Electronic Notes in Theoretical Computer Science 7 (1997)
URL: `http://www.elsevier.nl/locate/entcs/volume7.html` ?? pages

Efficient Compilation of Functional Languages by Program Transformation

André Santos [1]

Departamento de Informática
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

This article presents how automatic program transformation techniques can be used in a functional language compiler to get significant improvement in the performance of the code generated. The transformations used are simple, but when they are repeatedly applied and interact, they achieve results that often are obtained only through specific and more complex transformations.

1 Introduction

The compilation of languages by program transformation has been known and studied for many years [5,4,2]. But the study of program transformation techniques specifically for code improvement has often concentrated on computationally expensive transformations, and quite often the resulting effect of these transformations has been studied on small benchmarks, specially suited to reflect the performance improvements presented by the transformation on ideal circumstances, where it will achieve its best results.

In this article we present how program transformation techniques have been used to improve the performance of the code generated by the *Glasgow Haskell Compiler* (GHC) [8], an optimising compiler for the functional language Haskell [3] developed at the University of Glasgow.

Due to the need to be used in a production quality compiler, the transformations couldn't be computationally expensive, but on the other hand they needed to have significant effects in a large number of generic programs to be considered useful and be integrated into the compiler. The benchmarks to be

[1] Supported in part by CNPq, Grant 300955/95-2. Email: `alms@di.ufpe.br`. WWW: `http://www.di.ufpe.br/~alms`.

used, therefore, couldn't be based on small programs that exercised some particular feature of the language, but on larger programs that weren't originally written to be used as benchmarks.

Following these guidelines, a large number of relatively simple and computationally inexpensive transformations were studied. These transformations, when combined and repeatedly applied result in code that, in other compilers, is often achieved by means of more specific and computationally expensive transformations. In this article we describe the main transformations from the set used by the compiler, by presenting examples of their use to improve the quality of some code fragments that occur in real programs compiled by GHC.

We initially present the notation used in this article, the *Core* language, which is the intermediate representation used by the compiler when applying the transformations. We then describe three sets of transformations, many of them very simple, that achieved significant effects on the performance of a large set of benchmarks.

2 The *Core* Language

The intermediate language used by the compiler, the *Core* language, is itself a simple functional language, but with all the "syntactic sugar" (like pattern matching, list comprehensions and overloading) already transformed out to the simpler constructs of the *Core* language, by well-known techniques [7].

The language is presented in Figure 1. We can see that it has lambda abstractions, type constructors, basic (*unboxed*) data types, function and constructor application, `case`s and `let`s. The language also has type abstraction and application (based on the second order lambda calculus), which is used to preserve type information throughout the transformation process and is also used to guide some transformations and the code generator. For simplicity we will omit the use of this characteristic of the language in this article.

`let`s and `case`s play an important role in enabling the use of program transformation techniques, since they have an important operational reading:

- all evaluation takes place through `case`s, therefore `case`s represent evaluation;
- all dynamic memory allocation (i.e. heap allocation) is done by `let`s. Since we are dealing with a *lazy* language, the evaluation of the right hand side of a `let` only occurs if (and when) its value is demanded by the program (e.g. by a `case`). This evaluation takes place only once, since the first time a `let` is evaluated it is *updated* with its result [2].

Note that the fact that these language constructs have this operational reading results in the simplicity of the function (and constructor) application

[2] actually the heap location that represents the `let` is updated.

Program	$Prog$	$\rightarrow$ $Binding_1$; ... ; $Binding_n$ $n \geq 1$	
Bindings	$Binding$	$\rightarrow$ $Bind$	
		$\mid$ `rec` $Bind_1 \ldots Bind_n$	
	$Bind$	$\rightarrow$ var `=` $Expr$	
Expressions	$Expr$	$\rightarrow$ $Expr\ Atom$	Application
		$\mid$ $Expr\ ty$	Type Application
		$\mid$ $\lambda\ var_1 \ldots var_n$ `->` $Expr$	Lambda Abstraction
		$\mid$ $\Lambda\ ty$ `->` $Expr$	Type Abstraction
		$\mid$ `case` $Expr$ `of` $Alts$	Case Expression
		$\mid$ `let` $Binding$ `in` $Expr$	Local Definition
		$\mid$ $con\ Atom_1 \ldots Atom_n$	Constructor $n \geq 0$
		$\mid$ $prim\ Atom_1 \ldots Atom_n$	Primitive Op $n \geq 0$
		$\mid$ $Atom$	
Atoms	$Atom$	$\rightarrow$ var	Variable
		$\mid$ $Literal$	*unboxed* Object
Literals	$Literal$	$\rightarrow$ $integer \mid float \mid \ldots$	
Alts	$Alts$	$\rightarrow$ $Calt_1 ; \ldots ; Calt_n ; [Default]$	$n \geq 0$
		$\mid$ $Lalt_1 ; \ldots ; Lalt_n ; [Default]$	$n \geq 0$
Constructor Alts	$Calt$	$\rightarrow$ $con\ var_1 \ldots var_n$ `->` $Expr$	$n \geq 0$
Literal Alts.	$Lalt$	$\rightarrow$ $Literal$ `->` $Expr$	
Alt. default	$Default$	$\rightarrow$ var `->` $Expr$	

Fig. 1. *Core* Syntax

syntax, since all their arguments can be restricted to be *atoms*. This also has the effect that the number of simplification rules we will need to express when describing transformations is also reduced, since the places where an expression can occur is reduced (as opposed to applications having expressions as arguments).

The use of this notation, which makes evaluation and allocation explicit is essential to the success of the program transformation technique we will present, since:

- it is concrete enough to allow the presentation of exactly how we are reducing or affecting the cost of evaluation or allocation of an expression, but
- it is also abstract enough so that the transformations and results we present are readily applicable to any lazy functional language compiler, and not tied to a specific implementation or abstract machine used to implement the language.

3 Transformations

In [11] a complete list of the transformations used by the *Glasgow Haskell Compiler* is presented. In this article we will concentrate on only a few of them, and we will introduce them through examples of their effect. Each transformation leads to (or exposes opportunities to) other transformations, and therefore we can introduce many of them in a single example, and we will see how the same simple transformations occur in many of them.

3.1 Avoiding repeated evaluation

The expression `x+x` (where `x` has type `Int`) in a program, results in the following intermediate code in *Core*:

```
(\a -> \b -> case a of
               MkInt a# -> case b of
                             MkInt b# -> case a# +# b# of
                                           r# -> MkInt r#) x x
```

This occurs due to the *inline expansion* of the integer operator (`+`), which evaluates its two arguments (represented by the first two *cases*[3]) and only then it can execute the primitive operation (`+#`), which operates on (and returns) an *unboxed*) (machine) integer, without the integer type constructor `MkInt`[4]. Finally, a *boxed* integer is returned, this is done using the integer type constructor `MkInt`. In this particular case the two arguments are the same `x` variable. Notice that by distinguishing the *boxed* and *unboxed* version of integers, we have more control over the operational characteristic of the

[3] note that the arguments `a` and `b` may be expressions.

[4] An unboxed integer is not (and cannot be) an unevaluated expression (closure).

(`+`) operator, knowing exactly when we are dealing with a "machine" integer addition (`+#`) or the more abstract addition operator of the language (which deals with integers that may in fact be unevaluated expressions). This explicit notation for the two kinds of data types (boxed and unboxed) is presented and discussed in [9].

Actually, at this point we are already using a transformation: inline expansion (*inlining*), which repalaces one or more occurrences of an expression bound by a `let` by the `let`'s right hand side:

$$\texttt{let}\ \ x\ \texttt{=}\ e\ \texttt{in}\ \ \ldots x \ldots \Longrightarrow \texttt{let}\ \ x\ \texttt{=}\ e\ \texttt{in}\ \ \ldots e \ldots$$

In the example above we inlined the definition of the (`+`) operator.

The advantages obtained by inlining in general are:

- a `let` definition might be removed, if all occurrences of the variable are eventually inlined. This saves the cost of allocating the `let`;
- the inlined definition becomes available at the place where it is used, possibly exposing transformations like beta-reduction (discussed below);
- more contextual information is exposed to the inlined expression, which was not available at the place of the definition of the `let`.

Obviously the inlining process must be done carefully, to avoid code explosion due to excessive duplication of the `let`-bound expressions. The process of selecting a `let`-binding for inlining is a subject of research on its own, and we will not discuss it in this article. An specific analysis on criteria used for deciding which expressions to inline can be found in [11].

The second transformation we use is precisely beta-reduction [5]:

$$(\lambda x\ \texttt{->}\ \ e)\ y \Longrightarrow e[y/x]$$

In beta-reduction we are actually moving some of the evaluation that would take place during run-time to compile-time, since beta-reduction is the basic mechanism of evaluation in functional languages. We also expose more contextual information to the variable that is passed as argument, since it will not be an argument to the expression anymore, but will occur in the body of the (old) lambda expression.

Core syntax itself guarantees we cannot have problems of work duplication (duplicating an expression that is shared in an expression due to beta-reduction), since arguments to functions are always atoms.

With beta-reduction the expression now becomes:

[5] The notation $e[x/y]$ denotes an expression e in which *all* free occurrences of y are replaced by x.

```
case x of
  MkInt x# -> case x of
                MkInt y# -> case x# +# y# of
                              r# -> MkInt r#
```

We can then move on to another simple transformation, `case` *reduction* (or `case` *evaluation*), which has three basic forms:

$$\begin{array}{l} \texttt{case}\ \ C\ v_1 \ldots v_n\ \texttt{of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n\ \texttt{->}\ \ e \\ \quad \ldots \end{array} \Longrightarrow e[v_i/x_i]_{i=1}^{n}$$

in which the `case` is evaluating an expression which explicitly has the constructor (C) that occurs in one of the `case` alternatives.

$$\begin{array}{l} \texttt{case}\ \ v\ \texttt{of} \\ \ \ldots \\ \ C\ x_1 \ldots x_n\ \texttt{->}\ \ldots \left(\begin{array}{l} \texttt{case}\ \ v\ \texttt{of} \\ \ \ldots \\ \ C\ y_1 \ldots y_n\ \texttt{->}\ \ e \\ \ \ldots \end{array} \right) \\ \ \ldots \end{array} \Longrightarrow \begin{array}{l} \texttt{case}\ \ v\ \texttt{of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n\ \texttt{->}\ \ldots e[x_i/y_i]_{i=1}^{n} \\ \quad \ldots \end{array}$$

in which we know which constructor is bound to the variable, since it has already been scrutinised by an outer `case`, and

$$\begin{array}{l} \texttt{let}\ \ x\ \texttt{=}\ \ C\ x_1 \ldots x_n \\ \texttt{in}\ \ \ldots \left(\begin{array}{l} \texttt{case}\ \ x\ \texttt{of} \\ \quad \ldots \\ \quad C\ y_1 \ldots y_n\ \texttt{->}\ \ e \\ \quad \ldots \end{array} \right) \ldots \end{array} \Longrightarrow \begin{array}{l} \texttt{let}\ \ x\ \texttt{=}\ \ C\ x_1 \ldots x_n \\ \texttt{in}\ \ \ldots e[x_i/y_i]_{i=1}^{n} \ldots \end{array}$$

in which we know which constructor is bound to a variable, since the variable is explicitly bound to a constructor.

In our example this transformation (in its second form) leads us to the code we would like to obtain, in which we evaluate `x` only once:

```
case x of
  MkInt x# -> case x# +# x# of
                 r# -> MkInt r#
```

3.2 Lazy pattern matching

The compilation of pattern-matching in Haskell is done lazily, i.e. the expression on the right hand side of the pattern should only be evaluated if and when it is needed. This may lead to an innefficient code generation:

```
let (x,y) = expr in h x y
```

when transformed into *Core* becomes:

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in h x y
```

This way we allocate three heap objects, that may eventually be evaluated and updated (which is itself an expensive operation to perform).

But suppose the strictness analyser finds out that (`h x y`) is strict in `x` *or* `y`. This implies that it is also strict in `t`, i.e. we can be sure that `t` will be evaluated during the evaluation of (`h x y`). We can then use a transformation we call `let`-to-`case`:

$$\begin{array}{lcl} \texttt{let}\ \ v\ \texttt{=}\ \ e_v & \Longrightarrow & \texttt{case}\ \ e_v\ \texttt{of} \\ \texttt{in}\ \ e & & \quad C_k\ v_{k1} \ldots v_{kl}\ \texttt{->}\ \ \texttt{let}\ \ v\ \texttt{=}\ \ C_k\ v_{k1} \ldots v_{kl}\ \texttt{in}\ \ e \end{array}$$

Through this transformation we change the allocation of an object in the heap, which we know (from strictness analysis) will be eventually evaluated and updated, to an expression that immediately evaluates the right hand side of the original `let`, returning its result explicitly broken into its components. This transformation can only be used if the `let` variable is strict (i.e. we can guarantee that it will be eventually evaluated). A second condition is that, to avoid excessive code duplication, the transformation is only applied to single-constructor types, like tuples, integers etc., but not on lists, for example.

Applying the transformation we then have:

```
case expr of (w,z) -> let t = (w,z)
                      in let x = case t of (a,b) -> a
                         in let y = case t of (a,b) -> b
                            in h x y
```

`case` *reduction* (which we presented in the previous section!) then eliminates the `case` expression on the right hand side of `x` and `y`, leading us to:

```
case expr of (w,z) -> let t = (w,z)
                      in let x = w
                         in let y = z
                            in h x y
```

At this point `t` can be considered *dead code*, meaning it can be removed, and then `x` and `y` can be easily inlined (since they are bound to other variables) and (have their definitions) removed too:

```
case expr of (w,z) -> h w z
```

This is a much more efficient version for the pattern matching we had, since it is a strict version derived from the results of strictness analysis and simple program transformations.

3.3 Simplifying nested `case`*s*

The "`case`-of-`case`" transformation simplifies expressions in which a `case` expression evaluates another `case` expression:

$$
\begin{array}{l}
\texttt{case} \left(\begin{array}{l}\texttt{case}\ \ e_c\ \texttt{of}\\ alt_{c1}\ \texttt{->}\ \ e_{c1}\\ \dots\\ alt_{cm}\ \texttt{->}\ \ e_{cm}\end{array}\right) \texttt{of}\\
alt_1\ \texttt{->}\ \ e_1\\
\dots\\
alt_n\ \texttt{->}\ \ e_n
\end{array}
\quad\Longrightarrow\quad
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{case}\ \ e_c\ \texttt{of}}\\
alt_{c1}\ \texttt{->} & \texttt{case}\ \ e_{c1}\ \texttt{of}\\
 & \quad alt_1\ \texttt{->}\ \ e_1\\
 & \quad \dots\\
 & \quad alt_n\ \texttt{->}\ \ e_n\\
\dots & \\
alt_{cm}\ \texttt{->} & \texttt{case}\ \ e_{cm}\ \texttt{of}\\
 & \quad alt_1\ \texttt{->}\ \ e_1\\
 & \quad \dots\\
 & \quad alt_n\ \texttt{->}\ \ e_n
\end{array}
$$

A particular instance of this transformation is presented in [1] and [4]. There, it is essentially presented to "short-circuit" `if`s, a case which also occurs often in imperative languages. For example:

```
if (b1 && b2) then e1 else e2
```

where `b1` and `b2` are boolean expressions, and `&&` is the conjunction operator. If `b1` evaluates to false, there is no need to evaluate `b2`, since the result will be `e2` anyway. In our case, the definition of `&&` already has this property:

```
(&&) b1 b2 = case b1 of
               True  -> b2
               False -> False
```

We can now transform the `if` expression to a version using `case`s:

```
case (b1 && b2) of
  True -> e1
  False -> e2
```

We can then inline `&&`, obtaining:

```
case (case b1 of
        True -> b2
        False -> False
     ) of
  True  -> e1
  False -> e2
```

Now we are ready to apply the `case`-of-`case` transformation:

```
case b1 of
  True  -> case b2 of
             True  -> e1
             False -> e2
  False -> case False of
             True  -> e1
             False -> e2
```

The second innermost `case` now explicitly evaluates a constructor, and we can then apply the `case` *reduction* transformation, thus obtaining:

```
case b1 of
  True  -> case b2 of
             True  -> e1
             False -> e2
  False -> e2
```

This way we end up with the result we wanted, avoiding the call to the `&&` function and with a rather simple code.

But the example above shows a limitation of the transformation: `e2` occurs twice in the resulting expression, although it will be evaluated at most once (since the two occurrences are in separate alternatives of the `case` expression). This could lead to code explosion, if `e2` is a large expression.

This code duplication risk can be easily eliminated, by introducing a `let` variable to allow the sharing of the expression `e2`. This way we get the improvements of eliminating the common subexpressions, but without the cost of a common subexpression analysis (in this case). The final expression would be:

```
let v = e2
in case b1 of
     True  -> case b2 of
                True  -> e1
                False -> v
     False -> v
```

This `let` can also be annotated and implemented very efficiently, with a simple code pointer, not as a regular `let` (which is allocated in the heap).

This same method, with the aid of abstracting the free variables, can be used for `case`-of-`case` transformations involving any data types, not only for data types with constructors without arguments, like the booleans in the example above. This way the transformation is generalised to many cases in which it wouldn't be used before, either due to the risk of code duplication or due to the type of the expressions being evaluated.

3.4 *Moving* `let`*s locally*

A set of transformations for which the results were quite surprising were the transformations that move the position where a `let` occurs.

When we change the position of a `let` binding, we are essentially changing *when* it will be allocated, but not *when* it will be evaluated, since we are dealing with a lazy language.

Therefore we have some freedom of anticipating or delaying the allocation, restricted only by the fact that we have to keep the variables introduced by the `let` (and the ones used in its right hand side) in scope.

An interesting example occurs when we are defining a list constant:

```
let x = [1,2,3]
in ...
```

A possible translation into *Core* would be:

```
let x = let v1 = let v2 = 3:[]
                 in 2:v2
        in 1:v1
in ...
```

In this case we are allocating only one object (namely `x`) in the heap, before evaluating the `let` body. Only if (and when) the evaluation of `x` takes place (demanded by the `let` body) we will evaluate `x`. This will in turn cause the allocation of `v1` and the return of the expression `(1:v1)` as the result. `x` is then updated with this new value (`1:v1`). A similar process will happen if (and when) the value of `v1` is eventually needed.

The allocation is therefore done very lazily, and this leads to a single allocation in the case where `x` is never evaluated. On the other hand, if the entire list is eventually evaluated, we will have quite a few allocations and updates, since the `let`s' right hand sides are not in normal form (i.e. are not lambda expressions, constants or constructors, which don't need updates).

But there is an alternative translation, which could also be used:

```
let v2 = 3:[]
in let v1 = 2:v2
   in let x =  1:v1
      in ...
```

In this case we immediately allocate *three* objects in the heap, but they are already in normal form (constructors), therefore if they are ever evaluated we will not incur into the cost of performing updates. More than that, as we saw in Section 3.1, `let`s directly bound to constructors sometimes give us opportunities to perform `case` *reductions*, which we couldn't do if we used the first translation (only `v2` was explicitly bound to a constructor then).

The transformation of the first form into the second can be done by a transformation we call `let` *floating* outwards, since the `let`s are being moved from the right hand sides of other `lets`[6]:

$$\begin{array}{lcl} \texttt{let}\ \ x = \texttt{let}\ \ (\texttt{rec}\ )\ bind & & \texttt{let}\ \ (\texttt{rec}\ )\ bind \\ \qquad\qquad \texttt{in}\ \ e_x & \Longrightarrow & \texttt{in}\ \ \texttt{let}\ \ x = e_x \\ \texttt{in}\ \ b & & \qquad \texttt{in}\ \ b \end{array}$$

One might ask, then, why isn't the code already generated in the second form, instead of using a transformation to get to it? The answer is that there are cases in which we want to do exactly the opposite transformation, moving `lets` *into* other `lets`' right hand side!

A reason for that can be shown if we go back to our example of compilation of lazy pattern matching, in which we used the possibility of finding out that at least one of the tuple components was strict (i.e. guaranteed to be eventually evaluated):

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in h x y
```

[6] there is a similar transformation for recursive `lets`.

Now suppose function `h` is:

```
h x y = let v = x + y
        in (v*3,v+3)
```

If we inline function `h` we get:

```
let t = expr
in let x = case t of (a,b) -> a
   in let y = case t of (a,b) -> b
      in let v = x + y
         in (v*3,v+3)
```

Up to this point we can see the expression isn't strict in either `x` or `y`, since the result is a tuple (and therefore `v` itself isn't strict).

But what if we *float* `x` and `y`'s `let` bindings *inwards* to `v`'s right hand side[7]? We then get:

```
let v = let t = expr
        in let x = case t of (a,b) -> a
           in let y = case t of (a,b) -> b
              in x + y
in (v*3,v+3)
```

This way we moved the `let` bindings *as close as possible* to their use, and therefore we now have *more context information*! We can now see that the `let`s, in their new position, are used *strictly* in their body `(x + y)`. Therefore we now can apply the transformation we described in the lazy pattern matching example (Section 3.2), to get:

```
let v = case expr of (w,z) -> w + z
in (v*3,v+3)
```

By an extensive analysis of the cases where floating *inwards* or *outwards* has a better result, a set of heuristics to decide when each of them should be used was obtained, but the examples above present the main issues involved in the decision.

In [11,10] the details of these and other transformations involving `let`s is presented, including detailed analysis of their effects, advantages and disadvantages. It is also shown (informally) that the heuristics used to chose the direction in which the transformation is applied doesn't risk non-termination, which could occur if an expression had the same `let`s being repeatedly pushed inwards and outwards.

[7] this is only possible because `x` and `y` are only used in `v`'s right hand side.

4 Conclusion

As we saw through the examples presented in this article, the combined effect of simple transformations are much bigger than one would expect by looking at each transformations on their own.

At first sight the transformations are very simple and some of the code that they apply to would never be written by a programmer. During the process of compilation, though, they occur quite often and in unanticipated circumstances.

The set of transformations we presented uses simple, local transformations, and most of them don't depend on expensive analysis to be used.

In [11] the results of applying the set of transformations on 46 programs of the *nofib Benchmark Suite* [6] is presented. The suite consists of programs written by several different programmers, many of them with thousands of lines. They implement a very diverse set of applications and the majority of them were not written to be used as benchmarks, but to solve real problems.

The experiments have shown that instances of the transformations we presented occur very often, and that the combined effect of their use reduces, on average, 19% of execution time and 18% of the heap consumption, with peaks of 51% and 79%, respectively.

References

[1] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, November 1987.

[2] M. Fradet and Daniel Le Metayer. Compilation of functional languages by program transformation. *ACM Trans. on Programming Languages and Systems*, 13(1), January 1991.

[3] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[4] R. A. Kelsey. *Compilation by Program Transformation.* PhD thesis, Yale University, Department of Computer Science, May 1989. YALEU/DCS/RR-702.

[5] D. A. Kranz. *ORBIT - an optimising compiler for Scheme.* PhD thesis, Yale University, Department of Computer Science, May 1988.

[6] William Partain. The nofib benchmarking suite. In John Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow*, Scotland, 1992. Springer Verlag, Workshops in Computing.

[7] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[8] Simon Peyton Jones, C. Hall, Kevin Hammond, William Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, March 1993.

[9] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture*, pages 636–666, September 1991.

[10] Simon Peyton Jones, William Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *ACM International Conference on Functional Programming*. ACM Press, 1996.

[11] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, July 1995.