Experiments with strict STG code

Denis B Howe Geoffrey L Burn

Computing Department, Imperial College, London.
E-mail: dbh, glb@doc.ic.ac.uk

October 15, 1992

1 Abstract

Lazy evaluation has many desirable features for the functional programmer but imposes certain overheads on the execution mechanism. Much work has gone into making lazy evaluation efficient. One of the most promising projects in this area is GHC, Glasgow University's Haskell compiler based on the Spineless Tagless G-Machine.

Strictness analysis tells us where we can turn lazy evaluation into strict evaluation without changing the results of the program. Much theoretical work has gone into making strictness analysis more powerful and efficient enough for practical use. Less work has focussed on using the results of strictness analysis to generate better code and it is this that we consider here.

We have taken the results of one kind of strictness analysis, known as evaluation transformers, and used them to transform the lazy code produced by GHC to make it more strict. We describe the execution of lazy and strict code by the STG Machine and explain why the strict code is not always faster. Finally, we suggest alternative code generation strategies and program transformations which might improve the strict code.

2 Introduction — Evaluation transformers

Evaluation transformers[1] tell us how much evaluation we may do to the arguments in a function application in advance without changing the semantics of the program from that implied by lazy evaluation. For primitive types this will just tell us whether we can evaluate it in advance or not, for product (constructed) types such as tuples or lists it will also tell us how much of the structure we can evaluate.

This is useful for two reasons. Firstly, though functional programs appear to offer plenty of parallelism, lazy evaluation is inherently sequential. Evaluation transformers allow us to spawn parallel tasks to evaluate strict arguments without introducing any new non-terminating reduction sequence.

The second potential benefit, and the subject of this paper, is faster sequential execution. Lazy evaluation requires that a closure be built for each argument expression which is itself a function application. We want to avoid the cost of building and evaluating closures wherever possible by evaluating the argument expression before passing it to the function.

We have hand-annotated some test programs to tell the compiler to evaluate certain expressions in-line rather than building closures for them. We have based our experiments on GHC, the compiler produced by the GRASP project at Glasgow University. This compiles Haskell into an intermediate "Core" language and thence, via STG abstract machine code to C. See [2] for a description of STG machine code and how to compile it into C. Note that we only use the term “closure” for an unevaluated function application (which [2] calls a “thunk”) and not for data values.

We shall describe some example functions and the results obtained from making them strict.

3 Example 1

We will take as our first example the function:

```
f x = g (h x)
```

This function exhibits many of the characteristics of more complex examples we have studied but is simple enough that we can follow its evaluation, both lazy and strict, in detail. This is what the STG machine does when `f` is called under lazy evaluation:

1. `f` builds a closure in the heap for the expression `(h x)`,
2. copies `x` to the closure as a free variable,
3. pushes the address of the closure as an argument to `g` and
4. tail-calls `g`.
5. `g` eventually demands its argument, triggering evaluation of the closure.
6. The closure's code pushes an *update frame* (see below),
7. pushes its free variable `x` as an argument to `h` and
8. tail-calls `h`.
9. `h` eventually returns to `g`.
10. The closure `(h x)` is updated.
11. `g` uses its argument and
12. eventually returns to `f`'s caller.

The *update frame* is pushed by the closure to remind it to overwrite itself with its value when it returns, ie. when its tail-call to `h` returns. See [2] for the gory details. If we could prove that `g`'s argument was used only once then we could avoid the expense of an update frame and update but this is not always easy to do.

3.1 Generating strict STG code

Function `f` would normally be compiled into the following STG code:

```
f x = let v = h x in
      g v
```

A `let` in STG code always implies the construction of a closure. If we can determine that `g` is strict (either because it is a primitive or by using evaluation transformer analysis) then `f` can safely evaluate `(h x)` in-line and pass the result to `g`. We rewrite our original Haskell in the above form to make explicit the expressions for which closures would normally be built. We then annotate the closures we want evaluated strictly with a dummy function, `eval`:

```
f x = let v = eval (h x) in
      g v
```

We have written a Core language pass for GHC which transforms `let`s of the form:

```
let v = eval E in B
```

where `v` is a variable and `E` and `B` are expressions, into `case` expressions:

```
case E of v -> B
```

The `case` implies that `E` is evaluated[1] and variable `v` is bound to its value. Applying this pass to our annotated `f` gives the desired strict STG code:

```
f x = case h x of v -> g v
```

When this strict `f` is called the following happens:

1. `f` builds a *black hole* in the heap (see below),
2. pushes the address of the hole (`v`),
3. pushes a continuation address,
4. pushes an update frame for the hole,
5. pushes `x` as an argument to `h` and
6. tail-calls `h`.
7. `h` eventually returns to `f`'s continuation.
8. The black hole is updated.
9. `f`'s continuation pushes `v` as an argument to `g` and
10. tail-calls `g`.
11. `g` demands its (already evaluated) argument,
12. uses its argument and
13. eventually returns to `f`'s caller.

3.2 Result: it's slower

The bad news is that the strict version of `f` actually runs 8% slower than the lazy version. Below are the times for a SPARCstation ELC performing one million evaluations of `(f 42)` with `g x = x` and `h x = x` (all types `Int`). Using more expensive `g` or `h` would simply add the same time to both lazy and strict results. What matters is that `g` is strict.

v	time/s	speed-up
lazy	13.0	0%
strict	14.0	-8%

What has gone wrong?

[1] This is definitely true of STG code. We are assuming that no later Core transformation will turn our strict `case` back into a lazy `let`.

- Building and updating black holes or closures

  The savings from not building and updating a closure for `(h x)` are offset by the cost of building and updating a black hole and pushing its address.

  The black hole is a place to write `v` when it has been evaluated. It could be just an empty heap location but in practise it is set to point to a black hole *Info table* which tells the garbage collector how to handle the hole. Also, if `v` depended on itself the black hole's code would be entered and the program would abort.

- Entering data values or closures

  In the STG machine, a function obtains the value of a variable by *entering* it. This means following several indirections to code which will return the value of the variable. This mechanism works identically for data values in weak head normal form and for unevaluated closures (which is why [2] calls them both "closures").

  Thus the cost of entering the closure `(h x)` the first time it is demanded by `g` is replaced by the equal cost of entering its value.

- Returning data values

  The value of `(h x)` would not normally be returned until it was demanded by `g`. It is now also returned when first evaluated by `f`. This extra return is inevitable since we are only allowed to evaluate `(h x)` in advance because we know `g` will demand it. In the lazy version, `g` enters the closure which then enters `h` with a tail call so there is only one return corresponding to these two enters. In the strict version, `f`'s call to `h` and `g`'s entry of `v` are separate calls with separate returns.

  Returns involve loading the return value into registers (as explained in the next section), popping the return address and performing an indirect jump. This jump is as expensive operation to simulate in C. Making the return mechanism more efficient by using assembly code inserts instead of C will reduce the effect of the extra return but cannot eliminate it completely.

- Rearranging the stack or accessing free variables

  The cost of writing the free variables (just `x` in this case) into the closure and reading them back again is replaced by the cost of rearranging them on the stack to pass them to `h` and *stubbing* any stack slots which are no longer used. Stubbing helps prevent space leaks by overwriting unused pointers with a pointer to a dummy closure similar to a black hole. This may free the previously referenced object for garbage collection.

  The costs are one stack→heap→stack transfer for the lazy version and one stack→stack transfer plus one constant→stack transfer for the strict version. Stack and heap accesses have roughly equal costs in the current implementation.

  This rearrangement and stubbing is only necessary for function `f` because the address of the black hole which has been pushed after `f`'s argument prevents the re-use of that slot (and, in this case, the value in that slot) for `h`'s argument.

4 Returning unboxed values

Why can't we just pass `h`'s result directly to `g`? Why must we go to the expense of building and updating a black hole? The answer lies in the STG machine's return mechanism.

The STG machine is optimised for lazy evaluation where values are only returned when needed. In many cases this means we can avoid building the result in the heap by returning it *unboxed*, in registers. For example, pairs are returned in two registers, `Ret1` and `Ret2` which contain pointers to the boxed components. To cope with sum (multi-constructor) types, the return address actually points to a vector of code addresses with an entry for each of the type's constructors. For example, a return vector for a list type would have entries pointing to code to handle `Nil` and `Cons` values. Different return values can use different registers, eg. none for a `Nil` return, `Ret1` and `Ret2` for a `Cons`.

The problem therefore is that `f` gets `h`'s result unboxed but must pass it to `g` boxed. The easiest way to box the result is to arrange for an update to occur when `h` returns. The update mechanism provides code for each constructor to get the values from the appropriate registers and build the result in the heap. The black hole is created just as something to update.

The next example shows a different way to box an intermediate result.

5 Example 2

Our second example appears rather similar to the first:

```
f x = let (v1, v2) = h x in
      g (v1, v2)
```

Now `h` returns a pair of values which is passed to `g`. In fact, the first example could also have been dealing with pairs but making this explicit has important consequences, as we shall see. Most of the following discussion would apply to more realistic functions which actually did something interesting with `v1` and `v2` rather than just passing them unchanged as a pair but it is useful to be able to compare this example with Example 1.

The following lazy STG code is produced:

```
f x = let p = (h x) in
      let v1 = (case p of (v1', v2') -> v1') in
      let v2 = (case p of (v1', v2') -> v2') in
      let p' = (v1, v2) in
      g p'
```

All these closures look rather redundant but they are necessary if we want `f` to be lazy — nothing is evaluated unless `g` demands one of the components of its argument. If `g` is strict in both components of its argument then we can add the following annotations:

```
f x = let p = (h x) in
      let v1 = eval (case p of (v1', v2') -> v1') in
      let v2 = eval (case p of (v1', v2') -> v2') in
      let p' = (v1, v2) in
      g p'
```

Note that no `eval` is necessary for the `p` closure since it only occurs as the subject of the `case` expressions so its evaluation will be forced anyway. In fact, adding an `eval` for `p` prevents some useful later transformations. The closure `p'` is also special — it builds a data value which is already in normal form — so it doesn't need an `eval` either.

Following the transform implied by the `eval`s and some further ingenious Core optimisations, GHC produces the following rather good STG code:

```
f x = case h x of (v1, v2) ->
      let p' = (v1, v2) in
      g p'
```

This is good because it corresponds to the way that `h` will actually return its results — as an unboxed pair. The explicit `let p' = ...` wraps up `v1` and `v2` in order to pass them to `g` as a single boxed item. It compiles to much better C code than GHC's default black hole update mechanism.

5.1 Result: it's faster

Here are the execution times for one million evaluations of `(f 24)` with `g (x, y) = x + y` and `h x = (x, x)` (all types `Int`).

`v1` & `v2`	time/s	speed-up
lazy	36.2	0%
strict	18.2	50%

So adding `eval`s to Example 2 gives a 50% speed-up. Why is this result so much better than that for Example 1?

- Building and updating black holes or closures

  No closures or black holes are built or updated by the strict version of this function. Only one data value is built — the pair `p'` which is constructed in-line by `f`.

- Entering data values or closures

  The same considerations apply as for Example 1: `g` must enter its argument to get its value whether it is a closure or a data value.

- Returning data values

  As before, there is an extra return of `p'` to `f` in the strict version in addition to the returns demanded by `g`.

- Rearranging the stack or accessing free variables

  With no black hole address on the stack, it would be possible to re-use `f`'s argument (slot) for `h`'s argument. The latest version of GHC does this.

6 Example 3

Our final example is a function which returns a list of `n` zeros. Evaluation transformers tell us that if we wish to evaluate the whole structure of the list then we can annotate the function thus:

```
zeros :: Int -> [Int]
zeros n = case n of
          0 -> []
          _ -> let nm1 = eval (n - 1) in
               let tl = eval (zeros nm1) in
               0 : tl
```

It is interesting to consider the results for lazy and strict evaluation of `nm1` and `tl` independently. The times below are for one evaluation of `(force (zeros 1000000))` where `force` traverses the whole structure of its argument list but does not demand the value of any of its elements.

nm1	tl	time/s	speed-up
lazy	lazy	18.2	0%
strict	lazy	14.8	19%
lazy	strict	22.1	-21%
strict	strict	17.8	2%

Evaluating `(n - 1)` strictly gives a speed-up of 19% whereas evaluating the recursive call to `zeros` strictly gives a 21% slow-down. If we examine the C code produced we discover that, as in Example 1, the slow-down is due to constructing and updating a black hole to box the result of a function call (the recursive call to zeros).

The speed-up is obtained because the strict evaluation of `(n - 1)` can be compiled particularly efficiently. The compiler knows that integer subtraction is a C primitive so it outputs an in-line C expression to perform the operation. `n` will already have been evaluated and unboxed in order to test if it is zero so a single machine instruction suffices to subtract 1 from it and a few more to box the result for passing to the recursive call. This is much more efficient than building and updating a closure for this expression, as is done by the lazy code.

This example shows that, though it is not always beneficial to use all the strictness which can be deduced by evaluation transformer analysis, the analysis is necessary to find the useful strictness. In this case, we were only allowed to evaluate `(n - 1)` because it appeared in a strict context in the recursive call, which appeared in a strict context in the result.

7 Conclusions

It is possible to use strictness information to speed up certain sequential code but strict evaluation is not always faster than lazy evaluation, especially when the compiler and the underlying abstract machine are so well optimised for lazy evaluation. A combination of evaluation transformer analysis and the discoveries presented here will allow us to exploit as much strictness as is beneficial.

GHC's current method of boxing the result of a strict evaluation by updating a black hole is not always optimal. Building the hole is cheap but updates are expensive. We can avoid updates in strict code by providing our own in-line boxing code as in Example 2. We can provide such code for any data type, including integers (Haskell `Int` type) which are returned unboxed as a machine integer in a register but when passed as an argument are normally boxed with a `MkInt` constructor. Things get more complicated if the result has a sum type (eg. a list) because we will need different boxing code for each constructor, as does the update.

We need to know the type of the result (at least its top level constructor) so that we know what kind of return vector to push for it. This applies whether we want to use a black hole update or our own boxing code. This means that we cannot evaluate polymorphic expressions strictly.

Rather than try to make the boxing more efficient, it might be more profitable to try to pass on the results unboxed. We could compile functions to expect their strict arguments unboxed. There are a number of complications:

- Multiple arguments

  If a function wants several arguments to be passed unboxed it might be better to pass them on the stack rather than in registers.

- Higher order functions

  The identity of the function being applied (and thus its argument passing convention) might not be known until run-time. Burn[1], suggests giving each function an extra "prelude" entry point with code to ensure the correct degree of evaluation of each argument.

We could extend this idea to include boxing, ie. when the function is known at compile time, strict arguments can be passed boxed but when the function is entered because it was bound to some variable then we assume all arguments must be boxed and enter its prelude.

- Sum types

  The argument might have a sum type. If we still want to pass it unboxed we would need a different entry point for each constructor. Obviously, we could only do this for one unboxed sum-type argument. If there were several then a mixture of argument passing conventions would be required. This could still be an important optimisation because many functions start by performing a `case` on one argument. The direct entry points would then correspond to the alternatives of that initial `case`. This would avoid doing one `case` to evaluate the argument in advance and another on entry to the function to unbox it and choose the right alternative. This should perhaps be seen more as a program transformation than as a code generation technique.

Kranz's Escape Analysis[3], a technique for optimising the allocation of variables to registers, stack and heap might be relevant here.

An intermediate level of optimisation would be to always pass arguments boxed but guarantee that strict ones would have been evaluated. Functions could then access strict arguments more efficiently than lazy ones because they would not have to cope with the possibility of an unevaluated closure.

The ultimate goal of this work is to use automatic strictness analysis to allow the compiler to transform lazy functional programs into stricter, more efficient programs with the same semantics. The techniques we currently use for exploiting strictness information are not optimal but we have suggested a number of approaches for improving them.

References

[1] Geoffrey L Burn, "Lazy Functional Languages: Abstract Interpretation and Compilation" pub. Pitman in association with MIT Press, 1991
Research Monographs in Parallel and Distributed Computing.

[2] Simon L Peyton Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine"
Department of Computing Science, University of Glasgow G12 8QQ.
15 April 1992
FTP: ftp.dcs.glasgow.ac.uk
/pub/glasgow-fp/papers/spineless-tagless-gmachine.dvi

[3] Kranz, David A, "Orbit: An Optimising Compiler for Scheme"
YALEU/DCS/RR-632
February 1988