Transformation of higher-order programs

William Blum

November 4, 2005

In this report, I give an overview of higher-order program transformations and try to identify some interesting theoretical problems related to that topic.

Several kinds of transformation can be applied on programs. We distinguish mainly three categories: refactoring, optimization, compilation. The basic requirement common to all transformation is the preservation of the semantics of the program transformed. The performance (time and space complexity) needs not to be preserved.

1 Which transformation?

We give a short overview of the three kinds of transformation. I do not give a full survey of the topic here but just an introduction with some references to interesting related works.

1.1 Refactoring

The term "refactoring" refers to transformations that can be applied on programs at the source level. The aim of refactoring is not to derive new algorithm, but rather to give some automated ways of organizing and maintaining the source code of a program in order to make it more understandable.

Refactoring is a source to source transformation. The simplest refactoring transformation is probably identifier renaming (also known as α-conversion in the functional setting) but there are more complicated ones. Most refactoring tools come with a base of common refactoring functions, such as:

Method extraction consists in extracting all the code involved in the computation of the final value of a variable. The code is moved to a new method, and the variable in the original code is replaced by a call to that method.

Inlining Replacing all occurences of a calls to function f by the body of f.

Pull Up involves moving a method from class B to class A where B derives from A.

Pull Down involves moving a method from class A to class B where B derives from A.

There are several implementation of refactoring tools in the industry, mainly for first order programming languages like C++ and Java. Microsoft has implemented the standard refactoring functions in its development environment Visual Studio. There are also plugins available for Eclipse ([11]).

See [3] for a complete reference on refactoring.

For higher-order programming languages, there is a tool called HaRe ([8]). It's a refactoring tool for the Haskell language. It supports a dozen of refactoring including: α-conversion, memoisation (improve a function by storing intermediate computation and changing the abstract data type accordingly), removing duplicated code, generalise or specialise a definition (by partial evaluation), layered data types (splitting or merging ADT). See [7] for a catalogue of refactoring methods for functionnal languages.

1.2 Optimization

Optimization is achieved by applying transformation at the source level. The aim is to improve the performance of the executed code while conserving its semantics.

Oege de Moor and Ganesh Sittampalam developed a tool called MAG ([1]) for optimizing Haskell functionnal programs by transformation. Their approach relies on the observation that when writing a program, the programmer has to make a trade-off between abstraction and efficency. Abstraction usually improves readability while degrading performance. The idea is to let the programmer express the abstract version of the program as well as the transformation which will leed to an efficient version of the program. The program and its transformation together constitute a meta-program. MAG is based on this principle. Programs are expressed in Haskell and the transformations are expressed using rewriting rules.

To illustrate how MAG works, we give here an example from [1]: consider the function reversing a list. A naive definition would be:

```
let rec reverse l = match l with
[] -> []
| a::x = (reverse x) @ a
```

This is inefficient because the time complexity of the concatenation operator `@` is linear in the size of the list. Therefore the function reverse has a quadractic time complexity.

It is well known that this algorithm is equivalent to the following improved version:

```
let rec fastreverse =
 let fastreverse l ac = match l with
 [] -> ac
 | a::x = fastreverse x (a:ac)
in fastreverse l []
```

This version can be automatically computed by MAG from the first version and the following transformation rules:

```
cat0: [] @ xs = xs;
cat1: (x::xs) @ ys = x::(xs@ys);
catassoc: (xs @ ys) @ zs
  = xs @ (ys @ zs);
promotion: f (foldr plusl e xs)
  = foldr crossl e' xs,
   if {f e = e';
    fun x y -> f (plusl x y)
    = fun x y -> crossl x (f y)
   }
```

These rules express properties about the concatenation operator and the promotion rule (which permits to swap f and foldr when f is an homomorphism).

The key element of the MAG system is the matching algorithm that it used to determine which rewriting rule can be applied to the program, see [10] for more details about it.

1.3 Compilation

Compiling consists in transforming one program expressed in a language source into a program in target language. Usually the transformation is from a high-level to a low-level source language. Some optimization may take part during the compilation (loop fusion,...).

2 Challenges

2.1 Correctness of transformation

Currently, most refactoring tools suffer from a major bug: the transformations do not preserve the semantics of the programs transformed. The reason is that these transformations are not formally proved to be correct.

We would like to have automatic procedures to determine whether a given transformation preserves the meaning of programs.

In [12], Verbaere et al. proposed a scripting functionnal language which allow one to simply express source to source transformations. The functionnal nature of Jungl makes it easy to manipulate structures like abstract syntax trees and graphs used to represent programs. The patterns of the transformation are expressed with path queries (regular expression) over the AST. The following example from [12] illustrates this: it is a pattern describing a path from a variable occurrence var to its declaration as a method parameter: `![var] parent+ [?m:Kind("MethodDecl")] child [?dec:Kind("ParamDecl")] & ?dec.name == var.name`

Jungl allows one to invent and implement very complex refactoring transformations in a concise way.

No research has been done yet on the soundness of the transformation expressed in Jungl. Game semantics could be usefull here. It would be interesting to analyze Jungl programs in terms of transformation on strategies. If we can prove that a transformation conserves the strategies modulo the intrinsic preorder then it is sound. In other words, a transformation is sound if its strategy-transformer equivalent is the identity function of the set of startegies modulo the intrinsic preorder. The toy language Foil introduced in [4] would be an interesting example to start with. In Foil, the full abstraction given by game semantics boils down to regular languages semantics and the observational equivalence becomes decidable.

2.1.1 Compilation

In order to prove soundness of a compiler, it would be necessary to give a game semantics of the source and target language and then prove that the compiler transformation preserve that semantics.

Game semantics may be tricky to apply here since its use is not adapted to first order low level languages (like assembly language).

2.2 Higher-order pattern matching

Pattern matching is usually the key element of a program transformation tools. In order to apply transformation, the tool need to recognize a pattern in the program that correspond to a possible transformation.

Higher-order pattern matching is a long standing problem in computer science. It has first been stated by Huet in 1979.

In [6], it has been proven that the solvability of higher order matching equations, up to β equivalence is undecidable. However, it is still not known whether the problem is decidable in the general case (relatively to $\beta\eta$-equivalence).

We know that the problem is decidable up to level 4 ([5, 2, 9]), the level of the pattern matching problem being the highest order of the type of the variable to be solved.

What is the correspondence of the higher-order pattern matching in game semantics? Based on that, is it possible to find a fast algorithm for the order 4?

3 Can we do transformation using game semantics?

Suppose that we have a mechanical procedure to obtain from a strategy a program whose game semantics is that particular strategy. (Is there such an algorithm for a subset of Idealized Algol?). Let's denote this procedure ϕ (ϕ can map a given game semantics to several possible programs having this semantics).

Consider the term $M = \lambda x.\texttt{if x then 7 else Q}$. Then M $\texttt{true} \Downarrow \texttt{P}$ therefore $[\![M\texttt{true}]\!] = [\![7]\!]$.

What do we get from $\phi([\![7]\!])$? It would be nice to investigate whether there is a function ϕ which would give as a result the "simplest" program (where the notion of simplicity remains to be defined) for the given semantics. In our example we would like to have $\phi([\![M\texttt{true}]\!]) = P_7 = \phi([\![7]\!])$, where P_7 denotes the program with a single operation returning 7.

References

[1] Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 116–149, 1999.

[2] Gilles Dowek. Third order matching is decidable. In *Logic in Computer Science*, pages 2–10, 1992.

[3] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[4] Dan R. Ghica and Guy McCusker. The regular-language semantics of second-order idealized algol. *Theor. Comput. Sci.*, 309(1):469–502, 2003.

[5] Grard Huet. *Resolution d'quations dans les langages d'ordre 1, 2 . . . Ω.* PhD thesis, Universit Paris VI, 1976.

[6] Ralph Loader. Higher order beta matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.

[7] University of Kent. Catalogue of haskel refactoring transformation. http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/.

[8] University of Kent. Refactoring functional programs. http://www.cs.kent.ac.uk/projects/refactor-fp/.

[9] Vincent Padovani. *Filtrage d'ordre suprieure.* PhD thesis, Universit de Paris 7, 1996.

[10] Ganesh Sittampalam. *Higher-order matching for program transformation.* PhD thesis, University of Oxford, 2001.

[11] Mathieu Verbaere. Program slicing for refactoring. Master thesis, 2003.

[12] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. 2005.