold a c e1) (fold a c e2)

e1 = Add (Add (Const 1,Const 2),Const 3)
e2 = Add (Add (Const 0,Const 1),Const 2)
```

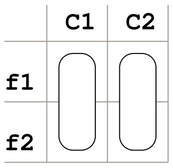
```
module ConstMod (toString,eval) where

 toString x = show x

 eval x = x
```

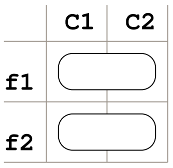
```
module AddMod (toString,eval) where

 toString x y = x ++ "+" ++ y

 eval x y = x + y
```

```
module Client where
import Expr
import ConstMod (eval,toString)
import AddMod (eval,toString)

toString x = fold AddMod.toString ConstMod.toString x

eval x = fold AddMod.eval ConstMod.eval x

r1 = print (Client.toString e1)
r2 = print (show (Client.eval e1))
r3 = print (Client.toString e2)
r4 = print (show (Client.eval e2))
```

Figure 3: Constructor decomposition in Haskell – program P_{data}

The initial program considered in this scenario has an architecture focusing either on function extensibility or on data extensibility, depending on a design choice. These two possible architectures are pictured in the following two diagrams.





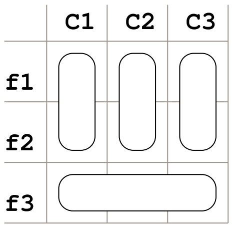
The left hand side diagram means that there is a module[1] for each constructor of the data type and the code of the functions is spread over these modules. This illustrates the situation in the architecture of Fig. 3 and also in the classical object approach (Composite design pattern). The right hand 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 (Fig. 1) and also in the Visitor design pattern.

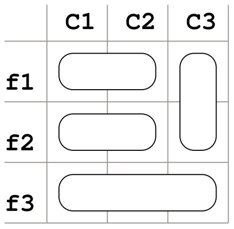
1. Extension : Introduce a new constructor `C3` (for instance `Mult`).
2. Extension : Introduce a new function `f3` (for instance `derive`).
3. Extension : Introduce a new constructor `C4` (for instance `Div`).
4. Extension : Introduce a new function `f4` (for instance `check_div_by_zero`).
5. Maintenance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

Figure 4: Evolution scenario

We now assume that we have chosen a particular solution to extend any axis by providing a new module (for instance, one of the solutions cited in [29]) and we examine what happens with the scenario of Fig 4. The progress of this scenario is illustrated by Figure 5(a) and is detailed below. Grey zones in the figure represent the code which has been added or modified.

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



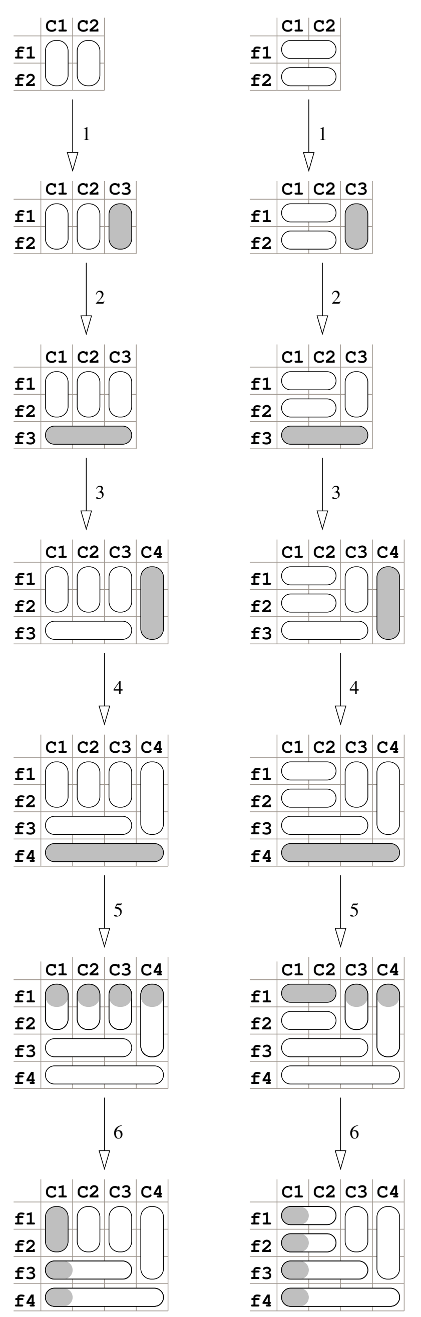


In the left-hand side diagram, the extension of the data type with `C3` is natural, and adding the function `f3` can be done with the chosen specific language feature (in this case, the "module" for `f3` has a different nature from the three other modules of the application).

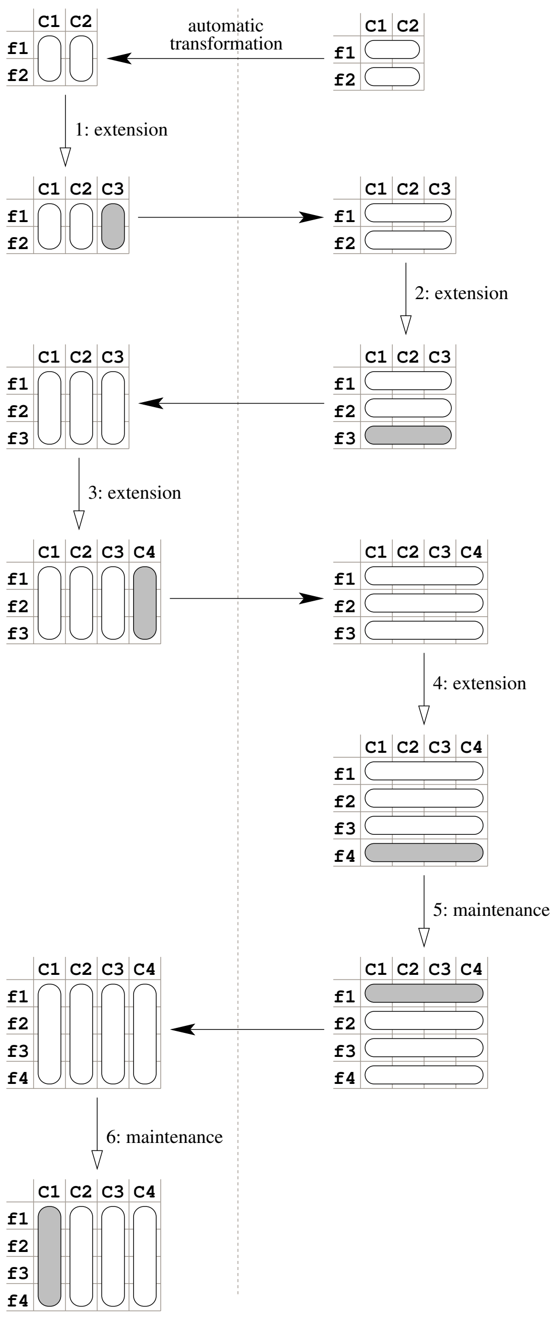
In the right-hand side diagram, we have extended `f1` and `f2` with the chosen specific modular feature to take `C3` into account and then we add `f3`. If we want the extension for `f3` to be fully modular, we have to define `f3` on `C1`, `C2` and `C3` in a single module. (Another solution would have 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: loss of regularity for extensions*).

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



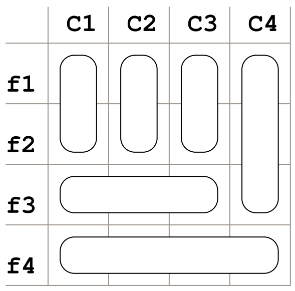
(a) Progress with a language-level solution for two initial architectures.

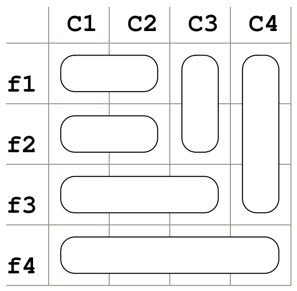


(b) Progress with architecture transformations.

Figure 5: Progress of the evolution scenario in different cases.

Two following extensions (evolutions 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 (evolutions 5 & 6). Now we have to modify `f1` (to correct an error or to cope with a change in its specification). In the (initially) data-centered architecture (left-hand side), the code for `f1` is already spread over several modules in the original program. In the (initially) operation-centered architecture, the code is finally also spread over several modules. This means that we have lost the benefit of the initial modularity: the maintenance is no more modular (*problem 2: loss of the initial modularity properties*).

This is the same for the maintenance of `C1`: in the data-centered architecture, the code has become spread over several modules. Moreover, the number of modules on which the code dealing with a constructor is spread is different for `C1` and `C3`.

The example of this section shows that the technical solutions for modular extensibility are not sufficient for modular maintainability.

3 Views and Transformations

Multiple views [7] aim at solving the problem described in the previous section. We now recast this in our setting.

3.1 Programs and Views

We call *views* of a program two or more textual representations of programs that have the same behavior (they are semantically equivalent in a given calculus). For instance, P_{data} and P_{fun} are two views of the same program (this is justified later in the paper).

3.2 A Solution to the Modular Maintenance Problem

We illustrate the use of views to solve the problem of modularity in evolutions by building an automated transformation of P_{data} into P_{fun} and its reverse transformation.

With such a tool, the programmer can choose the view in which the evolution he has to implement is modular. Fig. 5(b) illustrates the scenario given in section 2.3 with this approach. For instance, when the programmer wants to add a new constructor (evolution 1), the program is presented in the data-centered view; when the programmer wants to add a new function (evolution 2), the program is presented in the operation-centered view. Since none of these evolutions has to be made transversely to the considered axis of decomposition, the views of interest always keep a regular architecture. This approach enables to solve the problems 1 and 2 described in section 2.3.

This approach also has the following advantages compared to solutions to the expression problem:

- It does not rely on a particular programming language. As soon as two alternative programming structures can be expressed in a language, the corresponding transformation can be defined.

- The programmer who implements the evolution does not have to learn a new language or possibly complex language features. Of course, he has to cope with several views.
- The programmer does not have to learn a new type system. In particular, if the programming language is strongly typed, the different views are also strongly typed and the types they introduce are closely related. In this case, typing issues boil down to verify once for all that the program transformations do not break typing.
- The approach is not limited to the data-centered view *versus* the function-centered view. It is not even limited to two views.

Of course, this approach is not free from disadvantages:

- The programmer who implements the evolution has to cope with several views.
- The transformation has to be implemented, which requires some work from a "transformation designer" and a supporting tool.

3.3 Refactoring tools to navigate between views

Developing a program transformation from scratch is not easy. Refactoring tools provide a simple, high-level way to transform programs and are available for several kinds of languages. For this reason, we have chosen to explore the use of a refactoring tool to build our example of transformation.

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [13]

Given our definition of programs and views, a code refactoring changes the views of the considered program (a refactoring tool enables to pass from one view to another). Griswold [14] shows that refactoring tools can be used to change a function-centered architecture into a data-centered architecture. In particular, he exemplifies his technique with the program Parnas [22] had employed to illustrate different architectures for a same program, each with different advantages and disadvantages. Griswold uses refactoring tools to improve the structure of code. We adopt a more dynamical point of view, where an improvement is not absolute, but driven by a temporary need: once the driving evolution is implemented, we may need to revert to the initial architecture.

Some refactoring tools are provided with most popular IDEs, for several mainstream languages (Java in Eclipse [3] and NetBeans [1], C++, C#, VB in Visual Studio [5]). Refactoring tools have also found an interest in the academic community, and some tools which are based on sound foundations have been proposed, for instance for Haskell and Erlang [20], C [25], Smalltalk [23] or Lisp [15]. Finally, some refactoring tools have been specifically designed to support views in an object-oriented context [7, 24] (see section 6.3).

It is important to note that some refactoring tools are not sound. In particular, with the Eclipse refactoring tool for Java, after refactoring operations have been applied, the user has to fix broken code himself [3, 2]. For this reason, we focus in this paper on one of the refactorers which provides operations whose principles have been proved correct: the Haskell Refactorer (HaRe) [4, 19, 26]. The formalism used in [19] is a λ-calculus with *let-rec* with a mixed call-by-name/call-by-need strategy. A relation of equivalence based on the reduction rules of that calculus expresses the behavior preservation which is used to prove the correctness of the operations.

Before applying a refactoring operation, the Haskell Refactorer checks that the conditions to ensure its correctness are verified. For instance, it checks that a renaming does not introduce a name clash. When these conditions are not verified, the refactorer does not apply the changes and explains why.

4 Implementation of an Architecture Transformer with a Refactorer

In this section, we show how P_{fun} can be transformed into P_{data} by using the Haskell Refactorer, and the other way around. We also show how the chain of refactoring operations can be automated.

4.1 Decomposing the transformation into refactoring operations

We describe in this section the steps to transform P_{fun} into P_{data} with the Haskell Refactorer. As already said, each of these operations checks that the conditions that make the refactoring correct are verified.

In the following, we consider only the `eval` function. The chain of operations is the same on the `toString` function. Here is the code which is considered in P_{fun}:

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

```
--from Client
r2 = print (show (eval e1))
r4 = print (show (eval e2))
```

We now present the transformation steps. All the fragments of code we show in this section are the result of the use of the refactorer (except for the comments introduced by `--` and some empty lines which are skipped).

1. We introduce new local definitions for the code of each constructor case (Haskell Refactorer's *Introduce New Definition* operation).

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

2. In each of these new definitions, depending on the constructor concerned (`Const` or `Add`), we generalize either the arguments of the constructor, or the recursive calls of `eval` on these arguments (*Generalise def*).

```
eval (Const i)     = evalConst i
  where
    evalConst x = x
eval (Add (e1,e2)) = evalAdd (eval e1) (eval e2)
  where
    evalAdd x y = (x) + (y)
```

3. We lift the new local definitions to make them global (*Lift Definition to Top Level*).

```
eval (Const i)     = evalConst i

eval (Add (e1,e2)) = evalAdd (eval e1) (eval e2)

evalAdd x y = (x) + (y)

evalConst x = x
```

4. In the definition of `eval`, we generalize `evalConst` and `evalAdd` (*Generalise def*).

```
eval a c (Const i)     = c i

eval a c (Add (e1,e2)) = a (((eval a) c) e1) (((eval a) c) e2)
```

By applying that operation, all the existing references to `eval`, in particular in the module Client, are transformed to take the new parameters into account. In the body of `eval`, references to `eval` are replaced by `((eval f_Add) f_Const)`. In the module `Client`, references to `eval` are replaced by `eval eval_gen_1 eval_gen` where `eval_gen` is defined by `evalConst` in `EvalMod` and `eval_gen_1` is defined by `evalAdd`.

```
-- from EvalMod module
eval_gen_1 = evalAdd

eval_gen = evalConst
```

```
-- from Client module
r2 = print (show (eval eval_gen_1 eval_gen e1))
r4 = print (show (eval eval_gen_1 eval_gen e2))
```

5. We rename `eval` into `fold1` (*Rename*).

```
-- from EvalMod module
fold1 a c (Const i)      = c i

fold1 a c (Add (e1,e2)) = a (((fold1 a) c) e1) (((fold1 a) c) e2)
```

```
-- from Client module
r2 = print (show (fold1 eval_gen_1 eval_gen e1))
r4 = print (show (fold1 eval_gen_1 eval_gen e2))
```

6. We introduce a new definition named `eval` for the expression `fold1 eval_gen_1 eval_gen e1` in the module `Client`, we lift it at the top level, and abstract it over `e1`.

```
r2 = print (show (eval e1))

eval x = fold1 eval_gen_1 eval_gen x

r4 = print (show (fold1 eval_gen_1 eval_gen e2))
```

7. We fold the definition of `eval` to make it appear in `r4` (*Fold Definition*).

```
r2 = print (show (eval e1))

eval x = fold1 eval_gen_1 eval_gen x

r4 = print (show (eval e2))
```

8. We unfold the occurrences of `eval_gen` and `eval_gen_1` (*Unfold def*) and we remove their definitions (*Remove def*).

```
eval x = fold1 evalAdd evalConst x
```

9. We move the definitions of `evalConst` and `evalAdd` from `EvalMod` to `ConstMod` and `AddMod` and rename them into `eval`.

```
r2 = print (show (Client.eval e1))

eval x = fold1 AddMod.eval ConstMod.eval x

r4 = print (show (Client.eval e2))
```

10. We move the definition of `fold1` into the `Expr` module. The module `EvalMod` is now empty.

11. We remove useless imports of module `EvalMod` in the module `Client` (*Clean imports*).

In practice, after this sequence of refactorings, `fold1` and the `fold2` we get from the transformation of `toString` are α-equivalent. One of them may be deleted to find the exact P_{data} described in section 2.1 (this seems not to be supported by the Haskell Refactorer at the moment).

The layout is also not exactly the same as expected (*e.g.*, there are additional pairs of parenthesis).

Soundness. If we use behavior preserving refactoring operations, then the chain of refactoring operations is also behavior preserving (and P_{fun} is equivalent to P_{data}). Some Haskell Refactorer's operations's principles have been shown to be correct [26, 19]. However, there are some bugs left in the implementation, and not all the available operations have been proved correct, including some of the ones we are using[2]

Reverse transformation. A simple approach to build the reverse transformation would be to use the inverse of each operation used in the $P_{fun} \to P_{data}$ transformation in the reverse order (since $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$). However, the Haskell Refactorer does not provide an inverse for each operation, so our reverse transformation cannot be automatically derived from the first transformation.

We do not detail here the reverse transformation (the script is given in Figs 8 and 9). The key steps are to unfold the instances of `ConstMod.eval`, `ConstMod.toString`, `AddMod.eval`, `AddMod.toString` and to transform `eval` and `toString`, which are defined by calls to `fold_1` and `fold_2`, into plain recursive function definitions. This particular step is done by using the *Generative Fold* operation of the Haskell Refactorer (*folding* in [10], see [9]). Note that to use the generative fold, a preliminary step has to be done by hand: a function definition must be duplicated into a comment. We have introduced in the Haskell Refactorer a feature to support this duplication into comments in order to make the whole chain supported by the tool and to be able to automate it.

We also have had to add some features to the refactorer to simplify the code we get after unfolding some functions defined by equations with patterns in order to reach the code of P_{data} (see appendix B for the list of operations we have implemented into HaRe).

4.2 Automation of the process.

An engineering effort has been necessary to automate the transformation since the API of the Haskell Refactorer does not match our needs. In particular, HaRe is designed to be used interactively with text editors as Emacs and the parameters of the operations are cursor positions (line and column numbers) in source files. For this reason we have developed functions to locate sub-expressions of interest in files before calling HaRe operations with the computed parameters. This allows us to provide new interfaces for HaRe operations (at least for those we needed) that do not rely on the particular layout in the source files (see appendix A for the list of interfaces to HaRe operations we have implemented). Like original HaRe operations, our interfaces to HaRe operations are available as Emacs-Lisp functions in addition to interactive menu entries. This allows to invoke them in Emacs-Lisp programs.

Our transformations can thus be expressed by Emacs-Lisp programs. Since these programs are reduced to sequences of operations with side-effects, we call them *scripts*. Fig. 7 shows the script of the transformation $P_{fun} \to P_{data}$ (we assume the definitions of Fig. 6 are evaluated first).

```
(defvar f1         "eval"           )
(defvar f2         "toString"       )
(defvar c1         "Const"          )
(defvar c2         "Add"            )
(defvar f1mod      "EvalMod"        )
(defvar f2mod      "ToStringMod"    )
(defvar f1c1       (concat f1 c1)   )
(defvar f1c2       (concat f1 c2)   )
(defvar f2c1       (concat f2 c1)   )
(defvar f2c2       (concat f2 c2)   )
(defvar c1mod      (concat c1 "Mod"))
(defvar c2mod      (concat c2 "Mod"))
(defvar f1reducer  "fold1"          )
(defvar f2reducer  "fold2"          )
(defvar dummyc1    "c"              )
(defvar dummyc2    "a"              )
(defvar clientmod  "Client"         )
```

Figure 6: $P_{fun} \to P_{data}$ script preliminary definitions

[2]This problem is different from the problem in Eclipse for which the answer is : *"If the refactoring causes problems in other methods, these are ignored and you must fix them yourself after the refactoring."* [3] and *"Note that some modifications you make to the method, such as adding a parameter or changing a return type, may cause the refactored code to contain compiler errors because Eclipse doesn't know what to enter for those new parameters."* [2].

```
;; 1 - introduce new definitions
(haskell-refac-exhibitFunction f1 c1 f1c1 f1mod)
(haskell-refac-exhibitFunction f1 c2 f1c2 f1mod)
(haskell-refac-exhibitFunction f2 c1 f2c1 f2mod)
(haskell-refac-exhibitFunction f2 c2 f2c2 f2mod)

;; 2 - abstract some arguments in these definitions
(haskell-refac-generalise f1 c1 f1c1 f1mod "0" "x" "Curried" "OtherType")
(haskell-refac-generalise f2 c1 f2c1 f2mod "0" "x" "Curried" "OtherType")
(haskell-refac-generalise f2 c2 f2c2 f2mod "1" "y" "UnCurried" "RecType")
(haskell-refac-generalise f2 c2 f2c2 f2mod "0" "x" "UnCurried" "RecType")
(haskell-refac-generalise f1 c2 f1c2 f1mod "1" "y" "UnCurried" "RecType")
(haskell-refac-generalise f1 c2 f1c2 f1mod "0" "x" "UnCurried" "RecType")

;; 3 - lift the new functions to the top-level
(haskell-refac-makeGlobalOfLocalIn f1 f1c1 f1mod)
(haskell-refac-makeGlobalOfLocalIn f1 f1c2 f1mod)
(haskell-refac-makeGlobalOfLocalIn f2 f2c1 f2mod)
(haskell-refac-makeGlobalOfLocalIn f2 f2c2 f2mod)

;; 4 - abstract the functions of interest from the introduced functions
(haskell-refac-generaliseIdent f1 f1mod f1c1 dummyc1)
(haskell-refac-generaliseIdent f1 f1mod f1c2 dummyc2)
(haskell-refac-generaliseIdent f2 f2mod f2c1 dummyc1)
(haskell-refac-generaliseIdent f2 f2mod f2c2 dummyc2)

;; 5 - rename the functions of interest (they have become traversal functions)
(haskell-refac-renameToplevel f1 f1mod f1reducer)
(haskell-refac-renameToplevel f2 f2mod f2reducer)

;; 6 - reconstruct the functions of interest as calls to the traversal functions
;; with appropriate arguments
(haskell-refac-newDefFunApp f1reducer "3" f1 clientmod)
(haskell-refac-newDefFunApp f2reducer "3" f2 clientmod)
(haskell-refac-makeGlobalOfLocal f1 clientmod)
(haskell-refac-makeGlobalOfLocal f2 clientmod)
(haskell-refac-generaliseIdent f1 clientmod "e1" "x")
(haskell-refac-generaliseIdent f2 clientmod "e1" "x")

;; 7 - propagate the new definitions
(haskell-refac-foldToplevelDefinition f1 clientmod)
(haskell-refac-foldToplevelDefinition f2 clientmod)

;; 8 - unfold dummy definitions and remove them
(haskell-refac-unfoldInstanceIn (concat f1 "_gen") f1 clientmod)
(haskell-refac-unfoldInstanceIn (concat f1 "_gen_1") f1 clientmod)
(haskell-refac-unfoldInstanceIn (concat f2  "_gen") f2 clientmod)
(haskell-refac-unfoldInstanceIn (concat f2  "_gen_1") f2 clientmod)

(haskell-refac-removeDefCmd (concat f1 "_gen") f1mod)
(haskell-refac-removeDefCmd (concat f1 "_gen_1") f1mod)
(haskell-refac-removeDefCmd (concat f2 "_gen") f2mod)
(haskell-refac-removeDefCmd (concat f2 "_gen_1") f2mod)

;; 9 - move business functions to the appropriate modules
(haskell-refac-moveDefBetweenModules f1c1 f1mod c1mod)
(haskell-refac-moveDefBetweenModules f1c2 f1mod c2mod)
(haskell-refac-moveDefBetweenModules f2c1 f2mod c1mod)
(haskell-refac-moveDefBetweenModules f2c2 f2mod c2mod)

;; rename the business functions to make them share the same name
(haskell-refac-renameToplevel f1c1 c1mod f1)
(haskell-refac-renameToplevel f1c2 c2mod f1)
(haskell-refac-renameToplevel f2c1 c1mod f2)
(haskell-refac-renameToplevel f2c2 c2mod f2)

;; 10 - move the functions into the relevant modules
(haskell-refac-moveDefBetweenModules f1reducer f1mod "Expr")
(haskell-refac-moveDefBetweenModules f2reducer f2mod "Expr")

;; 11 - clean imports
(haskell-refac-cleanImportsCmd clientmod)
```

Figure 7: $P_{fun} \rightarrow P_{data}$ transformation script

```
;; reverse 10 -
(haskell-refac-moveDefBetweenModules f1reducer "Expr" f1mod)
(haskell-refac-moveDefBetweenModules f2reducer "Expr" f2mod)

;; reverse 9 - use specific names for business functions
(haskell-refac-renameToplevel f1 c1mod f1c1)
(haskell-refac-renameToplevel f1 c2mod f1c2)
(haskell-refac-renameToplevel f2 c1mod f2c1)
(haskell-refac-renameToplevel f2 c2mod f2c2)

;; reverse 9 - move business functions to the original modules
(haskell-refac-moveDefBetweenModules f1c1 c1mod f1mod)
(haskell-refac-moveDefBetweenModules f1c2 c2mod f1mod)
(haskell-refac-moveDefBetweenModules f2c1 c1mod f2mod)
(haskell-refac-moveDefBetweenModules f2c2 c2mod f2mod)

;; move f1 f2 to f1mod f2 mod
(haskell-refac-moveDefBetweenModules f1 clientmod f1mod)
(haskell-refac-moveDefBetweenModules f2 clientmod f2mod)

(haskell-refac-cleanImportsCmd clientmod)

;; reverse 4/5/6/7 - transform a call to fold into a recursive function

;; prepare the generative fold operations :
;; the equations for functions of interest are saved into  comments
(haskell-refac-duplicateIntoComment f1 f1mod)
(haskell-refac-duplicateIntoComment f2 f2mod)

;; unfold the use of the traversal function
;; in the definition of functions of interest
(haskell-refac-unfoldInstanceIn f1reducer f1 f1mod)
(haskell-refac-unfoldInstanceIn f2reducer f2 f2mod)

;; unfolding has produced case expressions that have
;; to be simplified
(haskell-refac-simplifyCasePattern f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc2 f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc2 f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc2 f1 f1mod)
(haskell-refac-removeLocalDef dummyc2 f1 f1mod)

(haskell-refac-simplifyCasePattern f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc1 f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc1 f1 f1mod)
(haskell-refac-unfoldInstanceIn dummyc1 f1 f1mod)
(haskell-refac-removeLocalDef dummyc1 f1 f1mod)

(haskell-refac-simplifyCasePattern f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc2 f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc2 f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc2 f2 f2mod)
(haskell-refac-removeLocalDef dummyc2 f2 f2mod)

(haskell-refac-simplifyCasePattern f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc1 f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc1 f2 f2mod)
(haskell-refac-unfoldInstanceIn dummyc1 f2 f2mod)
(haskell-refac-removeLocalDef dummyc1 f2 f2mod)

;; the case expressions introduced by the unfolding
;; have to be transformed into equations
(haskell-refac-caseToEq f1 f1mod)
(haskell-refac-caseToEq f2 f2mod)

;; fold the use of the traversal function in the body of the
;; functions of interest in order to get a recursive definition
;; (without a call to the traversal function)
(haskell-refac-generativeFold f1reducer "3" f1mod)
(haskell-refac-generativeFold f1reducer "3" f1mod)
(haskell-refac-generativeFold f2reducer "3" f2mod)
(haskell-refac-generativeFold f2reducer "3" f2mod)

;; comments introduced for the generative fold can be deleted
(haskell-refac-rmCommentBefore f1 f1mod)
(haskell-refac-rmCommentBefore f2 f2mod)

;; the traversal functions can be deleted
(haskell-refac-removeDefCmd f1reducer f1mod)
(haskell-refac-removeDefCmd f2reducer f2mod)
;; end of the transformation of the call to fold into a recursive function.
```

Figure 8: $P_{data} \rightarrow P_{fun}$ transformation script (first part)

```
;; reverse 1/2/3 - replace calls to the business functions by their bodies (unfold)
;; and delete the business functions.
(haskell-refac-unfoldInstanceIn f1c1 f1 f1mod)
(haskell-refac-unfoldInstanceIn f1c2 f1 f1mod)
(haskell-refac-unfoldInstanceIn f2c1 f2 f2mod)
(haskell-refac-unfoldInstanceIn f2c2 f2 f2mod)

(haskell-refac-removeDefCmd f1c1 f1mod)
(haskell-refac-removeDefCmd f1c2 f1mod)
(haskell-refac-removeDefCmd f2c1 f2mod)
(haskell-refac-removeDefCmd f2c2 f2mod)
```

Figure 9: $P_{data} \rightarrow P_{fun}$ transformation script (end)

5 Usage and Future Work

This section discusses the possibility of using views with the tool we demonstrated here and proposes future works to improve this use.

Development of the transformation. The first problem with our setting is that the transformation has to be implemented. We could ease this task by several means:

- Compute automatically the transformation based on the hints of the designer, as done in [7] and [24] or provide a more high level transformation specific language.
- Provide a list of examples to be used as transformation patterns.
- Once a transformation is defined, generate automatically the inverse transformation.
- Record the sequence of commands used in an interactive refactorer (as Eclipse does [2]).

Maintenance of the transformation. As the program evolves, the transformation may need to evolve too. We could study how an evolution impacts a transformation. Some simple examples seem to be directly tractable: for instance, if the evolution to be implemented is the renaming of a function, we can imagine that the refactoring tool that propagates the renaming could also propagate it into the transformation script.

Duration. The transformation of our example takes about 15 seconds (on a 2.8 GHz Intel Pentium R processor) and 7 seconds for the reverse transformation. This is longer than the compile time of the program (less than 1 sec.). We can suppose that the transformations can be integrated into the build process that takes place once a new version of the code is released, so that when a programmer has to implement an evolution, all the designed views are available. Of course, this supposes all the views have been designed together with the initial program.

Availability of a tool for a given language. We have chosen to use an existing refactoring tool instead or rebuilding one. We have produced 615 LOC to implement missing refactoring operations and 1160 LOC for the new interface. This has to be compared to the size of a refactoring tool (11 KLOC for Arcum [24]). So our approach depends on the tools available for the language of interest.

Failures From an operational point of view, as each operation of the chain requires some pre-conditions to be satisfied, it may occur that the user is informed that the transformation cannot be achieved only after some operations have been applied. In order to inform the user as soon as possible (statically instead of dynamically) that the transformation cannot be applied, we could compute the pre-condition of the transformation, for instance by following the work of Kniesel and Koch [17]. This would imply to provide a formal description of the pre-conditions and effects (post-conditions or Kniesel and Koch's backward-descriptions) of all the operations used.

6 Conclusion

6.1 Contributions.

The contributions of this report are the following:

- We give an example of use of multiple views in a functional programming language. That example comes from the expression problem.
- We implement that transformation with an existing refactorer, the Haskell Refactorer, and we automate it. To be able to do this, we extend the interface of the refactorer and add a few operations in it.

6.2 Comparison to view tools

Compared to some other multiple views implementations (in particular [7], [24]) and to the implementation of [15], our approach has the following pros and cons:

- $\oplus$ We rely on a previously existing refactorer. We had to implement few lines of code to adapt it to support views. The engineering effort is very small compared to building a dedicated tool. Moreover, the basic operations are already proved correct (yet not all of them, and not the concrete implementation).
- $\ominus$ The expression of the transformation is rather low level. It is imperative (as in [15]) while in [7] and [24] it is declarative. Moreover, it is not automatically invertible.
- $\neq$ We focus on dealing with fragments of business code while [7] focuses on classes and method interactions and [14] focuses on data-structure encapsulation (at least in the example from Parnas).

6.3 Related Work

Offering to the programmer (or designer or specifier) an appropriate view is not a new idea. Here is a review of some relevant related work.

Before Wadler introduced the expression problem [28], he had already proposed the notion of views [27]. His view feature makes it possible to use pattern matching with different representations of a data structure. At compile time different views of a data structure are defined and their relationships are specified by rewriting rules. At run-time, a single data structure is maintained and pattern matching on different representations are translated to accessor functions that compute a (partial) view from another one. This is closely related to our problem, but we do not focus on data structures but on code structures and we extend and maintain programs before run-time.

Griswold [14, 15] shows that elementary refactoring operations can be chained to provide architecture transformations. However, as already said in section 3.3, while Griswold's goal is to improve the structure of code, our goal is to dynamically adopt a structure which is convenient for a given task.

We share with Black and Jones [7] the same motivation and the idea that multiple views solves the tyranny of dominant decomposition problem. However, the techniques are rather different. In their implementation, the programmer describes properties of the fragments of code which are used to compute the views. In Shonle *et al.* [24], it is the transformation which is described. It is described declaratively by rewriting rules. Both works handle mainly object oriented language concepts (classes, methods, field accesses).

Functional programs are prone to be transformed. Numerous program transformations have been proposed. Some comes in pairs, or are invertible. For instance, Danvy have studied relationship between the continuation passing style and the direct style [11]. In general a program transformation is not invertible. Forster et al. [12] have proposed a domain specific language to define invertible transformations. Once a transformation is specified, the inverse transformation is automatically derived. This enables them, for instance, to share data between several applications that require different representations. Note that, a view can be partial and the original representation can be required in order to transform a modified view back to its original form by injecting the updated data. This work has been extended in order to deal with classes of equivalent representations (*e.g.*, two lists of associations (key,value) can be equivalent even if the order of the pairs are different) [8].

Views have also been introduced at the specification level. For instance, Jackson [16] shows how to compose Z specifications on different views of the same state. At the specification level, the composition is not computational (it does not require transformation) but declarative: invariants relate the different views of the state.

Literate programming [18] proposes to invert the role of code and comments (comments becomes the main view of a program and is commented by a few pieces of code). More importantly, literate programming enables to decompose and reorder pieces of programs. This way, the literate view for human has to be transformed into the code view for the compiler. However, the transformation is single way. This does not help for the maintenance or the evolution problem (since the code is not transformed but only reordered) but this proves its is important to present alternative views for the programmer.

Web Sites

[1] Refactoring – NetBeans Wiki. `http://wiki.netbeans.org/Refactoring`.

[2] P. Deva. Explore refactoring functions in Eclipse JDT. `http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/`, Nov. 2009.

[3] D. Gallardo. Refactoring for everyone. `http://www.ibm.com/developerworks/library/os-ecref/`, Sept. 2003.

[4] HaRe – the Haskell Refactorer. `http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`, 2003-2010.

[5] W. T. Software. Visual Assist X for Visual Studio. `http://www.wholetomato.com/`.

References

[6] L. A. Belady and M. M. Lehman. Programming system dynamics, or the meta-dynamics of systems in maintenance and growth. Technical report, IBM T.J. Watson Research Center, 1971.

[7] A. P. Black and M. P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, 2004.

[8] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In Necula and Wadler [21], pages 407–419.

[9] C. M. Brown. *Tool Support for Refactoring Haskell Programs.* PhD thesis, University of Kent, Sept. 2008.

[10] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, Jan. 1977.

[11] O. Danvy. Back to direct style. In *ESOP'92: Selected papers of the symposium on Fourth European symposium on programming*, pages 183–195, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[14] W. B. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, Department of Computer Science and Engineering, University of Washington, July 1991.

[15] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[16] D. Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.

[17] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3):9–51, Aug. 2004. Special Issue on Program Transformation.

[18] D. E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

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

[20] H. Li and S. Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, San Francisco, CA, USA, Jan. 2008.

[21] G. C. Necula and P. Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, CA, USA, Jan. 7-12, 2008.* ACM, 2008.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[23] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(Issue 4):253–263, 1997.

[24] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 175–184, New York, NY, USA, 2007. ACM.

[25] D. Spinellis. CScout: A refactoring browser for C. *Science of Computer Programming*, 75(Issue 4):216–231, Apr. 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[26] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation.* ACM SIGPLAN, Jan. 2008.

[27] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313, 1987.

[28] P. Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.

[29] 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 Interfaces added to Refactoring Operations

Introduce new def.

- `haskell-refac-exhibitFunction f c n m`

  In the equation concerning the constructor `c` in the definition of `f` in the module `m`, create a new local definition for the right hand side of the equation as `n` .

- `haskell-refac-newDefFunApp f n f' m`

  Find an application of the identifier `f` to `n` arguments in the module `m` and create a new local definition for that application as `f'`.

Generalise def.

- `haskell-refac-generalise f c f' m n x curry "OtherType"`

  In the module `m`, in the equation concerning the constructor `c` of the definition of `f`, let v be the `n`th argument of the constructor `c` in the pattern of the equation, generalise v in the local definition of `f'` and name `x` the new argument.

  The flag `curry` indicates whether the arguments of the constructor are curried or not to count the arguments.

- `haskell-refac-generalise f c f' m n x curry "RecType"`

  In the module `m`, in the equation concerning the constructor `c` of the definition of f, let v be the `n`th argument of the constructor `c` in the pattern of the equation, generalise *an application of f to v* in the local definition of `f'` and name `x` the new argument.

- `haskell-refac-generaliseIdent f m v x`

  In the definition of `f` in the module `m`, generalise the variable `v` and name the new parameter `x`.

Lift def to top level.

- `haskell-refac-makeGlobalOfLocalIn f d m`

  Lift the definition of `d` at the top-level. `d` is declared inside the definition of `f` in the module `m`,

Rename.

- `haskell-refac-renameTopLevel f m f'`

  Rename `t` declared at the top-level in the module `m` into `f'`.

Move def to another module.

- `haskell-refac-moveDefBetweenModules f m m'`

  Move the top-level definition of `f` from module `m` to module `m'`.

Unfold def./Fold def.

- `haskell-refac-unfoldInstanceIn d f m`

  Replace an instance of the identifier `d` by the boy of its definition, in the body of `f` in module `m`. If possible, a beta-reduction is applied (see the corresponding HaRe operation).

- `haskell-refac-foldToplevelDefinition f m`

  Fold the definition of function `f` of module `m` (see the corresponding HaRe operation).

Generative Fold

- `haskell-refac-generativeFold f "`*i*`" m`

  Select an application of the identifier `f` to *i* arguments in m and apply HaRe Generative Fold operation on it (a comment must be present before the affected declaration, see the HaRe operation).

Remove def.

- `haskell-refac-removeDefCmd f m`

  Remove the definition of `f` in the module `m`. `f` must not be used elsewhere.

- `haskell-refac-removeLocalDef d f m`

  Remove the definition of `d` which is local to `f` in the module `m`. `f` must not be used elsewhere.

Clean imports, Remove from export.

- `haskell-refac-cleanImportsCmd m`

  Call the Clean imports operation on the module `m` (remove the useless imports).

- `haskell-refac-rmFromExports f m`

  Remove `f` from the explicit exports of the module `m`. `f` must not be used in an other module.

B Added Refactoring Operations

Simplify a *case* pattern matching. This operation transforms the first code below into the second code below.

```
case (e1, e2, e3) of
         (y, p11, p12) -> b1
         (y, p21, p22) -> b2
```

```
let y = e1
in case (e2, e3) of
            (p11, p12) -> b1
            (p21, p22) -> b2
```

The patterns and the matched expression have to be tuples.

The refactoring applies when there is a same identifier at the same position in all the pattern tuples.

Input : select the whole case expression.

`haskell-refac-simplifyCasePattern f m`

Apply the above operation on a *case* expression in the definition of `f` in the module `m`.

Transform a *case* pattern matching into equations. Transforms the first code below into the second code below.

```
f x = case (x) of
            p1 -> e1
            p2 -> e2
```

```
f p1 = e1
f p2 = e2
```

The pattern in the initial case has to be reduced to a variable which occurs as a parameter of the function.

A second version of this function is available for pattern matching on pairs:

```
f x y = case (x,y) of
            (p11,p12) -> e1
            (p21,p22) -> e2
```

```
f p11 p12 = e1
f p21 p22 = e2
```

`haskell-refac-caseToEq f m`
`haskell-refac-caseToEq2 f m`

Select the case expression which is at the top-level of the body of the declaration of `f` in the module `m` and transform it into a set of equations.

Copy a declaration into a comment. Makes a copy of a declaration into a comment placed just above the declaration (to be used before applying Generative Fold).

`haskell-refac-duplicateIntoComment f m` Copy of the declaration of `f` of the module `m`.

Remove a comment. Deletes a comment.

`haskell-refac-rmCommentBefore f m` Delete the comment occurring before the declaration of `f` at the top-level of the module `m`.