# Secrets of the Yale Haskell Optimizer Revealed!

Sandra J. Loosemore
Department of Computer Science
Yale University
New Haven, CT 06520

July 7, 1993

## 1 Introduction

The expansions produced by the pattern-matcher and type-checker are often quite naive and inefficient. Before generating the output code, we would like to perform various simplifications such as:

- Removing unreferenced variable bindings.
- Dead code removal.
- Constant-folding dictionary lookup.

There are also a number of less obvious transformations that are beneficial because they result in fewer delay thunks or higher-order function calls. These include:

- Saturating curried function calls.
- Inlining functions that take functional arguments.
- Pipelining chains of list transformations.

The optimizer phase is performed after expansion of pattern-matching and before strictness analysis. The optimizer works by performing a code walk over the FLIC representation of the program, looking for various expression patterns, and replacing them with simpler expressions. Since these simplifications can make it possible to do further optimizations, the optimizer makes multiple passes over the FLIC structure.

## 2 Constraints and Non-Constraints

Since Haskell is a purely functional programming language, optimizations that involve code motion or discarding code are much more straightforward than in languages where side-effects are an issue. We don't need to consider order-of-evaluation issues, for example, or whether a variable may have been assigned to between its initial binding and a subsequent reference.

There is one constraint that has to do with the implementation of subsequent passes of the compiler: the optimizer must not produce code that contains shared structure. This means that the optimizer is free to destructively modify the FLIC structure, but it must also be careful not to introduce multiple references. Transformations such as inlining must be careful to make a complete copy of the structure (using `copy-flic-top`).

The function `optimize` takes the FLIC expression to be processed as an argument and returns the (possibly different or modified) expression as a result. Individual walker methods call `optimize` recursively on their components and are responsible for storing the result back into those components.

## 3 Let Expressions and Inlining

By far the most common classes of transformations applied by the optimizer are those dealing with inlining variable references and removing unused bindings from `let` expressions.

The pattern-matching phase introduces many variables that are bound to "simple" expressions such as constants or references to other variables. When walking a `let` expression, the optimizer identifies and marks these variables, and then any references to the variables are inlined by replacing the reference with a copy of its value expression. After all the references have been inlined, the variable binding can be removed.

It is also quite common for a variable to be bound to a more complicated expression, but for the binding to be referenced only once. Inlining these variables involves some tradeoffs; doing so often eliminates the need for creation of a delay thunk or makes it possible to do other optimizations at the point of reference, but if the single reference occurs within a nested function that is called many times the computation may be repeated unnecessarily. We have found that the benefits of inlining outweigh the disadvantages, so these variables are now inlined unconditionally. Note that this optimization takes place over two iterations: one traversal must be done to count the references, and a second to actually do the inlining of the references.

The Haskell programmer can also use annotations to force particular bindings to be inlined. This is especially useful for functions that take functional arguments – for example, `map` or `dropWhile`. Since these functions are usually called with a known value for the functional argument, inlining allows "firstification" of the calls within the function body. (Many of the functions in our version of the Haskell prelude have inline annotations for this reason.)

Inlining of function applications usually triggers a series of other optimizations to remove the bindings of the lambda variables to the actual arguments.

Consider:

```
let f = \ x y -> x : y
in f a b
```

Inlining `f` and simplifying the `let`,

```
((\ x y -> x : y) a b)
```

The next step is to turn the lambda application into a `let`:

```
let x = a
    y = b
in x : y
```

Then, `a` and `b` are inlined and the `let` simplified:

```
a : b
```

# 4 Function Applications

The lambda-to-let conversion noted in the previous section is just one of the transformations that the optimizer looks for in function applications.

The optimizer attempts to saturate all calls to functions of known arity. The purpose of this optimization is to allow a first-class call to the fast entry point for the function to be generated, instead of a call to its slow entry point that supports full currying. The saturation is performed by wrapping a lambda expression around the call. For example, a call such as:

```
foldr f z
```

would be rewritten as:

```
\ l -> foldr f z l
```

Once the call is fully saturated, the optimizer looks for further transformations such as constant-folding. One important special case is inlining calls to dictionary selector functions when the dictionary arguments are constants; this permits a first-class call to the type-specific function to be used instead of having to do the dictionary lookup at run-time. A number of standard prelude functions also have special-case optimizations associated with them. For example, the various numeric conversion functions have optimizers associated with them that do constant-folding at compile-time.

## 5 Lambda Expressions and Function Definitions

The extra lambda variables that are introduced by making function calls saturated are propagated upward through enclosing `let` and `if` expressions and eventually merged with any outer lambda expression.

It is also possible for named local functions to have their arity reduced. This happens when the optimizer notices that there are no higher-order calls to the function and that the same value for the argument is passed at all calls (with appropriate magic to determine fixed points for recursive functions).

Here is an example.

```
dropSpaces = \ l -> dropWhile isSpace l
```

The prelude function `dropWhile` is forced inline because of an annotation. Its expansion includes a locally defined recursive function:

```
dropSpaces =
  \ l ->
    let dropWhile2535 =
          \ ARG2536 ARG2537 ->
            if is-constructor/Nil ARG2537 then pack/Nil
              else
                if ARG2536 (sel/:/0 ARG2537)
                  then dropWhile2535 ARG2536 (sel/:/1 ARG2537)
                  else ARG2537
    in dropWhile2535 isSpace l
```

Next, the second argument (`ARG2537`) is recognized as having an invariant value `isSpace`. Replacing this lambda variable with a local binding is done in two steps. The first step involves adding a new binding for the rewritten function, and redefining the old binding in terms of it:

```
dropSpaces =
  \ l ->
    let dropWhile2542 =
          \ ARG2537 ->
            let ARG2536 = isSpace
            in if is-constructor/Nil ARG2537 then pack/Nil
                 else
                   if ARG2536 (sel/:/0 ARG2537)
                     then dropWhile2535 ARG2536 (sel/:/1 ARG2537)
                     else ARG2537
```

```
        dropWhile2535 =
          \ ARG2536 ARG2537 -> dropWhile2542 ARG2537
    in dropWhile2535 isSpace l
```

In the second step, the original function binding is forced inline and the usual lambda-to-let reductions applied. In this case, the binding of the invariant argument is also recognized as being inlinable. Eventually, this yields:

```
dropSpaces =
  \ l ->
    let dropWhile2542 =
          \ ARG2537 ->
            if is-constructor/Nil ARG2537 then pack/Nil
              else
                if isSpace (sel/:/0 ARG2537)
                  then dropWhile2542 (sel/:/1 ARG2537)
                  else ARG2537
    in dropWhile2542 l
```

# 6 Deforestation

We have implemented a form of deforestation optimization, as described in "A Short Cut to Deforestation" (Andrew Gill, John Launchbury, Simon L. Peyton Jones, FPCA 1993). The general idea is to transform code sequences like

```
filter p (map f l)
```

into a single loop, bypassing construction of the intermediate list.

This is done by defining most of the standard prelude list functions, as well as list comprehensions, in terms of two primitives: `foldr` for consuming lists, and `build` for constructing them. Currently, functions that are candidates for deforestation optimizations must be forced inline via an annotation.

The compression of the loops and the creation of the intermediate list is then performed by this transformation:

```
foldr k z (build g)   =>   g k z
```

In some cases, inlining of list-building functions may leave a `let` construct wrapped around the call to `build` in the function body. In order to permit recognition of the identity, the optimizer hoists the `let` bindings to surround the entire `foldr` call.

Any remaining calls to `foldr` and `build` are inlined. However, in order to permit some additional identities on `foldr` to be detected, `build` must be inlined first and a few more iterations of the optimizer performed to allow the simplifications to propagate through the code. These additional `foldr` identities include:

```
foldr (:) [] l          =>   l
foldr k z []            =>   z
foldr k z x:xs          =>   k x (foldr k z xs)
```

The first transformation is a basic list-copying identity. The latter two identities basically perform loop unrolling on functions applied to list expressions.

Finally, there is one other related special case pattern detected by the optimizer before `foldr` is inlined:

```
foldr (:) z l           =>   primAppend l z
```

Because it is so frequently used, we have implemented `primAppend` as a hand-coded primitive that is specially optimized to append eagerly when possible; in some cases, this can result in noticible performance improvements.

The deforestation optimizations introduce one additional quirk into the optimizer. Specifically, in order for the optimizer to be able to apply this technique by inlining previously compiled functions, the inline expansions must be saved before any calls to `build` and `foldr` they contain are inlined.

# 7 Structured Constants

The optimizer recognizes that applications of data constructors to constant arguments can be made into top-level constants instead of re-evaluated each time. It also detects common constant subexpressions, although in practice this does not have a significant impact on code size.

In the case where a local variable is bound to a structured constant, the entire variable binding is hoisted to top-level. For structured constants that appear in other contexts, a new variable binding at top-level is created.

# 8 Pattern-Matching Simplifications

The optimizer performs a number of optimizations to simplify the pattern-matching code produced by the CFN.

The output of the CFN is rather naive. Basically, for each Haskell `case` expression, it produces a `case-block` construct with clauses (corresponding to each of the alternatives) that are executed sequentially. A `return-from` is used to indicate a successful pattern-match and return from a clause. The `case-block` and `return-from` constructs are fully general but in most cases the pattern matching can be rewritten in terms of simpler `if` expressions.

Consider this example function:

```
foo x = case x of
          []     ->  0
          x1:[]  ->  x1
          x1:xs  ->  x1 + (foo xs)
foo :: [Int] -> Int
```

The CFN produces this output:

```
foo =
  \ x ->
    case-block #:PMATCH2542
      and is-constructor/Nil x
          return-from #:PMATCH2542 0
      and is-constructor/: x
          let CONEXP2543 = sel/:/1 x
          in and is-constructor/Nil CONEXP2543
                 let x1 = sel/:/0 x in return-from #:PMATCH2542 x1
      and is-constructor/: x
          let xs = sel/:/1 x
              x1 = sel/:/0 x
          in return-from #:PMATCH2542 i-Num-Int-+ x1 (foo xs)
      return-from #:PMATCH2542
        error
          "Pattern match failed at line 2 in file interactive."
```

There are some simplifications that are immediately obvious:

- The `is-constructor` tests on `x` in the second and third clauses are unnecessary, since the only other constructor for the list data type is already matched by the first clause.
- The automatically generated final error clause of the `case-block` is unreachable and should be removed.
- Since this example doesn't include any clauses with guards that might fail and fall through to the next clause, the conditional structure can be rewritten in terms of nested `if` expressions.

Performing these transformations (and also inlining the bindings of `x1` and `xs`) results in:

```
foo =
  \ x ->
    if is-constructor/Nil x
      then 0
      else
        if is-constructor/Nil sel/:/1 x
          then sel/:/0 x
          else primPlusInt (sel/:/0 x) (foo (sel/:/1 x))
```

Now let's examine the transformations on the control structure in more detail.

First, optimizations on the individual `case-block` clauses are performed. The CFN generally produces a (possibly nested) `and` expression to perform the pattern-matching tests, interleaved with `let` expressions to bind the pattern variables. There are a number of obvious identities that are recognized in `and` expressions: simplification of literal `True` and `False` subexpressions, nested `and` expressions, and unary `and` expressions.

The optimizer also does some simplifications to remove repeated or unnecessary `is-constructor` tests. If a previous `case-block` clause is already guaranteed to match against a particular constructor (i.e., there are no guards or other subexpressions which must also match), then any further `is-constructor` tests on that constructor will always return `False`. Likewise, if it can be determined that all other constructors for a particular data type must have already been matched by previous clauses, it can be determmined that the remaining `is-constructor` test will return `True`.

Some care must be taken in dealing with clauses that are known to fail to match. For example, it is incorrect to rewrite a clause such as `and exp1 False exp3` as simply `False`, because `exp1` must still be evaluated to preserve the strictness semantics.

Finally, rewriting of the `case-block` as nested `if` expressions is triggered by recognizing a pattern like:

```
case-block name
  (and test1 .. testn  (return-from name result1))
  ...
```

This is rewritten as:

```
if (and test1 .. testn)
   then result1
   else case-block name ...
```

and then the `case-block` in the `else` clause is rewritten recursively. If a clause consists of a single `return-from` expression, any remaining clauses are discarded as dead code.

As usual, if the first clause is a `let` expression, the bindings are hoisted to surround the entire `case-block` to permit these patterns to be recognized. Also, if a `return-from` appears as the body of a `let`, it is hoisted outside of the `let` to enable the `case-block` templates to be recognized. (Usually, most of the bindings of pattern variables and temporaries used for destructuring are optimized away anyway.)

# 9 Miscellaneous

There are a whole bunch of other identities that the optimizer is occasionally able to apply. These include:

- Folding an `if` expression with a constant test.
- Removing redundant `if-constructor` tests on boolean test expressions in an `if`.
- Folding `sel` when applied to a `pack` application.
- Folding `is-constructor` when applied to a `pack` application.
- Folding `is-constructor` when the corresponding data type has only the one constructor.

# 10 Taking Best Advantage of the Optimizer

A number of optimizations performed by the Yale Haskell system involve changes to the standard prelude definitions, so that calls to them may be compiled more efficiently. Users can also benefit by performing some of these same kinds of changes to their own code.

Users can gain the most benefit from the `foldr/build` optimizations by writing their own code in this style — particularly, using `foldr` to iterate over lists rather than writing specific recursive functions. We have rewritten most of the prelude list functions in this style, and we suggest that you look at the `PreludeList` source code for hints on some common idioms for handling additional arguments, iterating over multiple lists in parallel, and the like. Also note that this optimization is performed only within user-defined functions and not across call boundaries unless those functions have explicit inline annotations.

We have added inline annotations to most other prelude functions that take a functional argument as well. Normally, these functions are called with a known argument, and inlining enables the calls to that function within the body to be changed from higher-order calls to more efficient first-order calls. Again, users might want to add inline annotations to their own code in order to take further advantage of this optimization.

Another important use of inline annotations in the prelude is to allow dictionary lookup to be compiled away. The array operations are a good example of this. Computation of array indexing is an overloaded operation in Haskell, involving the `index` operation in class `Ix`. While any calls to `index` with a known dictionary parameter can be optimized to a first-class call to the appropriate method, if the call appears embedded in some other function — for example, the array indexing function `!` — this cannot be done, even if the types of the arguments of particular calls to this enclosing function are known. To avoid the dictionary lookup, the function containing the call must be forced inline with an annotation.

In some cases functions are too large to be inlined without causing excessive code bloat. It is often possible to rewrite such functions so that the part containing the higher-order function call or call to an overloaded operation is inlined, and the result passed to a helper function that is not inlined.

# 11 Tuning and Performance

The total number of iterations performed by the optimizer, as well as the point at which to inline `foldr` and `build`, has been determined by instrumenting the optimizer and experimenting with compiling the standard prelude.

Generally, however, doing this level of optimization on ordinary user code is overkill, particularly since it makes the compilation process noticibly slower. Because of this problem, we have made some of the more time-consuming optimizations optional and disabled by default. Users can re-enable these optimizations either globally by using the optimizers command or menu, or locally by a switch within the unit file for a particular compilation unit.

The optional optimizations are:

**inline** Controls whether to look for functions "simple" enough to be inlined, and whether definitions previously marked as inlinable are actually inlined. If this is disabled, the optimizer is very conservative about inlining.

**constant** Controls whether the optimizer looks for structured constants.

**foldr** Controls whether the optimizer attempts to perform the `foldr/build` deforestation. If disabled, the optimizer performs fewer total iterations and inlines all calls to `foldr` and `build` immediately.