HAL
archives-ouvertes.fr

Program Transformation based Views for Modular Maintenance

Akram Ajouli, Julien Cohen, Rémi Douence

To cite this version:

Akram Ajouli, Julien Cohen, Rémi Douence. Program Transformation based Views for Modular Maintenance. Troisièmes journées nationales du GDR GPL, Jun 2011, Lille, France. pp.218-219. hal-00670378

HAL Id: hal-00670378
https://hal.archives-ouvertes.fr/hal-00670378

Submitted on 15 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Program Transformation based Views for Modular Maintenance

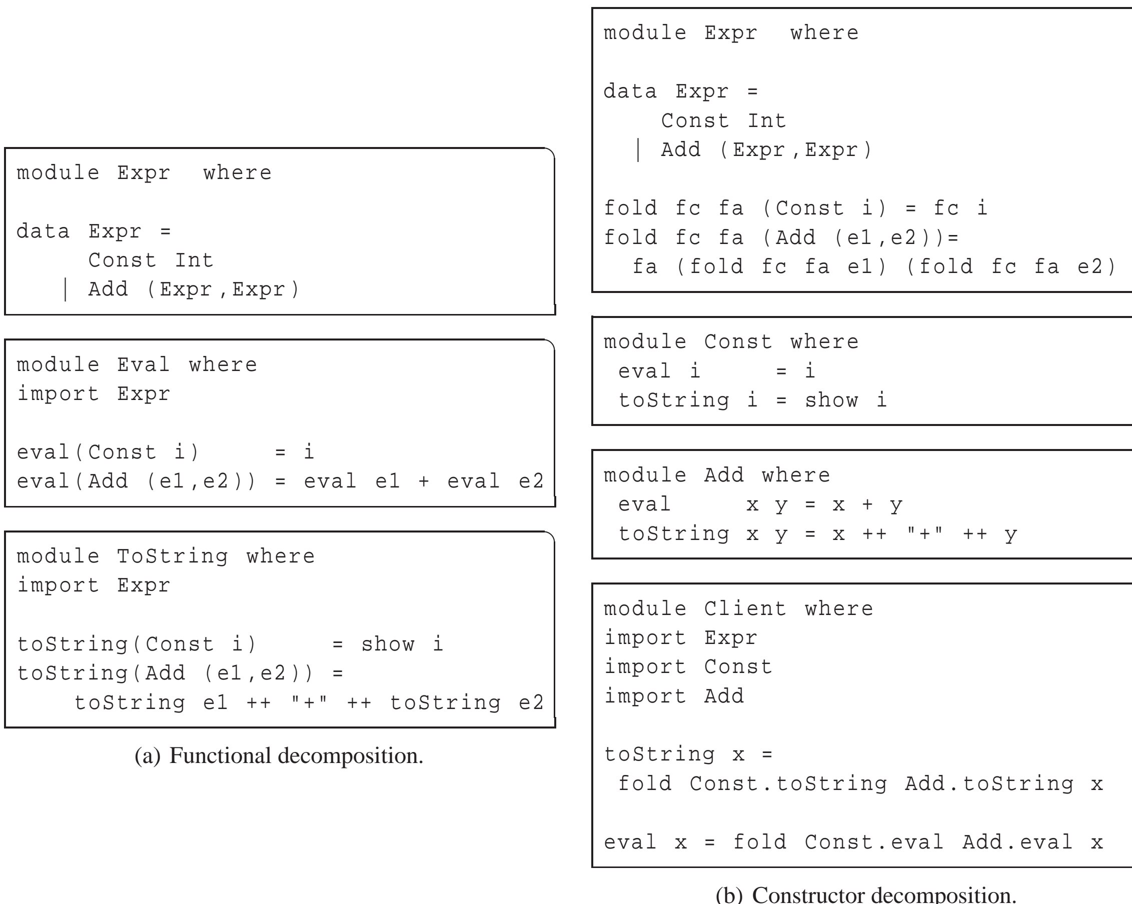
Akram AJOULI, Julien COHEN, and Rémi DOUENCE

ASCOLA group: EMN-INRIA, LINA

1 Context

Since modular programming is a practical solution for separation of concerns, it participates in reducing development time and it favors maintainability and reuse [5]. Despite of these benefits, modularity does not resolve the classical expression problem [7] and more generally the tyranny of the dominant decomposition [6]: evolutions are modular only on the principal axis of decomposition.

This is illustrated by the two possible structures for a same program given in Fig. 1. The architecture given in Fig. 1(a) is modular with respect to functions (it is modular to add or modify a function), its dominant decomposition is function-oriented. The architecture given in Fig. 1(b) is modular with respect to data constructors (when the data-structure is extended or adapted, it is modular to extend or update the corresponding functions), its dominant decomposition is data-oriented. In object oriented programming, the same duality is observed between the Visitor and the Composite design patterns.



(a) Functional decomposition.

(b) Constructor decomposition.

Fig. 1. Two alternative architectures for a same Haskell program.

Whatever structure is chosen, some evolutions will be cross-cutting (non modular). In particular, adapting a function in the data-oriented architecture is cross-cutting and vice versa.

To solve this problem, a practical solution would be to be able to choose the architecture each time one has an evolution to implement. This corresponds to a concept of views of programs where each view expresses the whole program, but with a different code structure (as in [1]). With such a solution, a programmer can implement an evolution in the view which is the most convenient. The practicability of this solution depends on the availability of a tool (possibly based on program transformations) to be able to pass from one view to another (from one code structure to another).

2 Contributions

We provide a prototype tool for the Haskell language to support the concept of views described above [2]. Our tool allows to build transformations to switch Haskell programs from one view to another. We do this by providing high-level interfaces to a refactoring tool for Haskell (HaRe [4]): transformations are built by chaining elementary operations of refactoring. Moreover, the refactoring tool ensures that the transformations preserve the semantics of the programs: since each elementary refactoring preserves the semantics, the whole transformation also does.

3 Challenges

To be practicable, our proposal must respond to the following challenges:

Automatic inference of program transformations. A disadvantage of our proposal is that the transformation has to be defined imperatively, which is tedious. To solve this, we must provide automatic transformation inference based on constraints expressed by the user. This would also avoid to have to maintain the transformations as the programs evolve.

Invertibility of program transformations. Once a first transformation is set up (automatically or not), the inverse transformation could be computed from the first one. However, all refactoring operations are not invertible and we cannot tell now what architecture transformations can be built with invertible elementary transformations.

Minimum precondition. As static type checking has advantages over dynamic typing with respect to assistance for development, a minimal precondition analysis performed on transformations can help to set up transformations and check their properties [3]. Thus, one could have the guaranty that a transformation will succeed before performing it.

References

1. Andrew P. Black and Mark P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, 2004.
2. Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem in a Functional Language. 19 pages, Research Report hal-00481941, version 2, January 2011.
3. Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3):9–51, Aug. 2004. Special Issue on Program Transformation.
4. Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, San Francisco, CA, USA, Jan. 2008.
5. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.
6. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Jr. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.
7. Philip Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.