On Program Transformation in the Glasgow Haskell Compiler

André Santos* Simon L Peyton Jones
University of Glasgow†

Abstract

In this paper we describe a series of program transformations that are currently being implemented in the Glasgow Haskell Compiler. They are semantics preserving program transformations and therefore are perfectly suitable for incorporation in a compiler. We describe some of the transformations, how they interact and their impact on the time/space behaviour of some programs.

1 Introduction

Program transformations can considerably improve the performance of certain programs, by transforming inefficient code into more efficient versions. In other cases it can change the space behaviour of a program, making it use less memory.

The Glasgow Haskell Compiler was designed with program transformations in mind, allowing for transformation passes to be easily incorporated into the compiler. Here we will show some of the transformations that have been actually implemented and how they affect the time/space behaviour of some programs.

We initially describe the basic idea behind some transformations and how they are carried out through an example. Then we discuss some of them in more detail and present future directions for research and some preliminary results.

The Glasgow Haskell Compiler works by translating a Haskell program into the Core Language (Figure 1), which is in fact the second order lambda calculus augmented with some constructs, like `case` and `let`. The Core Language allows us to express boxed as well as unboxed values, therefore allowing many transformations usually relegated to the code generator to be expressed as Core-to-Core transformations. It is in the Core Language that most of the program transformations we will describe are carried out.

Next, the program is translated from the Core Language to Spineless Tagless G-Machine (STG) Language [4], the abstract machine used by the compiler. Only at this level the transformations start to be more specific to the implementation model we use. Finally the code is translated to C, which is used as a high level assembler.

*Sponsored by CAPES, Brazil.

†Authors' address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, U.K. E-mail: {andre, simonpj}@dcs.glasgow.ac.uk

Program	*Prog*	→	$Binding_1$; ... ; $Binding_n$	$n \geq 1$
Bindings	*Binding*	→	`nonrec` *Bind*	
		|	`rec` *Binds*	
	Binds	→	$Bind_1$; ... ; $Bind_n$	$n \geq 1$
	Bind	→	*var* `=` *Expr*	
Expression	*Expr*	→	$Expr_1$ $Expr_2$	Application
		|	*Expr ty*	Type application
		|	λ *vars* `->` *Expr*	Lambda abstraction
		|	Λ *ty* `->` *Expr*	Type abstraction
		|	`case` *Expr* `of` *Alts*	Case expression
		|	`let` *Binding* `in` *Expr*	Local definition
		|	*con* $var_1 \ldots var_n$	Constructor $n \geq 0$
		|	*prim* $var_1 \ldots var_n$	Primitive $n \geq 0$
		|	*var*	Variable
		|	*Literal*	Unboxed Object
Literal values	*Literal*	→	*integer* | *float* | ...	
Alternatives	*Alts*	→	$Calt_1$; ... ; $Calt_n$; *Default*	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	→	*Con* $var_1 \ldots var_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	→	*Literal* `->` *Expr*	
Default alt	*Default*	→	`NoDefault`	
		|	`_ ->` *Expr*	
		|	*var* `->` *Expr*	

Figure 1: Syntax of the Core language

Pass	Action
Function Specialisation	Specialises overloaded functions used with same types/dictionaries
Simplifier I	Does some unfolding, β-reductions and some transformations
Simplifier II	Same as above with a more aggressive unfolding strategy
Simplifier III	Same as above also unfolding non-recursive functions with higher order arguments
Let-Floating	Floats let-bindings to the outermost possible position
Arg. Satisfaction Check	Reduces argument satisfaction checks in STG

Table 1: Compiler Options

2 Transformations

A lot of work has been done on techniques for transforming functional programs into more efficient ones. These techniques have sometimes resulted in the development of tools to aid the transformation process. Most of these tools, however, are dependent on user assistance and/or on very expensive analysis techniques in order to achieve good results. Therefore these techniques are not suitable for being incorporated into real functional language compilers.

We deal here with automatic semantics preserving transformations, which can be derived and carried out directly from the original program. In fact the selection of the transformations was mostly guided by inspection of the actual code generated by the compiler after each step, in small as well as large programs.

Most of the transformations we will describe are carried out in the Core Language, the language to which the original program is translated, by removing syntactic sugar and transforming out pattern matching, list comprehensions, etc. It allows us to express and exploit some ideas on unboxed values (as described in [5]), and this will be extensively demonstrated through the examples.

Some other transformations are expressed in the STG Language (STG-to-STG transformations), to which the Core language is translated immediately before code generation. The reason for having such transformations expressed as STG-to-STG transformations rather than Core-to-Core in the Glasgow Haskell Compiler is that they either exploit specific characteristics of STG or are more suitable to be done after the translations from Core to STG as this translation would expose more opportunities for transformations.

In Figure 2 we have an outline of the transformations. In Table 1 we give some more details on what each transformation pass does.

Some of the transformations described in the next sections are quite simple and widely known, but our main concern is with how they interact and how the compiled code can benefit from the different analysis/transformations. Also, although many of the expressions transformed out are very unlikely to appear in the original program, they are often found after the program is desugared to the Core Language, as we will see in the examples. One may argue that we

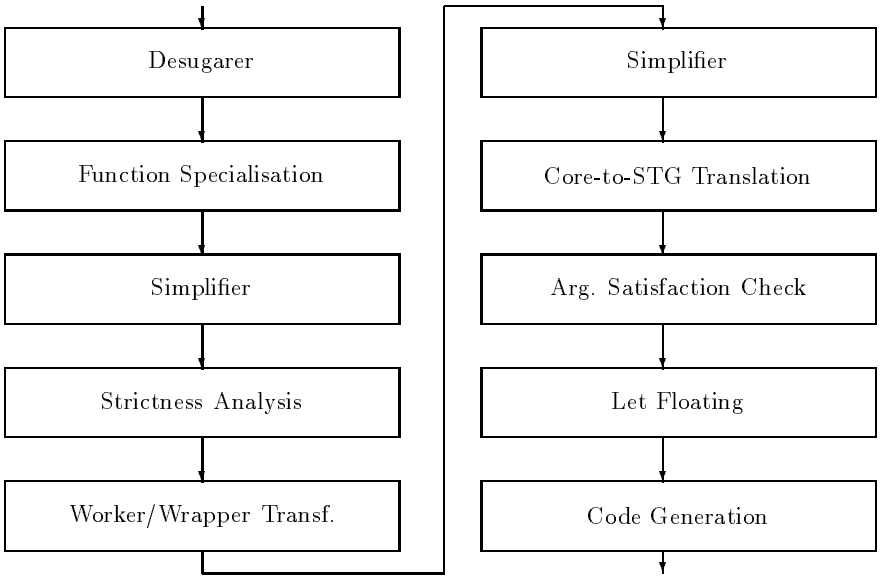

Figure 2: Transformation Passes

should avoid generating such expressions during the desugaring, but this would unnecessarily complicate this process.

We will introduce some transformations by giving an example and showing how they are used in it. The example we will use is quite simple, but the transformations that show up in it also occur very often in large programs.

3 An Example

In this section we will show how the transformations interact when generating a more efficient version of the factorial program. This will give an idea of the effect of the entire transformation sequence before we discuss some of the transformations in more detail.

A definition of the factorial function in the Core language could be

```
fact :: Int -> Int
fact = \ n -> case (n < (MkInt 1#)) of
                True -> MkInt 1#
                False -> n * fact (n - (MkInt 1#))
```

`MkInt` is used as the constructor for boxed integers. We put a `#` after a variable or value that is unboxed, as in [5].

We initially unfold the definitions of `-`, `*` and `<` to make explicit the unboxing/boxing operations on their arguments/results. The definition of `-` for example is

```
(-) = \ x y -> case x of
```

```
                MkInt x# -> case y of
                              MkInt y# -> case (x# -# y#) of
                                            r -> MkInt r
```

where `-#` is the subtraction operation on unboxed integers. These unfoldings will lead us to many cases where we will be unboxing a value that has previously been unboxed or that has just been boxed, which are redundant operations.

The first time the simplifier is applied it transforms the code by

- unfolding basic operations
- applying β-reductions where appropriate
- avoiding redundant boxing/unboxing of values
- doing case-of-case transformations where appropriate

These and other transformations are detailed in the next section. By doing this the code is transformed to

```
fact = \ n -> case n of
                MkInt n# ->
                    case (n# <# 1#) of
                      0# -> case (fact (case (n# -# 1#) of
                                          v# -> MkInt v#)) of
                                MkInt v'# -> case (n# *# v'#) of
                                               v''# -> MkInt v''#
                       _ ->  MkInt 1#
```

This already avoids many unnecessary boxing/unboxing operations, which is an improvement by itself. But it can do an even better job if we use a strictness analyser together with the Worker/Wrapper transformation, which will split the function in a worker/wrapper pair of functions. The idea of the Worker/Wrapper Transformation is to make minimal changes from the original functions while splitting, and let the simplifier do the rest of the job. Therefore we get an inefficient Worker/Wrapper pair which will become a more efficient one through the transformations:

```
fact :: Int -> Int
fact = \n -> case n of
               MkInt n# -> case fact' n# of
                             r# -> MkInt r#
fact' :: Int# -> Int#
fact' = \ n# -> let n = MkInt n#
                in case
                     (case (n# <# 1#) of
                        0# -> case (fact (case (n# -# 1#) of
                                            v# -> MkInt v#)) of
                                  MkInt v'# -> case (n# *# v'#) of
                                                 v''# -> MkInt v''#
                        _ -> MkInt 1#)
                   of
                     MkInt v'''# -> v'''#
```

Now the simplifier is called again to unfold the wrapper (`fact`) into the worker (`fact'`), to get the worker to call itself. By doing this we get many other opportunities for removing extra boxing/unboxing operations, case-of-case transformations, β-reductions, etc.

```
fact' = \ n# -> case (n# <# 1#) of
                  0# -> case (n# -# 1#) of
                          v# -> case fact' v# of
                                  v'# -> (n# *# v'#)
                  _ -> 1#
```

Clearly this definition is quite an improvement on the first one, by keeping the values unboxed during most of the computation.

4 Core-to-Core Transformations

Here we describe some of the transformations that are implemented as Core-to-Core transformations. The list is not exhaustive, but is intended to give a reasonable idea of the transformations carried out.

4.1 Function Specialisation Pass

The Function Specialisation Pass specialises local overloaded functions that are always used with the same types/dictionaries. An example of how it works follows.

```
f y = let double x = x + x in double (double y)
```

Assuming `plus` is the overloaded addition operation, which receives a type and dictionary as arguments, this would become (in Core)

```
f = /\ ty -> \ dict -> \ y ->
     let double = /\ ty' -> \ dict' -> \ x -> plus ty' dict' x x
     in double ty dict (double ty dict y)
```

As `double` is locally defined and called with the same type/dictionary we may transform this definition to

```
f = /\ ty -> \ dict -> \ y ->
     let double = /\ ty' -> \ dict' -> \ x -> plus ty' dict' x x
     in let double' = double ty dict
        in double' (double' y)
```

then as there is only one occurrence of `double` we inline it, obtaining

```
f = /\ ty -> \ dict -> \ y ->
     let double' = \ x -> plus ty dict x x
     in double' (double' y)
```

Some of the work of this pass could be eliminated if the typechecker already avoided these occurrences of types/dictionaries, but it is still not clear if all instances could be easily eliminated by the typechecker without further complicating it. Anyway, other transformations can introduce such expressions, which are worth transforming out.

Notice that this process is quite similar to common subexpression elimination. In general, common subexpression elimination in functional languages can generate unanticipated space leaks. In this case we do a very restricted version of common subexpression elimination, for dictionaries, so there is no such risk.

4.2 The Simplifier

The Simplifier consists itself of a series of very simple transformations. Many of these transformations deal with expressions that are very unlikely to be generated directly by a programmer, but are very often generated after desugaring Haskell to Core or by other transformation passes. It proceeds as follows:

- to guide the transformations, information is needed about the way in which each named value is *used*. This information is collected by a Substitution Analysis pass, which annotates every binder with information regarding whether it can be unfolded, removed or kept as a let-binding. The Substitution Analyser in its basic version counts occurrences of binders. If the binder does not occur in the program it is marked as not used and will be removed by the simplifier. If it occurs only once or is bound to a variable or a basic literal it is marked to be substituted (unfolded). If it occurs more than once it is marked for not being substituted, as this could possibly generate code and work duplication. Later we will discuss some more aggressive options that may unfold binders referenced more than once. In these cases we may duplicate code, but never duplicate work.

- Based on this information the program is simplified. This includes unfolding definitions, β-reductions, etc. We will describe some of them in more detail in the following sections.

- The process iterates until no more simplifications are possible. In typical programs quite a few iterations are necessary. Although we currently do not have a proof of the termination of the process, we are careful to ensure that transformations we are introducing do not interfere with the termination of the process.

Here we list some of the transformations implemented by the simplifier.

4.2.1 λ-elimination

An application of a lambda abstraction is always transformed to a `let` expression, thus:

(\$x$ `->` $body$) e $\quad\Longrightarrow\quad$ `let` x `=` e `in` $body$

4.2.2 Let-elimination

A `let`-binding may be removed, unfolded or kept as a let depending on the corresponding annotation that comes from the substitution analysis pass. The three possibilities are

- binding not used: `let` x `=` e `in` $body$ $\Longrightarrow$ $body$
- unfold the binding[1]: `let` x `=` e `in` $body$ $\Longrightarrow$ $body[e/x]$
- do not unfold: No action takes place. This is the most critical decision, as we want to take as many opportunities for simplifications as possible, and unfolding let bindings is a major source of it. This is why we allow some more aggressive options for the simplifier, which will select more bindings to be unfolded.

4.2.3 Case-elimination

If a `case` expression scrutinises a (saturated) constructor application, it can readily be eliminated:

`case` $(C\ e_1 \ldots e_n)$ `of` $\ldots\ C\ x_1 \ldots x_n$ `->` e
$\Longrightarrow$ `let` x_1 `=` e_1`;` $\ldots$`;` x_n `=` e_n `in` e

But a similar situation can arise in two other ways. Firstly, the `case` expression might be scrutinising a variable (or in general an expression[2]) which has already been scrutinised:

`case` e `of {`$\ldots C\ x_1 \ldots x_n$ `->` $\ldots$`case` e `of {`$\ldots C\ y_1 \ldots y_n$ `->` $body$`}}`
$\Longrightarrow$
`case` e `of {`$\ldots C\ x_1 \ldots x_n$ `->` $\ldots$`let` y_1 `=` x_1`;` $\ldots$`;` y_n `=` x_n `in` $body$`}`

Secondly, it might be scrutinising a variable which is `let`-bound to a (saturated) constructor application:

`let` x `=` $C\ x_1 \ldots x_n$ `in` $\ldots$`case` x `of`$\ldots C\ y_1 \ldots y_n$ `->` $body$
$\Longrightarrow$ `let` x `=` $C\ x_1 \ldots x_n$ `in` $\ldots$`let` y_1 `=` $x_1 \ldots y_n$ `=` x_n `in` $body$

This transformation is useful when x occurs many times in its scope, so the `let` expression is not going to be eliminated. Notice that the arguments to the constructor should be variables, so that no loss of laziness occurs; this can easily be guaranteed by introducing new `let` bindings.

4.2.4 Let-floating

Many `let` bindings can be floated outwards somewhat to increase the applicability of other transformations.

`let {`x `=` `let(rec)` $bind$ `in` e_x`}` `in` b
$\Longrightarrow$ `let(rec)` $bind$ `in let` x `=` e_x `in` b

This is useful if e_x is a constructor, or even if it is a partial application (then no update is required). Also STG can allocate the two closures at once after the transformation, and before two allocations would be executed.

The only possible disadvantage would be in cases when x would not be evaluated. In this case we would have allocated one extra closure (for $bind$) that would not be allocated in the first form.

Lets can be floated out of many other expressions, like lambda abstractions, case alternatives, letrecs, etc., but in performing these floatings we have to be careful not to float bindings outside binders used by it, so we need information

[1] Renaming of variables is performed during the substitution process to avoid name clashes.
[2] We currently handle only the case when it is a variable.

regarding free variables of expressions. In fact we have a more general let floating pass that handles all these cases, but it is done in the STG language. As when translating from Core to STG many new lets are generated, which may also be floated out, we would need such a transformation in the STG level anyway. Thus we perform here only the simpler cases, and leave the others to be done as a STG-to-STG transformation.

When floating lets a sort of space leak may occur, when, for example, a definition is floated to the top level. In this case the definition might not be garbage collected, and therefore keep some (possibly large) amount of memory in the heap which would have been garbage collected if the definition had not been floated. This may significantly increase the number of garbage collections or even, due to th increase in memory usage, make impossible to run the problem.

Although this does not seem to be a problem for most programs, a possible solution would be not to float definitions to the top level or whenever they ae floated have the option of not updating them, that is, recomputing them whenever they are used. As can be seen there is a space-time trade off in this aspect of the optimisation, where one can either choose to share the computation and use more memory or (possibly) recompute it but save space.

4.2.5 Case-of-case

$$
\begin{array}{ll}
 & \texttt{case (case } e \texttt{ of } p_{11} \texttt{ -> } e_{11} \texttt{; } \ldots \texttt{; } p_{1n} \texttt{ -> } e_{1n}\texttt{) of} \\
 & p_{21} \texttt{ -> } e_{21} \texttt{; } \ldots \texttt{; } p_{2m} \texttt{ -> } e_{2m} \\
\Longrightarrow & \texttt{case } e \texttt{ of} \\
 & p_{11} \texttt{ -> case } e_{11} \texttt{ of } p_{21} \texttt{ -> } e_{21} \ldots p_{2m} \texttt{ -> } e_{2m} \\
 & p_{1n} \texttt{ -> case } e_{1n} \texttt{ of } p_{21} \texttt{ -> } e_{21} \ldots p_{2m} \texttt{ -> } e_{2m}
\end{array}
$$

This was first described by Augustsson in his thesis [1]. It may duplicate code, but not work.

4.3 More Aggressive Options of the Simplifier

In fact we do not have a more aggressive version of the simplifier, but a more aggressive substitution analysis, that will select more expressions to be unfolded, other than the ones we can guarantee not to increase code size (as are used only once) or duplicate work (when the bindees are variables or basic literals).

The options currently available are:

4.3.1 Selecting Candidates for Unfolding

By unfolding function definitions we do not risk duplicating work, but risk duplicating code. This option heuristically selects small functions with certain characteristics to be unfolded.

Currently the bindings selected by this option are the non-recursive ones which define functions with small and relatively simple bodies. It was observed that this class of functions when unfolded usually exposes simple transformations and β-reductions at compile time and therefore (usually) do not result in significant code duplication. As said earlier this is a completely arbitrary selection and one can find examples where it may lead to a large code increase.

By avoiding recursive functions we avoid termination problems and as the bindings are always functions, basic values or variables no risk of duplicating work is incurred.

4.3.2 Unfolding Functions with Higher Order Arguments

The aim of this is to select functions that contain high order arguments to be (possibly transformed and) unfolded, eliminating part of the higher order nature of the program.

For dealing with higher order recursive definitions we expect to incorporate an analysis and transformations similar to those described in [2] for deciding when higher order arguments can be effectively removed by transforming the definitions and then specialising the functions.

For non-recursive higher order functions we will analyse whether their unfolding will expose further reductions and unfold based on this information.

The current implementation of this simply selects non-recursive higher order functions to be unfolded. This handles the unfolding of most monad definitions which allowed us to verify that by unfolding these definitions in a program that makes heavy use of them may give a significant reduction of execution time as well as space usage. Due to the specific nature of monads, these unfoldings expose a high number of compile time β-reductions which result in not only reduction in execution time but also avoid the large code increase one would expect by doing such unfolding.

Also, reducing the higher order nature of the program we make the work of the strictness analyser easier.

4.4 Transformations based on Strictness Analysis

These currently are Worker/Wrapper transformations, which consist of exposing unboxed values to transformations as described in [5]. We will not further describe it due to space limitations, but it is presented in the reference above.

After these transformations, the simplifier is used to unfold wrapper definitions, which by themselves expose many of the transformations the simplifier handles.

Currently the simplifier is used twice (when together with the strictness analyser and the Worker-Wrapper transformation). Before the strictness analysis it simplifies the code, allowing the strictness analysis to be performed in a simpler program, possibly allowing it to do a better or faster job. After strictness analysis the simplifier will unfold wrappers and explore any further transformations exposed by this.

5 STG-to-STG Transformations

Due to space limitations we will not describe the STG language in detail. But it can be seen as a simple functional language, and the only relevant information to understand why these transformations are in the STG rather than in the Core language are specific to the fact that the Glasgow Haskell Compiler is translated into STG and the expressions that are handled by these transformations can be generated during this translation. Therefore, to avoid duplicating identical

transformations in the Core Level, we decided to apply them solely in the STG language. Also some of them take advantage of specific characteristics of this abstract machine, and may not be useful transformations for other models.

5.1 Let Floating

The Let Floating pass floats out let bindings as far as possible towards the top level. By as far as possible we mean up to the point where one of its free variables is bound.

The algorithm used is similar to the one described in [6] which is used as part of the full laziness transformation.

Notice the similarity of this pass with the full laziness transformation. In fact, to have full laziness the only differences are that we would need to let-bind the maximal free expressions and also allow the insertion of let bindings between lambda-bound variables.

5.2 Reducing Argument Satisfaction Checks

STG can take advantage of transforming definitions like

```
let g = \v1 v2 -> v1 + v2          let g = \v1 v2 -> v1 + v2
in let f = \x -> g (x+1)     ⟹    in let f = \x y -> g (x+1) y
   in f 1 2                           in f 1 2
```

In the first one we have an argument satisfaction check for `f` and another for `g`, and in the second as we know that `g`'s application is saturated (has all the arguments that g will need) we can avoid this argument satisfaction check.

6 Some Results

In this section we present some preliminary results in the use of the transformations we described in the previous sections.

Here we have the results of applying different options in some programs from a benchmark suite [3]. All measurements are in CPU seconds in a Sun 4/40.

	No Simpl.	Simpl.	Simpl. + letfloating
exp 3 8	82.7	83.0	83.2
primes	9.2	9.1	10.1
cichelli	99.5	93.6	82.1
clausify	6.1	4.1	4.2
minimax	3.1	2.6	2.3
rewrite	2.2	2.4	2.0
compress	7.5	5.5	5.4

We have not mentioned in the tables above some of the transformation passes we have described. This is due to their minimal impact on these test programs. But for programs which have the appropriate characteristics the results are quite reasonable. Most of the transformations are always beneficial

(they do not impose extra overhead to the execution of a program), but a few of them may make a few programs run slower, e.g. let floating (see section 4.2.4), although this seems not to be the typical behaviour. This is the sort of problem that causes a worse performance in programs like `exp 3 8` and `primes`.

The space behaviour is also one of our targets, and although we will not present detailed results here, some transformations also significantly reduce the space requirement of certain programs, but a more detailed analysis of their impact on space usage is still to be done.

The effect on the compilation time of using the transformations above is usually minimal, and in some cases the total compilation time is even reduced, as less (or a simpler version) of the code is passed to the subsequent passes.

7 Conclusions

As expected, the initial results demonstrate that the transformations have quite different effects on different programs. As they do not rely on very expensive analysis techniques, their impact on compile time is minor, so it seems like they can be safely incorporated into the compilation process, as they may cause a reasonable gain in time/space performance.

We are currently working on extending this set of transformations and measuring their influence on the time/space behaviour of programs. This way we expect to supply sufficient information on them allowing one to select which set of transformations best suits particular applications, as well as allowing a user to select more aggressive options when he can afford to trade some extra increase in code size for a reduction in execution speed.

References

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

[2] W. N. Chin and J. Darlington. Removing higher-order expressions by program transformation, February 1991.

[3] W. D. Partain. The nofib benchmarking suite. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.

[4] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[5] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, Cambridge, September 1991.

[6] S. L. Peyton Jones and D. Lester. A fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, May 1991.