Compilation by Transformation in the Glasgow Haskell Compiler

Simon Peyton Jones André Santos*
University of Glasgow†

Abstract

In this paper we describe the full set of local program transformations implemented in the Glasgow Haskell Compiler. The transformations are presented as source to source transformations in a simple functional language. The idea is that by composing these simple and small high level transformations one can achieve most of the benefits of more complicated and specialised transformations, many of which are often implemented as code generation optimisations.

1 Motivation

Quite a few compilers use the *compilation by transformation* idiom. The idea is that as much as possible of the compilation process is expressed as correctness-preserving transformations, each of which transforms a program into a semantically-equivalent program that (hopefully) executes more quickly or in less space. Functional languages are particularly amenable to this approach because they have a particularly rich family of possible transformations. Examples of transformation-based compilers include the Orbit compiler ([9]), Kelsey's compilers ([8],[7]), the New Jersey SML compiler ([1]), and the Glasgow Haskell compiler ([15]). Of course many, perhaps most, other compilers also use transformation to some degree.

Compilation by transformation uses automatic transformations; that is, those which can safely be applied automatically by a compiler. There is also a whole approach to programming, which we might call *programming by transformation*, in which the programmer manually transforms an inefficient specification into an efficient program. This development process might be supported by a programming environment which does the book keeping, but the key steps are guided by the programmer. We focus exclusively on automatic transformations in this paper.

Automatic program transformations seem to fall into two broad categories:

- **Glamorous transformations** are global, sophisticated, intellectually satisfying transformations, sometimes guided by some interesting kind of analysis. Examples include: lambda lifting ([6]), full laziness ([5],[17]),

*Sponsored by the Brazilian agency CAPES under contract 2323/91-5.

†Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: `{simonpj,andre}@dcs.gla.ac.uk`. URL: `http://www.dcs.gla.ac.uk`.

closure conversion ([2]), deforestation ([19],[3],[11],[4]), transformations based on strictness analysis ([16]), and so on. It is easy to write papers about these sorts of transformations.

- **Humble transformations** are small, simple, local transformations. Individually they look very trivial. Here are two simple examples:

```
let x = y in E[x]                  ===>  E[y]

case (x:xs) of (y:ys) -> E1[y,ys]  ===>  E1[x,xs]
               []     -> E2
```

  The notation `E[]` stands for an arbitrary expression with zero or more holes. The notation `E[e]` denotes `E[]` with the holes filled in by the expression `e` (All transformations throughout the paper assume renaming is performed to avoid name capture problems. We discuss this in section 5.2). Transformations of this kind are almost embarrassingly simple. How could anyone write a paper about them?

This paper is about *humble transformations*, and how to implement them. Although each individual transformation is simple enough, there is a scaling issue: there are a large number of candidate transformations to consider, and there are a very large number of opportunities to apply them.

In the Glasgow Haskell compiler, all humble transformations are performed by the so-called *simplifier*. Our goal in this paper is to describe the complete set of transformations used by the simplifier and present measurements of their usefulness. It is important to notice that not all of the transformation were immediately obvious to us, but derived from close inspection of the intermediate code of the compiler.

2 The Core language

The simplifier performs transformations on the Core language (Figure 1), which is a desugared version of Haskell, i.e. a simpler language with things like list comprehensions and pattern matching already transformed out to simpler constructs.

The concrete syntax we use is conventional: parentheses are used to disambiguate; application associates to the left and binds more tightly than any other operator; the body of a lambda abstraction extends as far to the right as possible; the usual infix arithmetic operators are permitted; the usual syntax for lists is allowed, with infix constructor "`:`" and empty list `[]`; and, where the layout makes the meaning clear, we allow ourselves to omit semicolons between bindings and `case` alternatives.

Notice that the bindings in `let` expressions are all simple; that is, the left hand side of the binding is always just a variable. Function bindings are expressed by binding a variable to a lambda abstraction (although we permit ourselves the small liberty of sometimes writing the arguments of function bind-

Program	*Prog*	$\rightarrow$	$Binding_1$; ... ; $Binding_n$	$n \geq 1$
Bindings	*Binding*	$\rightarrow$	*Bind*	
		|	`rec` $Bind_1 \ldots Bind_n$	
	Bind	$\rightarrow$	*var* `=` *Expr*	
Expression	*Expr*	$\rightarrow$	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	λ $var_1 \ldots var_n$ `->` *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 $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	$\rightarrow$	*var*	Variable
		|	*Literal*	Unboxed Object
Literal values	*Literal*	$\rightarrow$	*integer* | *float* | ...	
Alternatives	*Alts*	$\rightarrow$	$Calt_1$; ... ; $Calt_n$; *Default*	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	$\rightarrow$	*Con* $var_1 \ldots var_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	$\rightarrow$	*Literal* `->` *Expr*	
Default alt	*Default*	$\rightarrow$	*var* `->` *Expr*	

Figure 1: Syntax of the Core language

ings to the left of the `=` sign). Similarly, the patterns in `case` expressions are all simple; nested pattern matching has been compiled to nested `case` expressions.

Other important points about the Core Language are:

- It is based on the Second order lambda calculus (i.e. with type abstractions and type applications), extended with `let`s and `case`s.
- Arguments are atomic, i.e. variables or literals. In section 5.1 we will discuss the reasons for this decision.
- it supports unboxed data types, and unboxed `case`s ([16]).
- Constructors and primitives are always saturated.
- The Core language has a direct operational reading: allocations are represented by `let`s and evaluation by `case`s.

3 Transformations

This section lists all the transformations implemented by the simplifier. Because it is a complete list, it is a long one. We content ourselves with a brief statement of each transformation, augmented with forward references to Section 4 which gives examples of the ways in which the transformations can compose together.

3.1 Beta reduction

If a lambda abstraction is applied to an argument, we can simply beta-reduce. This applies equally to ordinary lambda abstractions and type abstractions:

```
(\x  -> E[x]) arg      ===>    E[arg]
(/\a -> E[a]) ty       ===>    E[ty]
```

There is no danger of duplicating work because the argument is guaranteed to be a simple variable or literal.

3.2 Floating applications inward

Applications can be floated into a `let` or `case` expression. This is a good idea, because they might find a lambda abstraction inside to beta-reduce with:

```
(let Bind in E) arg   ===>    let Bind in (E arg)

(case E of {P1 -> E1;...; Pn -> En}) arg
===>
case E of {P1 -> E1 arg; ...; Pn -> En arg}
```

Notice that the second version might cause a minor increase in code size, as the argument is duplicated in each branch.

3.3 Floating `let` out of `let`

It is sometimes useful to float a `let` out of a `let` right-hand side:

```
let x = let(rec) Bind in B1             let(rec) Bind in
in B2                            ===>   let x = B1
                                        in B2

letrec x = let(rec) Bind in B1          let(rec) Bind
in B2                            ===>            x = B1
                                        in B2
```

This is particularly good if

- `B1` is in head normal form (`x` becomes a non-updatable closure)
- `x` is strict in `B2`, and therefore we know that `Bind` would be allocated anyway.

3.4 let to case

The `let`-to-`case` transformation can be done whenever we have a strict `let` that is not already in normal form, that is a `let` that is guaranteed to be demanded during the evaluation of its body and is not bound to a constructor or lambda expression:

```
let v = E1 in E2  ===> case E1 of v -> E2
```

The idea here is that in the original expression we are allocating a closure for `v` in the heap which only later will be evaluated (as it is strict) and possibly updated (if `v`'s closure was updatable). After the transformation we evaluate `E1` first and bind it to `v`, therefore saving the cost of the update and some heap allocation if the update was not done originally in place. Notice that the semantics of `case`s in Core mean that `E1` *is* evaluated, whereas in Haskell this would not be the case, and therefore this transformation would be not valid in Haskell.

Actually there is an extra restriction for doing this transformation: the `let` must *not* have a function type (that is, it must not be a function). This restriction is due to implementation details, as `case`s cannot scrutinise objects which have a function type.

3.5 Unboxing let-to-case

The unboxing `let`-to-`case` transformation is similar to the `let` to `case` transformation, but it has the advantage of exposing the structure of the expression, by explicitly exposing its constructor. To avoid code duplication this is only used when the type of the `let` binding is a single constructor data type, like n-tuples, boxed integers etc.

If `v` is of a constructor type and `E2` is strict in `v` then

```
let v = E1 in E2  ===> case E1 of C a b -> let v = C a b in E2
```

The extra advantage here compared to the previous transformation is that as the structure is made explicit transformations like the `case` of known constructor (Section 3.7) can be exposed. Also often the `let` binding introduced by the transformation is later eliminated.

3.6 Floating case out of let

`case`s may be floated out of strict `let`s:

```
let v = case E1 of           case E1 of
          C a b -> E2  ===>    C a b -> let v = E2
in E3                                   in E3
```

where `E3` is strict in `v` and `v` is not a free variable in `E1`.

Thsi transformation increases the scope of the `case`, and therefore it might expose transformations, such as `case` of known constructor, in `E3`. It is also good if `E2` is a head normal form expression, as `v` will no longer be a thunk and therefore no updates will be performed on it.

If the `case` has multiple branches we can still do the transformation but we would have some code duplication, as `E3` would occur in each of the branches.

This can be avoided by using a similar technique to the one we use for the `case` of `case` transformation (Section 3.11), where we create a new `let` binding (which we call a *join point*):

```
let v = case E1 of              let j v = E4
         C1 a b -> E2    ===>   in case E1 of
         C2 a b -> E3               C1 a b -> let v = E2 in j v
in E4                               C2 a b -> let v = E3 in j v
```

This avoids duplicating `E4` in each of the branches. The newly-created `let` can be implemented very efficiently (as we discuss in Section 4.3) and therefore does not introduce any major efficiency or allocation costs. Although we lose the benefit of increasing the scope of the `case` to include `E4`, we will still benefit in the cases in which `E2` or `E3` are head normal form expressions (no updates then).

Notice that even if one is already using the `let`-to-`case` transformation, which would remove many of the opportunities for this transformation, this transformation is still useful in cases when the `let`-to-`case` transformation cannot be used, like when the `let` rhs has a functional type.

Also, there is an interesting optimisation that uses the `case` floating from `let` transformation, but without the restriction on the `let` being strict: for *cheap, non-failing* `case`s. These are `case`s evaluating some primitive operations on unboxed values, such as primitive `Int` addition, subtraction, multiplication, and similar operations for `Float`s and `Double`s. An example of this is

```
let v = case x# +# 1# of         case x# +# 1# of
          r# -> I# r#     ===>     r# -> let v = I# r#
in E                                      in E
```

where names with `#` are unboxed values and `I#` is the constructor for boxed integers. In this case `v` is now bound to a constructor, therefore no updates are performed. Although this is doing some eager evaluation that might be unnecessary, the cost of the operation is small and the expression could end up being evaluated anyway. This transformation is sometimes called "cheap eagerness".

3.7 Case of known constructor

If a `case` expression scrutinises a constructor, the `case` can be eliminated. This transformation is a real win: it eliminates a whole `case` expression.

```
case (C a1 ... an) of
  ...                              ===>  E[a1...an]
  C b1 ... bn -> E[b1...bn]
  ...
```

If none of the constructors in the alternatives match, then the default is taken:

```
case (C a1 ... an) of                    let y = C a1 ... an
  ...[no alt matches C]...         ===>  in E
  y -> E
```

In this case

- we eliminate dead code.

- we save the cost of entering an expression that is already in head normal form.
- `y` may be used by the constructor reuse transformation.

The first two benefits also happen in the first example.

There is an important variant of this transformation when the `case` expression scrutinises a *variable* which is known to be bound to a constructor. This situation can arise for two reasons:

- An enclosing `let` binding binds the variable to a constructor. For example:
  ```
  let x = C p q in ... (case x of ...) ...
  ```
- An enclosing `case` expression scrutinises the same variable. For example:
  ```
  case x of ...
            C p q -> ... (case x of ...) ...
            ...
  ```
  This situation is particularly common, as we discuss in Section 4.1.

In each of these examples, `x` is known to be bound to `C p q` at the inner `case`. Assuming `x` is bound to `C a1 ... an` The general rules are:

```
case x of {...; C b1 ... bn -> E[b1...bn]; ...} ===> E[a1...an]

case x of {...[no alts match C]...; y -> E[y]}  ===> E[x]
```

3.8 Dead alternative elimination

Dead alternative elimination is similar to the `case` of known constructor transformation, but deals with the case when all we know about a variable is that it in *not* bound to some constructors. Assuming `x` is *not* bound to constructor `C`, we have:

```
case x of C a ... z -> E     ===>  case x of ...[other alts]...
          ...[other alts]...
```

We might know that `x` is not bound to a particular constructor because of an enclosing `case`:

```
        case x of C a ... z -> E1
                  other -> E2
```

Inside `E1` we know that `x` is bound to `C`. However inside `E2` all we know is that `x` is *not* bound to `C`.

This applies to unboxed `case`s also, in the obvious way.

3.9 Case elimination

If we can prove that `x` is not bottom, then the following rule applies:

```
case x of y -> E[y]          ===>  E[x]
```

We might know that `x` is non-bottom because:

- `x` has an unboxed type.
- There's an enclosing `case` which scrutinises `x`.
- It is bound to an expression which provably terminates.

If we apply this transformation even when `x` is not provably non-bottom we can only improve termination; therefore we provide a compiler flag to enable it all the time if the user does not worry about turning a non terminating program into a terminating one.

3.10 Floating `let`s out of a `case` scrutinee

A `let` binding can be floated out of a `case` scrutinee:

```
case (let(rec) Bind in E) of Alts  ===>  let(rec) Bind
                                         in case E of Alts
```

This increases the likelihood of a `case` of known constructor transformation, because `E` is not hidden from the `case` by the `let`.

3.11 Floating `case` out of `case` (`case` of `case`)

Analogous to floating a `let` from a `case` scrutinee is floating a `case` from a `case` scrutinee. We have to be careful, though, about code size. If there's only one alternative in the inner `case`, things are easy:

```
case (case E of {P -> R}) of  ===>  case E of {P -> case R of
  Q1 -> S1                                          Q1 -> S1
  ...                                               ...
  Qm -> Sm                                          Qm -> Sm}
```

If there's more than one alternative there's a danger that we'll duplicate `S1`...`Sm`, which might be a lot of code. Our solution is to create a new local definition for each alternative:

```
case (case E of {P1 -> R1; ...; Pn -> Rn}) of Q1 -> S1
                                              ...
                                              Qm -> Sm
===>
let s1 = \x1 ... z1 -> S1
    ...
    sm = \xm ... zm -> Sm
in case E of
     P1 -> case R1 of {Q1 -> s1 x1...z1;...; Qm -> sm xm...zm}
     ...
     Pn -> case Rn of {Q1 -> s1 x1...z1;...; Qm -> sm xm...zm}
```

Here, `x1 ... z1` are that subset of variables bound by the pattern `Q1` which are free in `S1`, and similarly for the other `si`.

Is this transformation a win? After all, we have introduced `m` new functions! Section 4.3 discusses this point.

3.12 Case merging

```
case x of                                   case x of
  ...[some alts]...             ===>          ...[some alts]...
  other -> case x of                          ...[more alts]...
               ...[more alts]...
```

Any alternatives in `[more alts]` which are already covered by `[some alts]` should first be eliminated by the dead-alternative transformation.

3.13 Eta expansion

We perform general η-expansion when we have an expression with a functional type that has arity greater than the number of lambdas enclosing it:

```
v = \ a b -> f a b      ===>   v = \ a b c -> f a b c
```

(assuming `f` has arity 3). This improves the efficiency as instead of creating a partial application of `f` when `v` is entered, (if it is being called with 3 arguments) `f` will be called directly. This also saves an argument satisfaction check (to check if enough arguments are already available during the call) in some implementations.

Notice that the notion of arity in this case is a bit different from the usual notion, as we do not intend to lose laziness by adding extra arguments to a function. We do not, for example, perform the following transformation:

```
v = \a b -> let x = e   ===>  v = \a b c -> let x = e
            in f x b                         in f x b c
```

as although it can receive 3 arguments (as we assume that `f` receives 3 arguments), if it is partially applied to two arguments it would have very different behaviours in the two expressions above:

- in the first one a closure for `x` is allocated and would be shared by the partial application (if the partial application was applied multiple times), while
- in the second one, as it only does any work after receiving the 3 arguments, the closure for `x` would be allocated and evaluated as many times as the partial application was applied, thus losing laziness.

Therefore the concept of arity we use is not directly related to the maximum number of arguments that a function may receive, but to the number of lambdas in its definition, i.e. the number of arguments that can be passed to the function before it performs any actual "work", like evaluate a `case` or a `let` expression for example.

Actually we sometimes do perform the η-expansion when we have a `case` expression in some circunstances. Let us analyse this case in more detail. assuming $e_1 \ldots e_n$ have a functional type:

```
case e of  p1 -> e1          \y -> case e of p1 -> e1 y
           ...      ===>                      ...
           pn -> en                           pn -> en y
```

- It is a bad idea to do this if e is not a simple variable, because it pushes a redex e inside a lambda. Even if e is a variable, doing this transformation moves an evaluation inside a lambda, which loses a small amount of work for each call of the lambda.

- If any of the e_i are redexes, it would also probably be a bad idea, for the same reason.

But if the two problems above do not occur, in particular if the scrutinee is a variable and therefore the (possible) work duplication is restricted to entering the variable, it is sometimes a very useful transformation.

Therefore the strategy is to do it for `cases` if

- the rhss have functional type

- e is a variable

- all the rhss are manifestly HNFs

3.14 `case` of `error`

`error` is a built-in function of the Glasgow Haskell Compiler, usually associated with pattern matching failures and other sorts of run-time errors. Its semantic value is the same as $\perp$.

Due to transformations like `case` of `case`, for example, we may end up with `error` as a `case` scrutinee, to which we can apply the following transformation

```
  case (error ty E) of Alts   ===>   error ty' E
```

where `ty'` is the type of the whole `case` expression.

3.15 Constructor reuse

The constructor reuse transformation tries to avoid allocating new objects (constructors) when there is an identical object in scope:

```
  case x of                        ===>   case x of
   a:as -> let y = a:as in E1              a:as -> E1[x/y]
```

This transformation may occur in two circunstances:

- there was an identical constructor expression bound by a `let`.

- there was an identical constructor expression bound by a `case` scrutinee.

The main points about this transformation are

- it avoids that an identical object is allocated and built in the heap when an already existing object in the heap can be used instead.

- it may increase the lifetime of objects.

3.16 Default binding elimination

```
case v1 of v2 -> e        ===> case v1 of v2 -> e[v1/v2]
```

Usually the code generator can generate better code if the default variable is not used in its rhs (it does not need to bind the result of the `case` evaluation to the default variable).

3.17 Inlining

The inlining transformation is simple enough:

```
let x = R in B[x]   ===>   B[R]
```

Inlining is more conventionally used to describe the instantiation of a function body at its call site, with arguments substituted for formal parameters. We treat this as a two-stage process: inlining followed by beta reduction (Section 3.1). Since we are working with a higher-order language, not all the arguments may be available at every call site, so separating inlining from beta reduction allows us to concentrate on one problem at a time.

The choice of exactly *which* bindings to inline has a major impact on efficiency. Specifically, we need to consider the following factors:

- Inlining a function at its call site, followed by some beta reduction, very often exposes opportunities for further transformations. We inline many simple arithmetic and boolean operators for this reason.
- Inlining can increase code size.
- Inlining can duplicate work, e.g. if a redex is inlined at more than one site. Duplicating a single expensive redex can ruin a program's efficiency.

Our inlining strategy depends on the form and size of `R` and on the places and number of occurrences of `x`. The algorithm can be described as follows: We inline an occurrence of `x` by `R` in `B` if and only if

- The occurrence is not in an argument position (unless `R` is itself an atom, as arguments can only be atoms)
- if `R` is manifestly a HNF
  - either `x` only occurs once in `B`
  - or `R` is small enough to duplicate
- if `R` is not an HNF
  - no occurrences of `x` are inside a non-linear lambda abstraction
  - either `x` occurs only once in `B`
  - or `x` occurs only once in any control path and `R` is small enough to duplicate

The notion of "small enough" is entirely arbitrary, and can be set by a flag to the compiler. It is mostly based on the syntactical size of the expression, but takes into account other characteristics of the expression.

By control path we mean `case` branches, that is if `R` is small enough and has only one occurrence in different `case` branches we may inline it, as it can only be evaluated once (only one `case` branch will be taken). Notice that if a `case` branch has another `case` expression with `R` occurring only once in each branch it is still only a single occurrence in a given control path.

Linear lambdas are lambdas that the compiler can guarantee (by means of a program analysis) that will only be applied (entered) st most once. This allows non HNF expressions to be inlined in some lambda expressions, otherwise non HNF cannot be inlined into lambda expressions due to the risk that the inlined expression will be evaluated multiple times (losing laziness) if the lambda expression is applied (evaluated) multiple times.

3.18 Dead code removal

If a `let`-bound variable is not used the binding can be dropped:

```
let x = E in B          ===>          B
```

supposing `x` is not used in `B`. A similar transformation applies for `letrec`-bound variables. Programmers seldom write dead code, of course, but bindings often become dead when they are inlined.

4 Composing transformations

The really interesting thing about humble transformations is the way in which they compose together to carry out substantial and useful transformations. This section gives a collection of motivating examples, all of which have shown up in real application programs.

4.1 Repeated evals

The expression `x+x` (where `x` is of type `Int`) in the source code generates the following code in the compiler:

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

due to the inlining of the (boxed) operator `+`, which unboxes its two arguments, applies the primitive (unboxed) operator `+#` to them and finally boxes the resulting value. In this case it unboxes `x` twice, but the `case` of known constructor transformation can eliminate the second evaluation of `x` and generate the code we expect:

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

The transformations in this case are making use of the notion of unboxed data types, as presented in [16].

4.2 Lazy pattern matching

Lazy pattern matching is very inefficient. Consider:

```
let (x,y) = E in B
```

which desugars to:

```
let t = E
    x = case t of (x,y) -> x
    y = case t of (x,y) -> y
in B
```

This code allocates three thunks! However, if `B` is strict in *either* `x` *or* `y`, then the strictness analyser will easily spot that the binding for `t` is strict, so we can do an unboxing `let`-to-`case` transformation:

```
case E of (x,y) -> let t = (x,y) in
                   let x = case t of (x,y) -> x
                       y = case t of (x,y) -> y
                   in B
```

whereupon the `case` of known constructor transformation eliminates the `case` expressions in the right-hand side of `x` and `y`, and `t` is then spotted as being dead, so we get

```
case E of (x,y) -> B
```

4.3 Join points

One motivating example is this:

```
if (not x) then E1 else E2
```

After desugaring the conditional, and inlining the definition of `not`, we get

```
case (case x of True -> False; False -> True}) of
      True  -> E1
      False -> E2
```

Now, if we apply our `case` of `case` transformation we get:

```
let e1 = E1 ; e2 = E2
in case x of True  -> case False of {True -> e1; False -> e2}
             False -> case True  of {True -> e1; False -> e2}
```

Now the `case` of known constructor transformation applies:

```
let e1 = E1 ; e2 = E2
in case x of True  -> e2
             False -> e1
```

Since there is now only one occurrence of `e1` and `e2` we can inline them, giving just what we hoped for:

```
case x of {True -> E2; False -> E1}
```

The point is that the local definitions introduced by the `case` of `case` transformation will often disappear again.

4.3.1 How join points occur

But what if they don't disappear? Then the local definitions will play the role of "join points"; they represent the places where execution joins up again, having forked at the `case x`. The "calls" to the local definitions should really be just jumps. To see this more clearly consider the expression

```
if (x || y) then E1 else E2
```

A C compiler will "short-circuit" the evaluation if `x` turns out to be `True`, generating code like:

```
      if (x) {goto l1};
      if (y) {goto l1};
      goto l2;
l1: ...code for E1...; goto l3}
l2: ...code for E2...
l3: ...
```

In our setting, here's what will happen. First we desugar the conditional, and inline the definition of `||`:

```
case (case x of {True -> True; False -> y}) of True -> E1
                                               False -> E2
```

Now apply the `case` of `case` transformation:

```
let e1 = E1 ; e2 = E2
in case x of True  -> case True of {True -> e1; False -> e2}
             False -> case y    of {True -> e1; False -> e2}
```

Unlike the `not` example, only one of the two inner `case`s simplifies, and we can therefore only inline `e2`, because `e1` is still mentioned twice[1]

```
let e1 = E1
in case x of True  -> e1
             False -> case y of {True -> e1; False -> E2}
```

The code generator produces essentially the same code as the C code given above. The binding for `e1` turns into just a label, which is jumped to from the two occurrences of `e1`.

4.4 `case` of `error`

The `case` of `error` transformation is often exposed by the `case` of `case` transformation. Consider

```
case (hd xs) of {True  -> E1; False -> E2}
```

After inlining `hd`, we get

```
case (case xs of [] -> error "hd"; (x:_) -> x) of True  -> E1
                                                  False -> E2
```

(we have omitted the type argument of `error` for simplicity). Now doing `case` of `case` we get

```
let e1 = E1 ; e2 = E2
in case xs of
     []    -> case (error "hd") of { True -> e1; False -> e2 }
     (x:_) -> case x            of { True -> e1; False -> e2 }
```

[1]Unless the inlining strategy decides that `E1` is small enough to duplicate; it is used in separate `case` branches so there's no concern about duplicating work. Here's another example of the way in which we make one part of the simplifier (the inlining strategy) help with the work of another (`case`-expression simplification).

Now the case-of-error transformation springs to life, after which we can inline e1 and e2 to get the efficient result

```
case xs of []    -> error "hd"
           (x:_) -> case x of {True -> E1; False -> E2}
```

4.5 Nested conditionals combined

Sometimes programmers write something which could be done by a single `case` as a sequence of tests:

```
if x==0::Int then E0 else
if x==1      then E1 else E2
```

After eliminating some redundant evals and doing the `case` of `case` transformation we get

```
case x of I# x# -> case x# of 0#    -> E0
                              other -> case x# of 1#    -> E1
                                                  other -> E2
```

The case-merging transformation puts these together to get

```
case x of I# x# -> case x# of 0#    -> E0
                              1#    -> E1
                              other -> E2
```

Of course many programmers would write a `case` rather than a sequence of conditions in the first place, but sometimes the sequence of tests cannot be eliminated from the source code because of overloading:

```
f :: Num a => a -> Bool
f 0 = True
f 3 = True
f n = False
```

The overloading forces `f` to be compiled to a sequence of conditionals, but if `f` is specialised to `Int` (a common situation) we will get the previous example again.

4.6 Error tests eliminated

The elimination of redundant alternatives, and then of redundant cases, arises when we inline functions which do error checking. A typical example is this:

```
if (x `rem` y) == 0 then (x `div` y) else y
```

Here, both `rem` and `div` do an error-check for `y` being zero. The second check is eliminated by the transformations. After transformation the code becomes:

```
case y# of 0# -> error "rem: zero divisor"
           _  -> case x# rem# y# of
                   0# -> case x# div# y# of r# -> I# r#
                   _  -> y
```

5 Design Issues

5.1 Atomic arguments

At this point it is possible to appreciate the usefulness of the Core-language syntax requirement that arguments are atomic. For example, suppose that arguments could be arbitrary expressions. Here is a possible transformation (supposing `f` is strict in its argument):

```
  f (case x of (p,q) -> p)      ===>  case x of (p,q) -> f p
```

Doing this transformation would be useful, because now the argument to `f` is a simple variable rather than a thunk. However, if arguments are atomic, this transformation becomes just a special case of floating a `case` out of a strict `let`:

```
  let a = case x of (p,q) -> p  ===>  case x of
  in f a                                 (p,q) -> let a=p in f a
```

which is further simplified to

```
  case x of (p,q) -> f p
```

There are many examples of this kind. For almost any transformation involving `let` there is a corresponding one involving a function argument. The same effect is achieved with much less complexity by restricting function arguments to be atomic.

5.2 Renaming and cloning

Every program-transformation system has to worry about name capture. For example, here is an erroneous transformation:

```
  let y = E                         =/=>  let y = E
  in (\x -> \y -> x + y) (y+3)            in (\y -> (y+3) + y)
```

The transformation fails because the originally free-occurrence of `y` in the argument `y+3` has been "captured" by the `\y`-abstraction. There are various sophisticated solutions to this problem but we adopted a very simple one: we uniquely rename every locally-bound identifier on every pass of the simplifier. Since we are in any case producing an entirely new program (rather than side-effecting an existing one) it costs very little extra to rename the identifiers as we go.

So our example would become

```
  let y = E                        ===>  let y1 = E
  in (\x -> \y -> x + y) (y+3)           in (\y2 -> (y1+3) + y2)
```

The simplifier accepts as input a program which has arbitrary bound variable names, including "shadowing" (where a binding hides an outer binding for the same identifier), but it produces a program in which every bound identifier has a distinct name.

5.3 Confluence and Termination

The set of transformations we have presented can be seen as a set of term rewiriting rules. We would like the set of transformations we use to be

- confluent: that we can apply the transformations in any order (when more than one is applicable) and we still get the same result. This is important to make sure that we are not losing transformations or generating worse code by choosing to apply one transformation before another one, when both are applicable.

- terminating: that the process of simplification terminates, meaning that we always get to a point where no transformation is applicable. One has to be particularly careful that one transformation cannot generate code that can be transformed back to the original code by other transformations; that no transformations undo the work of other transformations.

Although we tried to maintain these two properties and experimentally we have verified they are maintained we had no proofs of these properties for quite sometime.

As the transformations are in a very simple left to rigth form with very few side conditions they are good candidates to be treated as rewrite rules in a term rewriting system. In [13] a proof of confluence and termination of a subset of the rules was obtained, using the order-sorted equational theorem proving system MERILL [12], developed at Glasgow University. Initially the system was used to prove confluence and termination for the subset of the rules containing the `let` and `case` floating rules. Later the set was extended to include the constructor reuse, beta reduction and inlining, retaining the same properties.

6 Implementation of the simplifier

The basic algorithm followed by the simplifier is:

1. Analyse: perform occurrence analysis and dependency analysis.

   The occurrence analyser collects information about binders' occurrences, in particular the number of occurrences and their place. This will be used for inlining decisions (Section 3.17). This is "Global" information, therefore it could not be gathered while the simplifier pass is being run.

   Dependency Analysis is needed because while floating `let`s out of `let`s (Section 3.3) we may leave recursive bindings that are not necessarily recursive. As all the information that is needed for performing dependency analysis is already gathered when performing the Occurrence Analysis, we do it together with the Occurrence Analysis.

2. Simplify: apply as many transformations as possible.

3. Iterate: perform the above two steps repeatedly until no further transformations are possible. (A compiler flag allows the programmer to bound the maximum number of iterations.)

We make an effort to apply as many transformations as possible in Step 2. To see why this is a good idea, just consider a sequence of transformations in which each transformation enables the next. If each iteration of Step 2 only performs one transformation, then the entire program will have to be re-analysed by Step 1, and re-traversed by Step 2, for each transformation of the sequence. Sometimes this is unavoidable, but it is often possible to perform a sequence of transformations in a single pass.

7 Measurements

In Table 1 we have a raw count of the number of times some of the transformations we discussed are performed during the compilation of some programs from the *real* part of the **nofib** Benchmark suite ([14]). Knowing the benefits that each transformation presents individually, this gives some idea of the overall effect on the programs.

programs	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**
`compress`	547	35	17	5	3	7917	1	26	2	4
`fluid`	2391	627	42	71	56	1408	46	520	187	170
`gg`	810	384	55	36	21	514	54	384	135	187
`hidden`	509	263	13	9	39	165	33	115	148	33
`hpg`	2059	408	26	18	4	781	9	287	52	47
`infer`	585	164	5	10	13	126	7	148	33	25
`lift`	2023	137	87	7	26	2189	8	151	57	27
`maillist`	177	39	3	5	2	4	2	49	0	5
`parser`	4595	265	52	760	22	2359	63	451	49	298
`prolog`	640	114	20	14	5	139	20	118	28	32
`reptile`	1553	220	56	30	70	620	31	260	79	130
`rsa`	97	47	2	10	3	38	2	30	5	9

Table 1: Transformation Count

a	number of lines
b	inlining (Section 3.17)
c	constructor reuse (Section 3.15)
d	floating `case` out of `let` (Section 3.6)
e	`case` of `case`(Section 3.11)
f	floating `lets` out of `let` (Section 3.3)
g	floating `lets` out of `case` scrutinees (Section 3.10)
h	floating applications inward (Section 3.2)
i	`case` of known constructor (Section 3.7)
j	`let`-to-`case` (Sections 3.4 and 3.5)

Notice that there are many opportunities for transformations like `case` of known constructor, although they are rarely explicitly written in the source code. This is true for many of the transformations, that is they are generated by the compilation process after desugaring and inlining of expressions take place.

Actually many of these transformations were chosen by directly inspecting the intermediate code generated by the compiler.

In Table 7 we can see the effect of the transformations on various programs from the **nofib** Benchmark suite. We measure the number of instructions executed as this is a more reliable and reproducible number than run time. They were measured on a SparcStation 10. As the compiler relies on the use of some of the transformations we presented during the process of desugaring, the base line actually still includes *one* traversal of the code by the simplifier, which will perform a minimal set of transformations, e.g. inlining trivial expressions like `let a = b in E` (where `a` and `b` are variables). These numbers also exclude any benefits from strictness analysis, which is exploited by the compiler using the `let-to-case` and `case` floating from `let` transformations, therefore their effect (which is essentially the effect of strictness analysis as presented in [18]) is not reflected in these measurements.

	Simplified	
	No	Yes
`infer`	383,582,881	0.61
`maillist`	89,490,841	0.71
`parser`	269,289,414	0.75
`prolog`	9,298,035	0.76
`reptile`	64,101,111	0.77
`hpg`	826,552,571	0.77
`fluid`	50,675,650	0.80
`compress`	1,701,504,670	0.84
`lift`	3,155,801	0.85
`veritas`	2,756,900	0.86
`gg`	96,702,693	0.86
`hidden`	9,390,241,691	0.87
`rsa`	1,146,617,118	1.00
Minimum		0.61
Maximum		1.00
Median (13 programs)		0.80
Geometric Mean (13 programs)		0.80

Table 2: Total Instructions Executed

8 Conclusions

We have presented the complete set of local transformations performed by the simplifier pass of the Glasgow Haskell Compiler.

We believe this set of transformations, together with the overall design of the simplifier and the Core Language allows complex transformations to be performed by means of composing simpler transformations.

References

[1] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] A. Appel and T. Jim. Continuation-passing, closure-passing style. In *ACM Conference on Principles of Programming Languages*, pages 293–302, January 1989.

[3] W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, London, March 1990.

[4] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993. ACM Press.

[5] R. J. M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Programming Research Group, Oxford University, July 1983.

[6] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 190–203, Nancy, September 1985. Springer-Verlag.

[7] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *ACM Conference on Principles of Programming Languages*, pages 281–292, January 1989.

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

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

[10] J. Launchbury and P. M. Sansom, editors. *Functional Programming, Glasgow 1992*, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.

[11] S. Marlow and P. Wadler. Deforestation for higher-order functions. In Launchbury and Sansom [10], pages 154–165.

[12] B. Matthews. MERILL: An equational reasoning system in Standard ML. In *5th International Conference on Rewriting Techniques and Applications*, number 690 in LNCS, pages 414–445. Springer-Verlag, 1993.

[13] B. Matthews. Analysing a set of transformation rules using completion. 1994.

[14] W. Partain. The nofib benchmarking suite. In Launchbury and Sansom [10].

[15] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, March 1993.

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

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

[18] S. Peyton Jones and W. Partain. On the effectiveness of a simple strictness analyser. In *Functional Programming, Glasgow 1993*, Ayr, Scotland, 1993. Springer Verlag, Workshops in Computing.

[19] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.