Sharing Analysis + EVAL inlining + Unboxing = Deforestation

Thomas Johnsson

Abstract

We show by example that by combining sharing analysis, inlining of `EVAL`, and unboxing specialisation, of GRIN code, we can achieve deforestation (listlessness) transformation.

1 Introduction

In the GRIN (Graph Reduction Intermediate Notation) code compilation project, currently being pursued by Urban Boquist and myself, we're poised to take the next leap forwards in the execution speed of lazy functional languages![1]

More specifically, we aim at achieving better back end code generation than in current compilers for lazy functional languages, such as the Clean compiler, the Chalmers Haskell compiler, or the Glasgow Haskell compiler. Our approach can be summarised as follows.

- Supercombinators are compiled into an intermediate code form called GRIN (Graph Reduction Intermediate Notation), which is essentially a procedural form of G-machine code, using intermediate variables instead of stack (an abundance of examples will follow!)
- We explore state-of-the-art register allocation techniques, especially *interprocedural register allocation*, resulting in register allocation of arguments of function tailor-made to each function — a prerequisite for achieving as good register allocation as for loops in in imperative languages.
- We perform transformation of the GRIN code program, especially *inlining* of calls to `EVAL`, .i.e, replacing a call to `EVAL` by a case statement and individual calls to various known functions. Not only does it eliminate the call overhead of `EVAL`, it also increases the sise of the code portions voer which variables might usefully reside in registers, and it also uncovers further possibilites of exploiting tailor-made calling conventions of functions. This is described in [Boq95a, Boq95b].
- Further transformation is possible (and useful!) on the GRIN code level, such as further inlining of functions, unboxing, and deforestation.
- Transformations are greatly aided by an analysis of the GRIN code, called *constructor analysis*, which gives a safe approximation of what pointers might point to in the GRIN program. The analysis described in [Joh91] turned out to be too slow to be useful, but the most recent version is deemed to be fast and accurate enough for our purposes (not written up yet). The latest version also includes a simple form of sharing analysis,

[1] Modesty has never been a virtue of papers aimed for publication...

In this paper we first give an overview of the basic techniques for compilation to GRIN code, and inlining of EVAL. We then show how further inlining, plus unboxing specialisation [PJL91], plus exploiting sharing analysis information, can result in *deforestion* (listlessness) [Wad84, Wad88] almost as a byproduct ...

2 The basic approach explained

In this section we explain basic compilation into GRIN code, application of program analysis, and the inlining of `EVAL` calls (aided by the analysis).

2.1 Basic compilation into GRIN code

Consider the following program fragment, evaluating the sum of the numbers $1 \ldots 10$:

```
sum a [] = a
sum a (x:xs) = sum (a+x) xs
upto m n = if m>n then [] else m : upto (m+1) n
main = sum 0 (upto 1 10)
```

For the sake of the example, we use an accumulating version of `sum`. To make the example slightly more interesting in what follows, let us also assume that the `sum` function is also use at other place in the program.

The GRIN code, which is a procedural version of three address code, is a highly flexible intermediate form for compilation of heap-based languages, and it is possible to compile into GRIN in very many different ways, embodying spinelessness or not, differerent forms of tagging, etc.

The form of translation we use below is rather mundane, essentially in the same style as conventional G-machine translation. Each supercombinator becomes a GRIN procedure. Arguments of functions, evaluated or unevaluated, are put in boxes in the heap and pointers to these boxes are passed as the actual arguments (i.e., no fancy tagging or unboxing for now, nor is any strictness analysis assumed). Procedures returns a node as a result (and I do mean a node, not a pointer to one); no updating is done in procedures for supercombinators (initially), that is done by `EVAL`.

Below we show how one might typically translate the program above into GRIN code. We show it as a *state monadic* functional program. `unit` is the unit in the monad, ; is the bind. `store`, `fetch` and `update` are operations particular to this monad.

```
main =
    store (Cint 1); \t1 ->
    store (Cint 10); \t2 ->
    store (Fupto t1 t2); \t3  ->
    store (Cint 0); \t4 ->
    sum t4 t3; \(Cint r') ->
    print_int r' ;\() ->

    (.... use sum, t4 some more ....)

upto m n =
    EVAL m; \(Cint m') ->
    EVAL n; \(Cint n') ->
    if m' > n' then
        unit Cnil
    else
        store (Cint (m'+1)); \m1 ->
        store (Fupto m1 n); \p ->
        unit (Ccons m p)

sum a l =
    EVAL l ; \ll ->
    case ll of
    Cnil ->
        EVAL a ; \aa ->
        unit aa
    Ccons x xs ->
        EVAL a; \(Cint a') ->
        EVAL x; \(Cint x') ->
        store (Cint (a'+x')); \ax ->
        sum ax xs
```

All objects on the heap are referred to as *constructors*; it is the GRIN program which interprets them either a representing ordinary values (`Cint`, `Cnil` and `Ccons` above), or unevaluated expressions (`Fupto`).

An essential feature of our approach is that `EVAL`, which is normally hidden in the runtime system or by some 'tagless' pointer dispatch, also becomes a GRIN procedure — and thus susceptible to transformation!

The standard `EVAL` fetches the node pointed at and performs case scrutinisation. This `case` must enumerate all possible nodes that could ever occur in the program, and either return the node value (if canonical), or call the appropriate procedure to do the evaluation job and the update (if non-canonical).

The `EVAL` for the above program might look like this:

```
EVAL l =
    fetch l; \ll ->
    case ll of
    Cint x'    -> unit ll
    Cnil       -> unit ll
    Ccons x xs -> unit ll
    Fupto m n  -> upto m n; \v ->
                  update l v; \() ->
                  unit v
    Fsum a l   -> sum a l; \v ->
                  update l v; \() ->
                  unit v
    (... other possible cases ...)
```

It is perfectly possible and safe to be content with the above GRIN program, and subject it to the low level target code generator/register allocator, and still be able to do a fair job with it [Boq95b]. However, the main point of this work is to do some further transformations, notably inlining of `EVAL`.

2.2 Constructor analysis

It is perfectly possible to inline each and every call of `EVAL` with the body of the `EVAL` procedure, as described above. However, then each such `case` would typically mention a large number of cases which are 'impossible' at this particular point — if nothing else, for type reasons! For instance, if `EVAL` is performed prior to an addition, then that `EVAL` would expect either a `Cint` node of the closure of a function that returns a `Cint` node (e.g., `Fsum` in our example). Thus in order to avoid code explosion, it is highly desirable to weed out as many impossible cases as possible from each such `case`. Our *Constructor analysis* aims at remedying this. It is a form of pointer analysis, to find out what pointer might point to at all places in the program.

Our analysis is based on *abstract locations*, and each `store` operation returns the same abstract location each time. Thus the `store` operations of the program are simply consecutively numbered and annotated with the abstract location they return. For instance, the `store(Fupto m1 n)` in `upto` returns the abstract location 6. Basic values are abstracted to 0. For reasons of efficiency and accuracy of the analysis, The procedure `EVAL` is not analysed, instead the analysis assumes the 'standard behaviour'. Below the binding occurrences of the variables are annotated with their possible values inside `{ }`, as inferred by the current version of our constructor analysis. Since variables are either pointer-valued or node-valued, possible abstract values ar either sets of abstrac pointers (numbers), or sets of abstract nodes (e.g. the varible `ll` in `sum`).

```
main =
    store (Cint 1); \t1{1} ->
    store (Cint 10); \t2{2} ->
    store (Fupto t1 t2); \t3{3}  ->
    store (Cint 0); \t4{4} ->
    sum t4 t3; \(Cint r{0}) ->
    print_int r

upto m{1,5} n{2} =
    EVAL m ; \(Cint m'{0}) ->
    EVAL n ; \(Cint n'{0}) ->
    if m' > n' then
        unit Cnil
    else
        store (Cint (m'+1)); \m1{5} ->
        store (Fupto m1 n); \p{6} ->
        unit (Ccons m p)

sum a{4,7} l{3,6} =
    EVAL l ; \ll{Ccons[{1,5},{6}], Cnil[]} ->
    case ll of
    Cnil ->
        EVAL a ; \aa ->
        unit aa
    Ccons x{1,5} xs{6} ->
```

```
EVAL a; \(Cint a'{0}) ->
EVAL x; \(Cint x'{0}) ->
store (Cint (a'+x')); \ax{7} ->
sum ax xs
```

The analysis also infers the following *abstract store* (i.e. mapping from abstract locations to abstract nodes):

```
1 := {Cint[{0}]}
2 := {Cint[{0}]}
3 := {Fupto[{1},{2}], Ccons[{1,5},{6}], Cnil[]}
4 := {Cint[{0}]}
5 := {Cint[{0}]}
6 := {Fupto[{5},{2}], Ccons[{1,5},{6}], Cnil[]}
7 := {Cint[{0}]}
```

Originally, a `Fupto` node is stored in abstract locations 3 and 6. But since the value of `upto` is either a Nil or Cons, these abstract locations are updated with these values as well, since `EVAL` might potentially see them (more about this later, on the topic of sharing analysis).

2.3 `EVAL` inlining

The result of the constructor analysis is used to decide how to inline `EVAL`. A few different cases can be distingushed:

Canonical constructor(s) only: Consider the `EVAL a` in `sum`. The variable `a` can be either abstract location 4 or 7, which are both `Cint`s only! Thus the `EVAL a` can be replace by `fetch a`. In general, this can be done when the pointer being `EVAL`ed can only point to any of several different `canonical` constructors (since no evaluation is necessary).

A single non-canonical constructor: For instance, if `l` could only point to a `Fupto` node, then

```
EVAL l
```

could be replaced by

```
fetch l; \(Fupto m n) ->
upto m n; \v ->
update l v; \() ->
unit v
```

No case scrutinisation is necessary, but the corresponding procedure must be called and the node updated.

Otherwise , The `EVAL` call is replaced by a full-blown `fetch`-and-`case`, which includes the possible cases.

Thus the procedure `sum` becomes like this then all its `EVAL`s have been inlined:

```
sum a l =
    (fetch l; \ll ->
     case ll of
     Cnil ->
        unit ll
     Ccons x xs ->
        unit ll
     Fupto m n ->
        upto m n ; \t6 ->
        update l t6; \() ->
        unit t6
     (... other cases ...)
    ); \ll ->
    case ll of
    Cnil ->
        fetch a
    Ccons x xs ->
        fetch a; \(Cint a') ->
        fetch x; \(Cint x') ->
        store (Cint (a'+x')); \ax ->
        sum ax xs
```

For the purpose of improving conditions for interprocedural register allocation, this is sufficient: we have replaced general `EVAL` calls, which we don't know what procedures they will actually call, with explicit calls to known functions [Boq95b], enabling better use of tailormade register argument passing conventions.

2.4 Further simplification of the code

But further simplifications/improvements of the `sum` immediately suggest themselves! If we copy the second `case` into each of the branches of the first `case`, yielding

```
case ll of
Cnil ->
    unit ll; \ll ->
    case ll of
    Cnil ->
        (Cnil case)
    Ccons .... ->
        (Ccons case)
Ccons ->
    unit ll; \ll ->
    Cnil ->
        (Cnil case)
    Ccons .... ->
        (Ccons case)
Fupto m n ->
    ....
```

then this obviously simplifies to

```
case ll of
Cnil ->
    (Cnil case)
Ccons ->
    (Ccons case)
Fupto m n ->
    ....
```

Thus the entire `sum` procedure simplifies to:

```
sum a l =
    fetch l; \ll ->
    case ll of
    Cnil ->
        fetch a
    Ccons x xs ->
        fetch a; \(Cint a') ->
        fetch x; \(Cint x') ->
        store (Cint (a'+x')); \ax ->
        sum ax xs
    Fupto m n ->
        upto m n ; \t6 ->
        update l t6; \() ->
        case t6 of
        Cnil ->
            fetch a
        Ccons x xs ->
            fetch a; \(Cint a') ->
            fetch x; \(Cint x') ->
            store (Cint (a'+x')); \ax ->
            sum ax xs

    (... other cases ...)
```

2.5 Inlining of 'conventional' calls

It is of course possible to inline also calls to 'conventional' procedures. So let us inline the call of `upto` in the `sum` procedure:

```
sum a l =
    fetch l; \ll ->
    case ll of
    Cnil ->
        fetch a
    Ccons x xs ->
        ( ... )
    Fupto m n ->
        (
        fetch m ; \(Cint m') ->
        fetch n ; \(Cint n') ->
        if m' > n' then
            unit Cnil
        else
            store (Cint (m'+1)); \m1 ->
            store (Fupto m1 n); \p ->
            unit (Ccons m p)
        ); \t6 ->
        update l t6; \() ->
        case t6 of
        Cnil ->
            fetch a
        Ccons x xs ->
            fetch a; \(Cint a') ->
            fetch x; \(Cint x') ->
            store (Cint (a'+x')); \ax ->
            sum ax xs

     (... other cases ...)
```

The `if` from the original `upto`, which returns either a `Cnil` or a `Ccons`, is followed by an update — but also a `case` which scrutinises this value! Thus we move/copy the second case into the branches of the `if`, and simplify (also eliminating a redundant `fetch m`).

```
sum a l =
    fetch l; \ll ->
    case ll of
    Cnil ->
        fetch a
    Ccons x xs ->
        ( ... )
    Fupto m n ->
        fetch m ; \(Cint m') ->
        fetch n ; \(Cint n') ->
        if m' > n' then
            update l Cnil; \() ->
            fetch a
        else
            store (Cint (m'+1)); \m1 ->
            store (Fupto m1 n); \p ->
            update l (Ccons m p) \() ->
            fetch a; \(Cint a') ->
            store (Cint (a'+m')); \ax ->
            sum ax p

     (... other cases ...)
```

In the `Fupto` case, we have effectively merged the code of `sum` and the code of `upto`, while eliminating many redundant case scrutinisations — this is effectively a compile-time version of the vectored return mechanism of the Spineless Tagless G-machine [PJS89].

2.6 Sharing analysis

It has turned out to be easy to augment then current constructor analysis with a simple form of sharing analysis. The current analysis this also infers which abstract locations that might be shared.

So if the original program example was not a fragment, but the *whole* program, our constructor analysis would infer the following abstract store:

```
*1 := {Cint[{0}]}
*2 := {Cint[{0}]}
 3 := {Fupto[{1},{2}]}
 4 := {Cint[{0}]}
*5 := {Cint[{0}]}
 6 := {Fupto[{5},{2}]}
 7 := {Cint[{0}]}
```

The locations marked with a * are possibly shared, the others are unique in the sense the `EVAL` (or a `fetch`) will only look at such a value once. The missing `Cnil` and `Ccons` cases of locations 3 and 6 reflect the fact that the first (and only!) time `EVAL` is performed on such a location they contain what was originally put there by a `store (Fupto ...)`, the result of subsequent updating is never seen.

2.7 Exploiting sharing analysis

Since the result of a subsequent updating is never seen, it is of course redundant to do the updating! So in fact the original `EVAL l` in `sum` can be replaced by

```
fetch l; \(Fupto m n) ->
upto m n
```

thus avoiding the case scrutinisation altogether.

However, for the sake of being slighly more general, we continue to assume that the orginal `sum` function was used at other places in the program, so that the `EVAL l` could encounter other nodes than just `Fupto`, but that the `Fupto` node is non-shared. Then the `EVAL l` expands to

```
(fetch l; \ll ->
 case ll of
 Fupto m n ->
    upto m n
 (...other cases...)
); \ll ->
```

and the `sum` function eventually transforms to:

```
sum a l =
    fetch l; \ll ->
    case ll of
    Fupto m n ->
        fetch m ; \(Cint m') ->
        fetch n ; \(Cint n') ->
        if m' > n' then
            fetch a
        else
            store (Cint (m'+1)); \m1 ->
            store (Fupto m1 n); \p ->
            fetch a; \(Cint a') ->
            store (Cint (a'+m')); \ax ->
            sum ax p
    (... other cases ...)
```

i.e., just as the previous version, except that the updating of `l` is not needed.

3 The new idea: unboxing yields deforestation

We now notice that all `store` operations (in the `Fupto` branch) are immediately fetched again in the next call of `sum` — they are used solely for transmitting argument values in boxes! Thus we can make a specialised 'unboxed' version of sum. In a functional notation, we would make the following 'Heureka' definition:

$$\mathtt{sumupto}\, a'\; m'\; n' \equiv \mathtt{sum}(\mathtt{Cint}\, a')(\mathtt{Fupto}(\mathtt{Cint}\, m')(\mathtt{Cint}\, n'))$$

```
sum a l =
    fetch l; \ll ->
    case ll of
    Fupto m n ->
        fetch m ; \(Cint m') ->
        fetch n ; \(Cint n') ->
        fetch a ; \(Cint a') ->
        sumupto a' m' n'

    (... other cases ...)

sumupto a' m' n' =
    if m' > n' then
        unit (Cint a')
    else
        unit (m'+1); \m1' ->
        unit (a'+m'); \ax' ->
        sumupto ax' m1' n'
```

The `sumupto a' m' n'` function is a complete 'listless' version of the `sum a' (upto m' n')` function composition — i.e., no intemediate list is used.

Thus we reach the conclusion, that

> *unboxing of unevaluated closures, together with sharing analysis, can achieve deforestation (listlessness) —almost for free.*

4 Distinguising shared and unshared at runtime

To be able to make a better distiction at runtime between shared and unshared closures, the following trick should be highly effective.

It could well be the case that at the point of inlining an `EVAL`, the `EVAL`ed pointer could point to e.g. an `Fupto` closure, but it could come from two different abstract locations, one shared and one un-shared. Just looking at the tag of the closure will not be sufficient to determine whether it is unshared or not.

Abstract locations in the heap is synonymous with original store points in the GRIN program. After sharing analysis, it is known whether a specific `store` will be shared or not. Thus the trick is:

> *Use two different tags for closures, one denoting a possibly shared closure, and one denoting a surely un-shared closure.*

For example, use

```
store (Fupto_u ....); \p-> ...
```

for stores that will be unshared, and

```
store (Fupto ....); \p-> ...
```

that might be shared. Then at the point of inlining `EVAL`, `EVAL l` will be inlined with:

```
(fetch l; \ll ->
 case ll of
 ...
 Fupto m n ->
    upto m n;\t->
    update l t;\()->
    unit t
 Fupto_u m n ->
    upto m n
 ...
); \ll ->
```

References

[Boq95a] Urban Boquist. Interprocedural Register Allocation for Lazy Functional Languages. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.

[Boq95b] Urban Boquist. Interprocedural Register Allocation for Lazy Functional Languages. Licentiate Thesis, Chalmers University of Technology, Mars 1995.

[Joh91] Thomas Johnsson. Analysing Heap Contents in a Graph Reduction Intermediate Language. In S.L. Peyton Jones, G. Hutton, and C.K. Holst, editors, *Proceedings of the Glasgow Functional Programming Workshop, Ullapool 1990*, Workshops in Computing, pages 146–171. Springer Verlag, August 1991.

[PJL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture*, Sept 1991.

[PJS89] S. L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[Wad84] Phil Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile time. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 45–52, Austin, Texas, 1984.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358, Nancy, March 1988.