The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages

Urban Boquist and Thomas Johnsson

Department of Computing Science
Chalmers University of Technology
Göteborg, Sweden
E-mail: {boquist,johnsson}@cs.chalmers.se

Abstract. Low level optimisations from conventional compiler technology often give very poor results when applied to code from lazy functional languages, mainly because of the completely different structure of the code, unknown control flow, etc. A novel approach to compiling laziness is needed.
We describe a complete back end for lazy functional languages, which uses various interprocedural optimisations to produce highly optimised code. The main features of our new back end are the following. It uses a monadic intermediate code, called GRIN (Graph Reduction Intermediate Notation). This code has a very "functional flavour", making it well suited for analysis and program transformations, but at the same time provides the "low level" machinery needed to express many concrete implementation concerns. Using a heap points-to analysis, we are able to eliminate most unknown control flow due to evals (i.e., forcing of closures) and applications of higher order functions, in the program. A transformation machinery uses many, each very simple, GRIN program transformations to optimise the intermediate code. Eventually, the GRIN code is translated into RISC machine code, and we apply an interprocedural register allocation algorithm, followed by many other low level optimisations. The elimination of unknown control flow, made earlier, will help a lot in making the low level optimisations work well.
Preliminary measurements look very promising: we are currently twice as fast as the Glasgow Haskell Compiler for some small programs. Our approach still gives us many opportunities for further optimisations (though yet unexplored).

1 Introduction

Although the execution speed of programs written in a lazy functional language, like Haskell, have increased substantially since these languages first appeared, it is still the case that they are slower and consume more memory than imperative programs, in almost all cases.

The reason for functional programs being so slow, is, of course, that functional languages in general, and lazy languages in particular, are so abstract and "far

from the machine". Thus, it is very hard for the compiler to optimise the program with good results. Put in another way, we can say that the laziness has a, not negligible, runtime cost.

One of the purposes of this paper is to show how this cost can be reduced by doing more aggressive optimisations than current compilers do. As part of that we will attack the well known problem that conventional (imperative) compiler optimisations do not apply very well to code produced from a lazy functional language, or, if they apply, produce far from satisfactory results. As we will later show, one important reason for this is the laziness, or rather those properties of the generated code that encode the lazy evaluation strategy (e.g. building and forcing *delayed computations*).

Our first, and most important, principle for solving this problem is to do *interprocedural* optimisation, i.e., we let the compiler optimise several procedures together (currently the whole program at once). This should be seen in contrast to the standard method of *global* optimisation, where only one procedure is optimised at a time,[1] a method that is often quite sufficient for imperative programs. This will be explained in more detail in section 2.

In this paper we will describe a novel back end for a compiler for a lazy functional language. The most interesting features of this back end are:

- It is interprocedural, aiming at much more aggressive optimisations than current compilers do.
- The intermediate code, called GRIN (Graph Reduction Intermediate Notation), has a very "functional flavour", which makes it well suited for analysis and program transformations. But, at the same time, it has the "low level control" that is needed.
- Using a two step process: a heap points-to analysis + a single GRIN program transformation, we are able to eliminate most *unknown control flow* (or actually give a good approximation to the real control flow), by inlining calls of `eval` and `apply`, in the program.
- The GRIN code is compiled (and optimised) using a series of, each very simple, GRIN source-to-source program transformations, which taken together will produce greatly simplified code.
- With the GRIN transformations as a basis, the resulting (machine) code will be of a form that is suitable for conventional optimisation techniques. In particular we use an interprocedural register allocation algorithm, developed especially with *call intensive* languages in mind.

The organisation of the rest of this paper is as follows. In section 2 we elaborate on the problem of implementing lazy evaluation, and try to motivate why interprocedural compilation is so important. Then, in section 3, we describe the overall structure of our compiler, and back end. In sections 4 to 7 we introduce the intermediate code, GRIN, and describe how it is compiled and optimised using program transformations. The particular heap analysis used is discussed

[1] We use these terms as found in most compiler literature, i.e. *global* does not really mean global...

```
main = sum (upto 1 10)

upto m n = if m > n then []
                    else m : upto (m+1) n

sum l = case l of  []     -> 0
                   (x:xs) -> x + sum xs
```

Fig. 1. A small program, summing the numbers from 1 to 10.

in section 8. After all GRIN transformations, the code is translated into real machine code, described in section 9, and a number of low level optimisations are applied; the register allocation is described in section 10 and an overview of the other optimisations is given in section 11. We present some preliminary results in section 12. Finally, we conclude with related work and further development of our back end.

2 Lazy evaluation

To explain why lazy evaluation hinders optimisation and to show how interprocedural compilation can be a first step in solving this we will discuss a small example, the program in figure 1. This program will also be used as the running example throughout this paper.

The program is written using a syntax similar to Haskell. If we had written this program in an imperative language (and using an imperative style) we would most certainly have used real **loops** to sum the numbers, because we know that imperative compilers are good at optimising loops, and often rather poor at optimising procedure calls.

If we imagine the program as written in a *strict* functional language, its execution would result in a *call graph* as the one in figure 2.

We define a node in the call graph as the union of all invocations of the corresponding function. An arc in the call graph means that a function call may occur in the direction of the arrow. Note that call graphs are approximations to what will happen in an execution of the program (but always safe approximations).

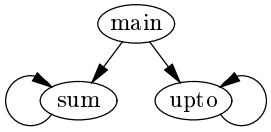

Fig. 2. The "strict" call graph

Returning to our example program, the strict call graph illustrates what will happen in a strict execution of the program. The `main` function will call the `upto` function which will produce a list of numbers. The `upto` function will create this list using recursion (i.e. a kind of loop). It will eventually return to `main`, which will directly call `sum`. The `sum` function will consume the list, also using recursion, and sum up the numbers (i.e a second loop). In this strict version of the program, the two

loops are still quite "visible". We could imagine a compiler noticing that both `sum` and `upto` make recursive calls, and try to optimise this "as a loop".

However, if we turn to the call graph for the same program when executed in a lazy language it will look quite different, and much less attractive from a compilers point of view (figure 3).

Here, we imagine a standard implementation of lazy evaluation using *graph-reduction*. One additional procedure is added to the call graph, the special `eval` procedure. This is the procedure used to *force* (or evaluate) a suspended computation. Even though this is normally hidden in the runtime system of an implementation, we can think of `eval` as an ordinary procedure, which will turn its argument into *weak head normal form*. If, in the call graph, a particular procedure calls `eval` it will mean that the procedure needs the value of a *closure* (which might be a suspended computation). On the other hand, if `eval` calls a procedure, it means that a suspended computation of that procedure was forced (by someone else).

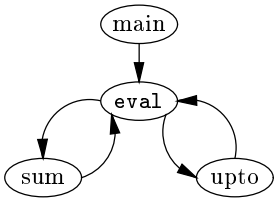

Fig. 3. Original "lazy" call graph

There are a number of different ways to implement and optimise this "forcing" (see for example [Joh84,PJ92]), but they all have one thing in common: the code will have to do an "unknown call" when it is faced with a suspended computation. By this we mean that it is *unknown at compile time* to which procedure such a call will jump. In our call graphs this will be seen as first a "call" to `eval`, and then a new call from `eval` to the suspended procedure.

Unfortunately, these unknown calls are one of the main reasons why conventional compiler optimisations will give so poor results for lazy functional languages. When the compiler is faced with an unknown call (i.e. unknown control flow), it will normally have to make the most pessimistic assumptions possible, like for example not allowing any values to be held in registers. And, since the functional programming style encourages "small" functions, it is not surprising that a global optimiser, that can only optimise the code between two calls at a time, will give so poor results in most cases.

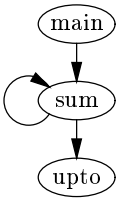

Fig. 4. Improved "lazy" call graph

The consequence of this is that if we want to use conventional optimisations effectively, we will **have to** eliminate most (or all) of the "unknown control flow". The way we do this is described in section 5. Seen in the call graph, it will have the effect of completely eliminating the `eval` procedure, and replacing each arc to `eval` with a safe superset of "real" calls, i.e., arcs to ordinary procedures. After this, we will get the call graph in figure 4.

This illustrates the "looping" behaviour that will actually happen in a lazy evaluation of this program. The loop will be in the `sum` function (using recursion) and, once each iteration, it will call the `upto` function to produce the next number

(i.e. a new cons cell). The call graph also makes clear the producer/consumer relationship, that is so typical of lazy evaluation, between the `upto` and `sum` functions. An aggressive optimiser that is allowed to optimise the `sum` and `upto` functions together could take advantage of this knowledge and produce much better code compared to the original program (with `eval`).

3 Our compiler

To be able to compile real Haskell programs, we use our back end in conjunction with an already existing front end, hbcc, by Lennart Augustsson [unpublished]. Hbcc is a state of the art Haskell front end. Using hbcc we get well optimised code in a "low level functional" style, comparable to for example the *Core* language [PJ96] used by the Glasgow Haskell compiler. The code is lambda lifted, i.e., has only super combinators, and most "high level" Haskell constructions, like overloading, have been transformed away.

The structure of the back end (extended with hbcc) can be seen in figure 5. The front end, i.e., hbcc, uses standard separate compilation. Our back end (which is a stand alone program) will collect the code produced from all hbcc-compiled files (for a program spread over several files) and optimise the whole program at once. Thus, the entire system uses separate compilation in the front end, whereas the GRIN back end (currently) need the whole program at once.

The first part of the back end uses the intermediate code, GRIN, and gradually transforms and optimises the code into a very simple form. After that, the code is translated into machine code for a hypothetical RISC machine, and the second part of the back end uses this RISC code. After the low level optimisations, the RISC code is finally "pretty printed" as assembler code for the Sparc processor. However, the RISC code is not very Sparc-specific, so it would not be a large project to generate code for a different processor. Also, the optimisations done are mostly "generic" in nature, and would apply to any RISC processor.

4 GRIN - the intermediate code

The purpose of the GRIN intermediate code is the same as for the G-machine [Joh84] code: to provide a framework and vehicle for compilation of lazy functional languages. Thus, GRIN provides similar primitives as the G-machine does, but does it on a slightly higher level. GRIN can be thought of as a procedural language, where statements inside procedure bodies are essentially three-address code. GRIN is quite flexible: it is possible to compile lazy functional languages in a variety of ways, with different forms of tagging, unboxing, etc. The GRIN code is actually quite a general form of intermediate language, which could be a suitable intermediate form for compilers of many 'heap based' languages (e.g. Lisp, SML, possibly even languages like Smalltalk), although GRIN has some special provisions to accommodate for lazy evaluation.

We will continue to use the program in figure 1 as our running example. Figure 7 shows how it can be translated into GRIN code.

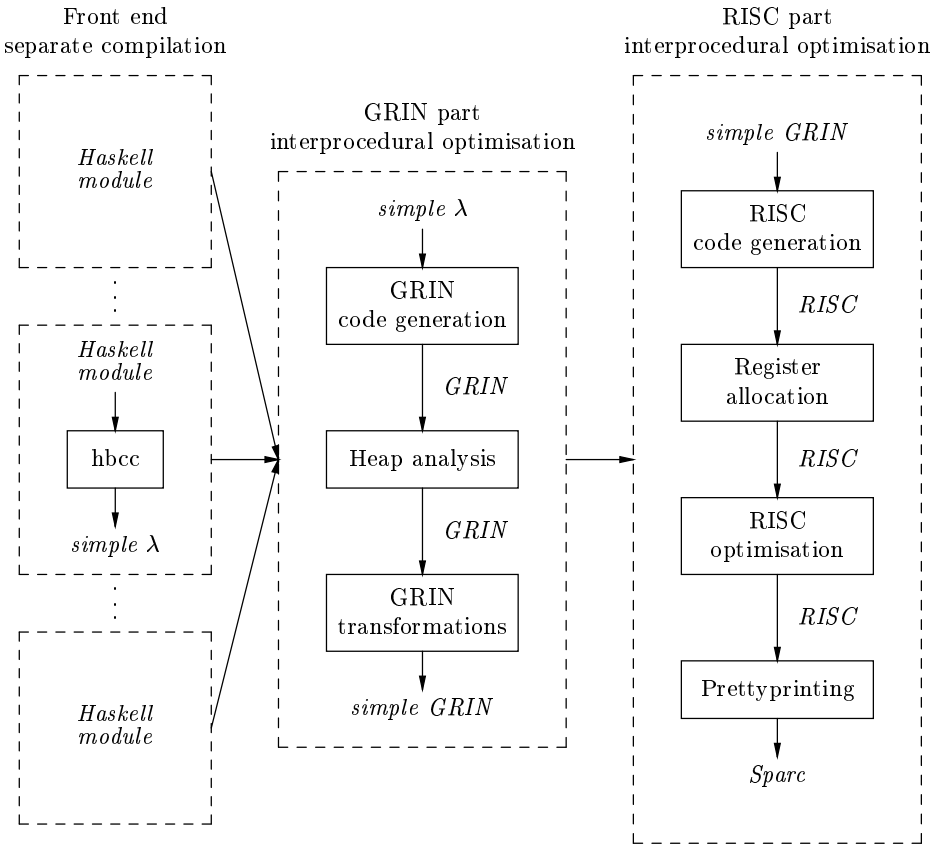

Fig. 5. Overview of the compiler

Sooner or later in the translation process, one has to be confronted with the issue of updating due to call-by-need. We have chosen to make updating explicit in the GRIN language. We currently fancy writing the GRIN programs as *state monadic* [Wad92], *first order*, *strict*, *functional programs*[2]. A simplified GRIN syntax is given in figure 6.

The `unit` operation corresponds to the unit in the monad, and `;` is the bind operator. `store`, `fetch` and `update` are operations particular to this monad. Compilers often represent the code with "three address statements", and so do we: assume the existence of primitive operators for basic values, like `int_add`, `int_gr`, etc. By convention, the names of basic valued variables end with `'`.

First a note on terminology: A *node value* (or just *node*) is a `tag` (e.g., `CInt`, `CNil`, `Fupto`) possibly followed by some arguments (pointers or basic values).

[2] Although monads normally are higher order constructs, we consider the GRIN monad as "built-in". All other operations in GRIN are first order.

prog	→ { *binding* }+	
binding	→ *var* { *var* } = *exp*	definition
exp	→ *sexp* ; λ*val* -> *exp*	sequencing
	| `case` *var* `of` alt_1 || ... || alt_n	case
	| `if` *var* `then` *exp* `else` *exp*	conditional
	| *sexp*	operation
alt	→ *val* -> *exp*	
sexp	→ *var* { *val* }+	application, function call
	| `unit` *val*	return value
	| `store` *val*	allocate new heap node
	| `fetch` *var*	load heap node
	| `update` *var val*	overwrite heap node
	| (*exp*)	
val	→ (*tag* { *val* })	complete node
	| *literal*	constant
	| *var*	variable
	| ()	empty

Fig. 6. GRIN syntax (simplified)

As the name suggests, node values quite often reside in the heap — however, node values may also be the value of local variables, and returned as values by procedures, and so on (and may eventually be allocated to one or several registers).

The GRIN language itself does not make any *a priori* interpretation of the different node values, it is the GRIN program which interpret them as representing either ordinary constructor values of the source language (the tags `CInt`, `CNil`, `CCons`) or unevaluated expressions (`Fupto`, `Fsum`). As a naming convention, we use node tag names beginning with `C` and `F` to denote ordinary data constructors and unevaluated function applications, respectively.

Although the hbcc front end does a fair amount of strictness analysis, unboxing, etc, for the sake of the exposition here, we assume a rather unsophisticated translation, essentially in the same style as in the G-machine (ie, no fancy tagging or unboxing, nor is any strictness analysis assumed).

In our basic translation scheme, 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. Procedures return node values as a result (*not* a pointer to one in the heap!).

An essential feature of our approach is that `eval`, which is normally hidden in the runtime system (or e.g. done as a 'tagless' pointer dispatch) can be written as an ordinary GRIN procedure — and thus also is susceptible to transformation! Figure 8 shows the accompanying `eval` procedure for our example.

```
main =     store (CInt 1)      ; \t1 →
           store (CInt 10)     ; \t2 →
           store (Fupto t1 t2) ; \t3 →
           store (Fsum t3)     ; \t4 →
           eval t4             ; \(CInt r') →
           int_print r'

upto m n = eval m       ; \(CInt m') →
           eval n       ; \(CInt n') →
           int_gr m' n' ; \b'→
           if b' then
               unit CNil
            else
               int_add m' 1       ; \m1'→
               store (CInt m1')   ; \m1 →
               store (Fupto m1 n) ; \p →
               unit (CCons m p)

sum l =    eval l ; \l2 →
           case l2 of
               CNil ->        unit (CInt 0)
               CCons x xs ->  eval x        ; \(CInt x') →
                              sum xs        ; \(CInt s') →
                              int_add x' s' ; \ax'→
                              unit (CInt ax')
```

Fig. 7. GRIN code for the program in figure 1.

```
eval l = fetch l ; \l2 →
         case l2 of
             CInt x'    -> unit l2
             CNil       -> unit l2
             CCons x xs -> unit l2
             Fupto m n  -> upto m n   ; \v →
                           update l v ; \() →
                           unit v
             Fsum  l    -> sum l      ; \v →
                           update l v ; \() →
                           unit v
```

Fig. 8. GRIN code for the accompanying `eval` procedure.

The standard `eval` takes a pointer to a node, and in the case of an unevaluated function application, makes sure the node gets evaluated and updated; `eval` finally returns the value thus pointed at. This means that `eval` must `fetch` the node pointed at, and perform case scrutinisation. This case must enumerate all possible nodes that `eval` might ever encounter (which without flow analysis is easiest done by enumerating all nodes ever named). `eval` either returns

the C-node so encountered, or calls the appropriate procedure to evaluate an application, and updates the F-node with the value returned.

5 The heap points-to analysis result

The transformation of the GRIN code, especially inlining of `eval`, is greatly aided by a program analysis, which gives a safe approximation to what possible nodes pointers might point to, at all points in the GRIN program. In this section we describe more precisely *what* this analysis returns, for our running example. Later we we describe *how* this analysis is implemented (section 8).

The main aim of our analysis is to determine, for each call of `eval`, a safe approximation to what different nodes, `eval` might get when it fetches a node via its argument pointer. But in effect, the heap analysis determines the abstract values of all variables in the GRIN program, as well as an abstract description of the heap.

In the concrete semantics, the value of a variable is either a basic value, a pointer into the heap, or an entire node value (as returned by a procedure or

---

`t1` $\rightarrow \{ 1 \}$	`m'` $\rightarrow \{ BAS \}$	`l2` $\rightarrow \{$ `CNil`$[\,], $`CCons`$[\{1,5\}, \{6\}] \}$
`t2` $\rightarrow \{ 2 \}$	`n'` $\rightarrow \{ BAS \}$	`x` $\rightarrow \{ 1,5 \}$
`t3` $\rightarrow \{ 3 \}$	`b'` $\rightarrow \{ BAS \}$	`xs` $\rightarrow \{ 6 \}$
`t4` $\rightarrow \{ 4 \}$	`m1'` $\rightarrow \{ BAS \}$	`x'` $\rightarrow \{ BAS \}$
`r'` $\rightarrow \{ BAS \}$	`m1` $\rightarrow \{ 5 \}$	`s'` $\rightarrow \{ BAS \}$
`m` $\rightarrow \{ 1,5 \}$	`p` $\rightarrow \{ 6 \}$	`ax'` $\rightarrow \{ BAS \}$
`n` $\rightarrow \{ 2 \}$	`l` $\rightarrow \{ 3,6 \}$	

Fig. 9. Abstract environment of the analysis result.

$1 \rightarrow \{$ `CInt`$[\{BAS\}] \}$
$2 \rightarrow \{$ `CInt`$[\{BAS\}] \}$
$3 \rightarrow \{$ `Fupto`$[\{1\}, \{2\}],$ `CNil`$[\,],$ `CCons`$[\{1,5\}, \{6\}] \}$
$4 \rightarrow \{$ `Fsum`$[\{3\}],$ `CInt`$[\{BAS\}] \}$
$5 \rightarrow \{$ `CInt`$[\{BAS\}] \}$
$6 \rightarrow \{$ `Fupto`$[\{5\}, \{2\}],$ `CNil`$[\,],$ `CCons`$[\{1,5\}, \{6\}] \}$

Fig. 10. Abstract heap of the analysis result (*without* sharing analysis).

$1 \rightarrow \{$ `CInt`$[\{BAS\}] \}$	shared
$2 \rightarrow \{$ `CInt`$[\{BAS\}] \}$	shared
$3 \rightarrow \{$ `Fupto`$[\{1\}, \{2\}] \}$	unique
$4 \rightarrow \{$ `Fsum`$[\{3\}] \}$	unique
$5 \rightarrow \{$ `CInt`$[\{BAS\}] \}$	shared
$6 \rightarrow \{$ `Fupto`$[\{5\}, \{2\}] \}$	unique

Fig. 11. Abstract heap of the analysis result (*with* sharing analysis).

`eval`). In the abstract semantics, all basic values are abstracted to a single one, *BAS*.

For abstract locations we use a bounded domain of locations $\{1, 2, \ldots maxloc\}$ where *maxloc* is the total number of `store` statements in the GRIN program. Each occurrence of a `store` statement generates the same abstract location every time it is executed. It is as if each `store` statement had its own little heap (a feasible implementation), and the abstract pointer is simply the identity of the heap, thus abstracting away from the relative position in this 'little' heap. The abstract values of pointer valued variables, and arguments of nodes, are sets of abstract locations. See also section 8 and figure 16.

Figure 9 shows the abstract environment derived for our running example (the GRIN program in figure 7). The `store` statements of the program have been numbered $1 \ldots 6$, thus the abstract values of variables `t1`, `t2`, `t3`, `t4`, `m1`, and `p` are $\{ 1 \}$, $\{ 2 \}$, ..., $\{ 6 \}$.

The analysis also returns an *abstract heap*, which maps abstract locations to abstract node values. Figure 10 show a possible abstract heap derived for our running example (this will be refined shortly).

Consider the `eval` of `m` in the `upto` procedure. The abstract value derived for `m` is $\{ 1, 5 \}$. Both 1 and 5 in the abstract heap are `CInt` nodes; thus this `eval` is trivial, the value is already evaluated.

Now consider the `eval` of `l` in the `sum` procedure. The abstract value derived for `l` is $\{ 3, 6 \}$, and according to figure 10 both these locations might be either `Fupto`, `CNil`, or `CCons` nodes.

It has turned out to be quite easy to incorporate a sharing analysis into the points-to analysis. Not only does this provide useful information for update avoidance, it also serves to improve the precision of the points-to analysis as such!

Thus, in actual practice our analysis also returns a third component, a sharing table, which maps abstract locations to its sharing properties: True if the abstract location is shared, i.e., if a concrete instance of the abstract location may be subject to a `fetch` more than once, False otherwise. In our example, and in general at the stage of compilation where points-to analysis is currently applied, it is only `eval` that performs `fetch` operations; however, we might also want to analyse the program later in the process where `fetch` operations can occur explicitly.

Figure 11 shows the abstract heap the analysis derived for our running example, together with its sharing information. Thus we can see that abstract locations 3 and 6 now only contain `Fupto` nodes, and that these locations are unique (non-shared). The explanation for this is that both these nodes are born as `Fupto` nodes, and even though `eval` might update such a concrete location with either a `CNil` or a `CCons`, this will never be visible. A location may only become shared if it is a possible value of a nonlinear variable (i.e., used more than once). The nonlinear variables in our example are: `m`, `n`, and `m'`.

The abstract environment of our running example does not become modified by the use of a sharing analysis part; but in general it might well be.

6 GRIN transformations

6.1 EVAL inlining

After the heap points-to analysis, the next step in the compilation process is to inline all calls of `eval`. In general, one might replace a call of `eval` by its entire body (see figure 8). This would, however, in most cases be blatantly wasteful, since at each `eval` point only a (small) subset of the nodes can be present — the result of the points-to analysis gives a safe subset. Further, one may also omit the accompanying `update` operations if according to the sharing information the corresponding abstract locations are unshared.

Let us discuss the most general cases first, from the point of view of our running example. Consider the `eval l` in the `sum` procedure, and let us assume for a moment that we do not have any sharing information, and that according to the analysis the possible nodes encountered here are `Fupto`, `CNil` and `CCons`. The code:

```
eval l ; \l2 → rest
```

could then be expanded into (by replacing the call by the body of `eval`, substituting actual arguments for formal ones, and deleting impossible cases):

```
(fetch l ; \l2 →
 case l2 of
     CNil       -> unit l2
     CCons x xs -> unit l2
     Fupto m n  -> upto m n   ; \v →
                   update l v ; \() →
                   unit v
) ; \l2 → rest
```

However, as can be seen from figures 9 and 11, the information actually derived by the analysis for this case is that `l` points to a `Fupto` node, and it is unique (unshared). This information can be used for two things:

- since there is only one node, `Fupto`, no case analysis needs to be done,
- since the pointer in known to be unshared, no updating needs to be done.

So, the "`eval ; \l2 →` *rest*" can actually be inlined into the much better:

```
( fetch l ; \(Fupto m n) →
  upto m n
) ; \l2 → rest
```

The second `eval` appearing in `sum`, the `eval x`, is much simpler in character. Since all the nodes that `x` might point to (in fact there is only one, `CInt`) are ordinary value nodes, `eval` does not have to call any evaluation procedure to get the actual value, instead a simple `fetch` will do. Figure 12 shows the complete `sum` procedure with both `eval`s inlined accordingly.

The inlining of `eval` does not actually happen in one ad-hoc step as indicated here. Rather, the resulting inlining of `eval` shown above is the result of a large number of small transformations — see next section.

```
sum l = (fetch l ; \(Fupto m n) →
         upto m n
        ) ; \l2 →
        case l2 of
            CNil ->        unit (CInt 0)
            CCons x xs ->  fetch x       ; \(CInt x') →
                           sum xs        ; \(CInt s') →
                           int_add x' s' ; \ax'→
                           unit (CInt ax')
```

Fig. 12. The procedure `sum` with its `eval`s inlined.

6.2 The GRIN transformation machinery

Although the `eval` inlining above is a very important transformation, it is, in fact, only a small part of a large number of GRIN program transformations. The main idea behind the GRIN transformation machinery is to use many, each very simple, GRIN source-to-source transformations. Each transformation is of course correctness-preserving and hopefully performance-improving. Even though each single transformation will make only a very small change, they will, taken together, produce greatly simplified and optimised GRIN code.

Many of the transformations are very "local" in the sense that they will try to find a small sub-expression, of a larger GRIN expression, that matches a certain pattern, and if found, transform it to a slightly different sub-expression. Other transformations are a bit more involved. Remember also that we assume that the front end has already transformed the program "as much as possible' on the functional level, and hbcc does indeed implement most "standard" functional transformations.

6.3 Example transformations

Rather than going through all transformations, we will concentrate on a few examples. Some transformations are rather general in nature, some are more specialised. We will show a few of each kind.

Monad unit laws – copy propagation. Since we use a monad to structure GRIN we can directly use the *monad laws* [Wad92], that all monads must satisfy, as transformation rules. The *left unit* monad law is usually written as (we use `;` as the bind operator, as in GRIN):

`(unit x) ; m` $\equiv$ `m x`

We can get a more useful transformation (denoted $\Rightarrow$) by instantiating `m` as a binding:

```
(unit x) ; m
  ⇒   { instantiate m }
(unit x) ; (\v -> k)
  ⇒   { use unit law }
(\v -> k) x
  ⇒   { β reduction }
k[x/v]
```

I.e., we can always delete a `unit` to the left of a binding and simply do a substitution instead, i.e., we have eliminated a "copy". Note that `x` and `v` above must not necessarily be variables, they could equally well be complete nodes:

```
unit (CInt a') ; \(CInt b') -> k   ⇒   k[a'/b']
```

There is also a corresponding *right unit* monad law, which will give rise to the following transformation:

```
m ; \v -> unit v   ⇒   m
```

Bind associativity. In any monad the bind operator must be associative [Wad92], and this turns out to be very useful for transformation purposes. During the transformation process it is a good idea to keep the GRIN code normalised, i.e., keep the GRIN syntax tree "right skewed" with only bind operations as its spine. Unfortunately this property can be destroyed by any transformation that introduces new code, for example inlining a call to `eval`. Say, for example, that we have the code "`g ; h`" and we want to replace `g` by the sequence, "`m ; k`". Then, we would like to restructure the code as shown in figure 13, to keep the right skewed property.

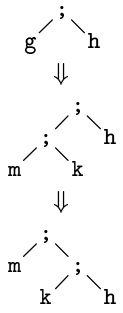

Fig. 13. Normalisation

But this is exactly what the associativity monad law tells us! Shown in GRIN terms, the general transformation is:

```
(m ; \a -> k a) ; \b -> h b   ⇒   m ; (\a -> k a ; \b -> h b)
```

Unboxed values. Our current front end, hbcc, uses the *unboxing* methods described in [PJL91]. It is often very good at transforming strict function arguments and function results to unboxed representations. However, it has some shortcomings. The method in [PJL91] cannot unbox a function that returns a value of a datatype whose (single) constructor takes more than one argument, like for example a pair. There is simply no way to express that in functional code, a function must always return exactly one value. Unboxing function arguments of such types (single constructor, more than one argument) is mentioned in [PJL91], but unfortunately not implemented in hbcc. As an example, a strict pair argument that is unboxed can be replaced by two arguments, one for each component of the pair, a transformation sometimes called *arity raising*. A final shortcoming of the unboxing done by hbcc is that it is only attempted in rather restricted contexts (sufficiently strict etc).

In GRIN there is no problem handling any of the cases above. As an example, we can express that a function returns an unboxed pair, i.e. simply returns the two components of the pair (in the final code, this will be done using two registers). To show our transformation, we will give an example of how a function that returns a boxed integer can be changed to return an unboxed integer. In GRIN, a function that *returns* an integer will do it using the `unit` operation. This means that the actual tag is visible, so we can simply remove it:

```
unit (CInt x')  ⇒  unit x'
```

Of course, we will now also have to change all calls to the function. Assuming the function we have just unboxed was called `f`, we will transform all calls to `f` (in any context):

```
f as  ⇒  f as ; \y' -> unit (CInt y')
```

Many of the "extra" units and lambdas that might get inserted are trivially eliminated using the monad law transformations described above. As an example, assume that the above call to `f` appeared right before a lambda pattern:

```
f as ; \(CInt a') -> m
  ⇒  { unbox f }
(f as ; \y' -> unit (CInt y')) ; \(CInt a') -> m
  ⇒  { associativity + left unit laws }
f as ; \y' -> m[y'/a']
```

The effect of this will be to eliminate all "tagging costs" associated with calls to `f`. Some special care has to be taken with tail calls, but they can be handled as well. Unboxing procedure arguments, and types where the node have several arguments can be done in a completely analogous way.

Simplifying nodes. Some of the GRIN transformations have more the nature of *simplifications* rather than *optimisations*. An example of the former is a transformation we call *vectorisation*. The aim of vectorisation is to make GRIN variables that contain node values (i.e. a tag possibly with arguments), more concrete by transforming to multiple variables, each containing a simple value (basic value or pointer). Consider the following:

```
foo l = fetch l ; \l2 ->
        case l2 of
            CNil       -> nil_body
            CCons x xs -> cons_body
```

This could be a function with a single list argument, where the points-to analysis has shown that the argument will always be evaluated (so all that remains of the `eval` is a `fetch`). If we look at the `l2` variable above, it will contain a complete node, either a `CNil` tag or a `CCons` tag and two arguments. We will now replace `l2` with three simple variables (this is what we call a *vector*):

```
foo l = fetch l ; \(t' a as) ->
        case (t' a as) of
            CNil       -> nil_body
```

```
        CCons x xs -> cons_body
```

In fact, we consider the actual tags to be basic values. Hence the variable `t'`, it will bind the tag itself. Note that if `t'` is `CNil`, then `a` and `as` are undefined. This will hopefully be a bit more clear after the right hoisting transformation below. Before that, though, we will simplify the vectors (variable nodes) that we just introduced. The case expression in our example depends only on the tag, so let us make that explicit:

```
foo l = fetch l ; \(t' a as) ->
        case t' of
            CNil  -> nil_body
            CCons -> cons_body[a/x,as/xs]
```

After this transformation, all case expressions will be only a "case test", they will not bind any variables.

Right motion hoisting. The `fetch` operation above will load a complete node from memory, and bind its various components to the three variables.[3] To further simplify the GRIN code, we now split the `fetch` into its three components:

```
foo l = fetch l[0] ; \t' ->
        fetch l[1] ; \a  ->
        fetch l[2] ; \as ->
        case t' of
            CNil  -> nil_body
            CCons -> cons_body[4]
```

By the notation "`l[`n`]`" we mean the n:th component of the node pointed to by `l`. We now see that since `a` and `as` are not used in the `CNil` branch of the case expression, their corresponding `fetch` operations can be moved (or hoisted) into the `CCons` branch:

```
foo l = fetch l[0] ; \t' ->
        case t' of
            CNil  -> nil_body
            CCons -> fetch l[1] ; \a  ->
                     fetch l[2] ; \as ->
                     cons_body
```

We call this transformation *right motion hoisting*. It is interesting to compare this to the *let-floating* described in [PJPS96]. One of the let-floating variants, where a let-binding is floated into a case branch, does look quite like our transformation (one difference is of course that a let will allocate storage in the heap, whereas our `fetch` will only read from the heap). However, the code above is an example

[3] Note that this does not imply anything about the way a node actually gets stored in the heap! It only says that there should be some way to extract the tag, etc.

[4] We omit the substitution henceforth.

of a situation where we benefit from the extra "low level control" of the GRIN code. The transformation example above is not possible on the "functional level", because there is no way to distinguish between the different components of the value (the node variable `l` above).

A good thing about this transformation is that it can decrease memory *bandwidth*, by not fetching unnecessary values. However, the transformation might also have a negative impact on execution time. If the `CCons` branch is the most common one, and if the values `a` and `as` are needed early in *cons_body*, it might, in fact, be better to *prefetch* them before the case test (for reasons of memory *latency*). On the other hand; if the tag is already loaded, the rest of the node is probably already in the cache, so subsequent loads will be cheap. More experiments are needed to determine if this optimisation really is beneficial.

More transformations. The effect of all GRIN transformations, of which we have only shown a few, is to gradually turn the GRIN code into a very simple form, with all operations made explicit. This will make the actual code generation (to real machine code) quite simple (see section 9).

7 Higher order functions

True to the GRIN philosophy, also function objects are represented by node values. Just like the G-machine and most other combinator-based abstract machines, function objects in GRIN programs exist in the form of curried applications of functions with too few arguments. Consider again the function `upto` of our running example, which takes two arguments. We represent the function object of `upto` by a node `Pupto_2`, and an application of `upto` to one argument by a node `Pupto_1` e. The naming convention we use is that the prefix `P` indicates a partial application, and `_2` etc. is the number of *missing* arguments.

In analogy with the generic `eval` procedure, programs which use higher order functions must also have a generic `apply` procedure, which must cover possible function nodes that might appear in the program. An example is shown in figure 14. `apply` returns the value of a function value (node) applied to one additional argument. Generally, `apply` just returns the next version of the function node with one more argument present, except when the final argument is supplied: then the call of the procedure takes place.

GRIN does not provide a way to do a function application of a variable in a lazy context directly, e.g., build a representation of `f x` where `f` is a variable, instead a closure must be wrapped around it; this is the purpose of the `ap2` procedure.

In the further compiling of programs which uses higher order functions, also `apply` calls are inlined, in much the same way as `eval` calls.

If the application in the original program has more than one argument, several `apply` statements in sequence must be used. This arrangement is conceptually simple (a great advantage when it comes to the points-to analysis). Although it implies the unnecessary construction of intermediate function values,

```
apply f x = case f of
                Pupto_2    -> unit (Pupto_1 x)
                Pupto_1 y  -> upto y x
                           ⋮
```

Fig. 14. The `apply` procedure.

```
twice f x = store (Fap2 f x) ; \t1 →
            eval f ; \f2 →
            apply f2 t1

ap2 f x   = eval f ; \t2 →
            apply t2 x
```

Fig. 15. The GRIN code for the function `twice f x = f (f x)`.

a sequence of inlined `apply`s can easily be simplified to avoid these intermediate function values.

8 The innards of the points-to analysis

When we designed our points-to analysis, an overriding design goal was that it had to be very fast, in order to be able to analyse large entire programs at once — if it had to be done at the cost of some precision, then so be it! The result is an analysis which we think is no costlier than, e.g., live variable analysis. We accomplish this by the following means:

- a single abstract heap approximates the real heap at all times and at all program points: this makes the analysis flow insensitive,
- a single abstract environment approximates all local environments; this arrangement does not actually impose any extra approximation, since all local variables are uniquely named and an abstract local environment is always a subset of this 'global' abstract environment,
- the analysis is insensitive to calling context: a procedure parameter gets its abstract value by 'unioning' the corresponding actual parameters at the call sites, and the same abstract return value is returned as a result of all calls for each procedure.

As mentioned in section 5, we also include a sharing analysis into the points-to analysis machinery. Not only does this provide desired sharing information for update avoidance, it also serves to improve the precision of the points-to result!

The analysis works in two steps:

- setting up a system of data flow equations, for the variables of the abstract environment and the abstract heap,
- solving the equations.

$$
\begin{array}{rll}
\widehat{Loc} = 1, 2, \ldots & & \text{“Cons points”} \\
\widehat{Val} = \{BAS\} + \widehat{Loc} & & \text{“Small” values} \\
\widehat{V} = \mathrm{P}(\widehat{Val}) & & \\
\widehat{Node} = Con \times \widehat{V}^* & & \text{Node values} \\
\widehat{N} = \mathrm{P}(\widehat{Node}) & & \\
\widehat{Heap} = \widehat{Loc} \to \widehat{N} & & \\
\widehat{VarVal} = \widehat{V} \cup \widehat{N} & & \text{Abstract values of variables}
\end{array}
$$

Fig. 16. Abstract domains for the heap points-to analysis.

$$
\begin{array}{llll}
\texttt{t1} = \{\,1\,\} & \texttt{m'} = \{\,BAS\,\} & & \texttt{x} = \texttt{l2} \downarrow \texttt{CCons} \downarrow 1 \\
\texttt{t2} = \{\,2\,\} & \texttt{n'} = \{\,BAS\,\} & & \texttt{xs} = \texttt{l2} \downarrow \texttt{CCons} \downarrow 2 \\
\texttt{t3} = \{\,3\,\} & \texttt{b'} = \{\,BAS\,\} & & \texttt{x'} = \{\,BAS\,\} \\
\texttt{t4} = \{\,4\,\} & \texttt{m1'} = \{\,BAS\,\} & & \texttt{s'} = \{\,BAS\,\} \\
\texttt{r'} = \{\,BAS\,\} & \texttt{m1} = \{\,5\,\} & & \texttt{ax'} = \{\,BAS\,\} \\
\texttt{m} = \texttt{t1} \sqcup \texttt{m1} & \texttt{p} = \{\,6\,\} & & \texttt{ru} = \{\,\texttt{CNil}[\,], \texttt{Ccons}[\,\texttt{m}, \texttt{p}\,]\,\} \\
\texttt{n} = \texttt{t2} \sqcup \texttt{n} & \texttt{l} = \texttt{t3} \sqcup \texttt{xs} & & \texttt{rs} = \{\,\texttt{CInt}[\{BAS\}]\,\} \\
 & \texttt{l2} = \mathit{EVAL}(\mathit{FETCH}\,\mathit{heap}\,\texttt{l}) & &
\end{array}
$$

Fig. 17. Abstract environment equations.

$$
\begin{array}{rl}
\mathit{heap} = [\; 1 \to & \{\,\texttt{CInt}[\{BAS\}]\,\} \\
2 \to & \{\,\texttt{CInt}[\{BAS\}]\,\} \\
3 \to & \{\,\texttt{Fupto}[\,\texttt{t1}, \texttt{t2}\,]\,\} \sqcup \texttt{ru} \\
4 \to & \{\,\texttt{Fsum}[\,\texttt{t3}\,]\,\} \quad \sqcup \texttt{rs} \\
5 \to & \{\,\texttt{CInt}[\{BAS\}]\,\} \\
6 \to & \{\,\texttt{Fupto}[\,\texttt{m1}, \texttt{n}\,]\,\} \sqcup \texttt{ru} \;\;]
\end{array}
$$

Fig. 18. Abstract heap equations.

$$
\begin{array}{l}
\mathit{EVAL}\,S \;=\; \{\mathit{tag}\,L \mid \mathit{tag}\,L \in S \wedge \mathit{is\text{-}value\text{-}node}\,\mathit{tag}\} \\
\mathit{FETCH}\,\mathit{heap}\,\{l_1, \ldots, l_n\} \;=\; \mathit{heap} \downarrow l_1 \sqcup \ldots \sqcup \mathit{heap} \downarrow l_n \\
\{\ldots, \mathit{tag}\,[v_1, \ldots, v_i, \ldots], \ldots\} \downarrow \mathit{tag} \downarrow i \;=\; v_i \\
\{\ldots \mathit{tag}[x_1, \ldots, x_i, \ldots], \ldots\} \sqcup \{\ldots \mathit{tag}[y_1, \ldots, y_i, \ldots], \ldots\} = \\
\qquad\qquad \{\ldots \mathit{tag}[x_1 \cup y_1, \ldots, x_i \cup y_i, \ldots], \ldots\}
\end{array}
$$

Fig. 19. Utility functions.

We first describe the basic machinery without the sharing analysis part, and then discuss the modifications needed to implement the sharing analysis, and to deal with higher order functions.

8.1 The basic analysis machinery

Deriving the equations. From the GRIN program, we set up a system of data flow equations which describes the values of the variables in the abstract environment, and the locations of the abstract heap. Figures 17 and 18 shows these equations for our running GRIN program example (figure 7).

The abstract domains are summarised in figure 16. We now proceed to explain each of the different forms of equations.

The abstract environment contains the variables of the GRIN program, plus one variable for each procedure, which denotes the return value of such a call: `ru` for `upto`, and `rs` for `sum`.

To begin with, quite a lot of the variables can immediately be seen to have the value $\{BAS\}$.

As mentioned, we use a single fixed abstract location for each `store` statement: hence the equations for `t1`, `t2`, `t3`, `t4`, `m1`, and `p` in the environment part. The *heap* variable has at each location the value of the corresponding `store`, possibly unioned with one more item. If an abstract location has the value of a node which represent an unevaluated function application, in our case `Fupto` or `Fsum`, we simply set these locations also to have the value of the corresponding return values, since such nodes will most likely be updated with these eventually: hence the union with `ru` and `rs` respectively.

Parameter variables, like `m`, `n`, and `l`, has the value of the union of all the actual arguments of the applications of the program, both direct calls, e.g., `sum xs`, or 'lazy calls' .e.g., `store (Fupto m1 n)`. So for instance, `m` is the first argument of `upto`, and hence gets the value `t1`$\sqcup$`m1` from the two different `store`s of `Fupto`s.

The variable `l2` holds the value of an `eval`: this is simply obtained by taking the union of the value of the abstract locations which `l` might point to (done by *FETCH*), and then extracting those nodes that represent value constructors (done by *EVAL*).

Values for variables which are bound in a `case`, like `x` and `xs`, get their abstract value by extracting out the corresponding component value for the variable being `case`d upon.

In taking the union ($\sqcup$) of sets of node values, these are unioned tag-by-tag so that in the resulting set there is only one element for each node tag (see figure 19). Unions of sets of abstract pointers are like ordinary unions.

Solving the equations. Having obtained the data flow equations for the points-to analysis, the natural way of solving these equations is by fixpoint iteration, starting with an empty abstract environment and an empty heap, and apply the equations until a fixpoint is reached.

In our implementation, convergence is speeded up by using a 'depth first ordering' [ASU86, sec. 10.9] of the variable equations. Using this for our example, it takes only 4 iterations to converge to a fixpoint.

8.2 Adding sharing analysis

As mentioned already in section 5, a sharing analysis can easily be incorporated into the points-to analysis by adding a third component, suitably called a *sharing heap*, a mapping from abstract locations to a boolean value: True if an instance of this abstract location might be shared, False if it cannot be.

The initial value of the sharing heap is all Falses. The value of the sharing heap is awkward to express in equational form, instead we do as follows: during an iteration, an abstraction location is set to shared, i.e. True, either if it might be pointed to by a nonlinear variable, or from another shared location.

In the abstract heap the return value part, e.g., `ru` of location 3, is only $\sqcup$'ed if the same abstract location becomes shared.

The semantic function *EVAL* also needs to be modified: it cannot just take the subset of the node values which represent proper values, but needs to extract 'by itself' from elsewhere what the values of an `Fupto` application, etc, might be. We omit those details here.

8.3 Higher order functions

Higher order functions cause no fundamental difficulties to add into our analysis — only practical ones!

The abstract value of an `apply f a` depends, obviously, on the abstract value of `a`: if the value of `f` contains e.g., a `Pupto_2` node, then the `apply` contains `Pupto_1` a where a is the abstract value of `a`; if the value of `f` contains e.g., a `Pupto_1` b node, then the value of the `apply` contains whatever `upto` returns since the application becomes saturated at that point (c.f. figure 14).

Previously, a parameter of a procedure gets its abstract value from the union ($\sqcup$) of some other variables – for example `l` = `t3` $\sqcup$ `xs`. Unfortunately, due to the `apply` calls, the equations for the variables are no longer this static. Now, if a call of some procedure occurs as a result of an `apply` call, then the actual arguments of those call need to be $\sqcup$'ed with the abstract values of the corresponding parameter variables.

We have solved that problem practically as follows. A single variable, let us call it `aa` here, is used to collect all the possible 'extra' arguments as a result of the `apply`s in the program. It is convenient to represent these extra arguments by closure nodes for those applications, e.g., `Fupto` x_1 x_2. If there is an `apply a b` in the GRIN program, then there is also an *APPLICATONS* a b being $\sqcup$'ed to the rhs of `aa`. If a call of, e.g., `upto` could occur as a result of that `apply` (in which case the value of `a` need to contain a `Pupto_1` c node), then the value of *APPLICATIONS* a b should contain a `Fupto c b` node. Finally, the variable for the parameter need to extract the relevant part of `aa`: the first argument of `upto`, for example, need to add `aa` $\downarrow$ `Fupto` $\downarrow 1$ to the right hand side of its equation.

9 RISC code generation

After all GRIN transformations, the resulting code is translated into machine code for a hypothetical RISC machine (load-store architecture) assuming an infinite number of available *virtual* registers. These virtual registers will later be mapped onto real machine registers by the register allocator. The translation to RISC code is rather straightforward, since the final GRIN code is in a very simple form.

Each procedure is represented as a flow graph of *basic blocks*. The (intraprocedural) flow graphs are at this stage always DAGs, since GRIN can not represent (intraprocedural) loops. Later in the compilation, tail recursion optimisations will indeed, turn tail calls (to the same function) into "real loops" in the flow graph. On the interprocedural level, i.e., between procedures, the flow graphs are linked together using *call* and *return* edges.

Throughout the entire back end we do a lot of book keeping and analysis aimed at determining enough information about what registers contain *root pointers*, i.e., need to be followed during GC. For space reasons we will have to postpone a description of how this is done to a future paper.

10 Interprocedural register allocation

We believe good register allocation to be a vital optimisation. For reasons explained in section 2, we also believe it to be important to do *interprocedural* register allocation for lazy functional languages. Or, put in another way, what we need, to implement these languages efficiently, is to minimise the procedure *call and return overhead*, and doing interprocedural register allocation can be a good method for achieving that [Cho88].

Our register allocation algorithm was described in [Boq95a,Boq95b], and, for space reasons we will not explain it in detail here (although it has changed a bit since then). In summary, it is an interprocedural *graph colouring* algorithm, based on *optimistic* graph colouring [BCKT89], but with several additions; e.g. interprocedural *coalescing* and a restricted form of *live range splitting*.

Cheap procedure calls. The register allocator helps reducing the procedure call penalty in the following ways:

- It is very successful in passing procedure *arguments* in registers, using tailor-made argument registers for each procedure.
- It often achieves good *targeting*, i.e., a value that later will be used as argument in a call, will actually be *calculated* in the correct register. In most cases, no extra "register shuffling" will be necessary at the call site.
- Likewise with procedure *return values* (we will often use more than one register to return a result).
- Local variables that are *live*[5] across a call site, can often be kept in a register during the call. This should be seen in contrast to a global register allocator, which normally will have to save and then restore certain registers around each call site, if they risk being clobbered by the callee.

Graph colouring. The main task of the register allocation algorithm is to build an *interference graph* (conflict graph) for the complete program, and then *colour* it. We initially assume that all variables are allocated to *virtual* registers.

[5] A variable is said to be *live* at a certain point if its value may be used on some execution path leading from that point.

If the allocator fails to find a colour for some variable it will be *spilled*, i.e., the value kept in memory instead, or *splitted*, i.e. kept in different registers during different periods. The "variables", that participate in the colouring are: procedure arguments and return values, local variables and all kinds of temporaries introduced by the compiler.

11 RISC optimisations

The RISC optimiser implements a number of different low level optimisations. Naturally, we implement the "standard" optimisations for lazy functional languages, like heap pointer and tail call optimisations. We also optimise stack usage (the stack pointer, frame building and the return address) using a *shrink-wrap* technique similar to the one used in [Cho88] to optimise the use of callee-saves registers. For example, the return address register,[6] can be seen as a callee-saves register, i.e. we need not save it until we are certain to do a new procedure call. In a similar way, we can avoid creating a stack frame,[7] until it is absolutely needed. The different stack optimisations, together with tail call optimisations, often succeed very well in creating small tight loops for tail recursive functions.

Other optimisations are of a more general kind, like instruction scheduling and branch optimisations. The Sparc processor is a *delayed-load* architecture with call and branch *delay slots*, which means that it is important to separate loads instructions and uses of the loaded result, and to fill delay slots with useful instructions. We use a rather standard instruction scheduler to accomplish this, in the style of [GM86].

Currently, most our RISC optimisations are placed *after* the register allocation (see figure 5), which might seem a bit odd in comparison with conventional compilers where normally many optimisations are done *before* the register allocation. However, in our case it turns out that many of the standard optimisations are subsumed by transformations already done at the GRIN level.

12 Measurements

We have compared our back end to the Chalmers and Glasgow Haskell compilers, hbc and ghc, respectively. Our implementation is still rather experimental, and unfortunately we have not been able to compile any large programs. Therefore, the measurements shown here should be taken for what they are, only toy programs experiments. On the other hand, one positive thing about having small test programs is that it is possible to examine the code produced in various stages of the compilation, to get "fair" tests between compiler "sub-parts". The measurements in figure 12 show four example programs: `nfib 32`, `sieve2` (summing all primes below 10000), `fqueens` (the queens problem of size 10, a first order program) and `hqueens` (ditto, but using higher order functions). The

[6] Assuming a RISC style jump-and-link instruction used for doing procedure calls.

[7] On the Sparc we use the standard system stack.

		hbc[a]	GRIN	hbcc+GRIN	ghc[b]
`nfib`	instructions	341.9	123.4	81.1	-
	stack	84.6	28.2	21.1	-
	time	7.6	4.6	**1.8**	2.2
`sieve2`	instructions	72.9	24.2	24.2	-
	stack	17.2	3.1	3.1	-
	time	3.2	**1.7**	1.9	3.5
`fqueens`	instructions	174.3	49.4	48.9	-
	stack	44.6	3.0	3.0	-
	time	5.1	2.0	**1.9**	4.4
`hqueens`	instructions	198.3	-	57.8	-
	stack	50.8	-	3.3	-
	time	5.0	-	**2.0**	4.1

[a] `hbc-0.9999.1 -O -msparc8`

[b] `ghc-0.29 -O2 -fvia-C -O2-for-C`

Fig. 20. Performance of some small programs.

column marked just GRIN means that handwritten GRIN code is input to our back end. The intention of this is to create a fair comparison of hbc's and our back ends. We have made sure that this code is written exactly as what is input to hbc's back end, i.e., the same strictness and boxity, and it reads and writes exactly the same nodes in the heap. In other words, the hbc and GRIN columns will perform exactly the same graph reduction.

The column marked hbcc+GRIN shows our back end together with the hbcc front end, and is supposed to be a more fair comparison against the ghc column. Ghc and hbcc should be roughly comparable as front ends.

Our main measurements are done using a tool to collect dynamic instruction counts.[8] We show the total instruction count and the total number of stack references (loads + stores), all in millions of instructions. We also include some timings.[9] However, given how hard it is to accurately measure time on a UNIX system, especially for such small programs, these should be seen mainly as a reference. Garbage collection times are not included (for any compiler).

If we look at the total instruction count, we can see that the GRIN column is roughly 3 times as fast as hbc, i.e. our back end compared to the hbc back end. Moving to the hbcc+GRIN column, we see that we get slightly better yet, around 3-4 times fewer instructions executed compared to hbc. The total instruction count also correlate rather well with the timings. Comparing hbcc+GRIN with ghc, we are roughly twice as fast as ghc for all the examples except `nfib`.

We have included stack reference counts as a measure on how well our register allocator succeeds, since good register allocation typically means that stack allocated variables (including temporaries and function parameters and results), have been allocated to registers instead. Looking at the figures, we see a dra-

[8] Unfortunately, we have not yet been able to make this tool work for ghc binaries.

[9] User times on a 40MHz SuperSparc with 1Mb external cache and 80Mb memory.

matic reduction in the number of stack references for our back end compared to hbc, ranging from 70% to 95% eliminated stack references.

13 Related work

Interprocedural optimisation. Recently, various interprocedural optimisations have gained increasing popularity, simply because they are much more powerful than their corresponding global (i.e., per procedure) optimisations. Practical difficulties with whole-program optimisation can be reduced by the use of an integrated programming and optimisation environment, like the $\mathbf{R}^n$ environment [CKT86].

For a lazy language like Haskell, compilers typically compile one *module* at a time. At first sight, this might appear as a good opportunity to optimise several procedures at once. However, it seems as if this does not apply very well to low level optimisations, like those presented in this paper, where the actual *dynamic* control flow is important, as explained in section 2. In a lazy language, a function that is *local* to a module in the source code, might very well *escape* from the module at run time (if it is built into a closure) and then be called from somewhere else (using `eval`).

GRIN and transformations. Our intermediate code, GRIN, is in its essence not very different from any other intermediate code used to implement lazy functional languages (usually called code for a particular *abstract machine*); e.g. the G-machine [Joh84], the ABC-machine [SNvGP91] and TIM [FW87]. In some sense, GRIN is on a slightly "higher level" than the machines mentioned above. On the other hand, compared to the STG language [PJ92], GRIN is more "low level" which has proven itself useful in some transformations (see section 6).

The idea of "compilation by transformation" is not new, see for example [KH89,App92]. In particular, the idea of using a large number of very small transformations is similar to what is used by the *simplifier* [PJ96] in the Glasgow Haskell Compiler. The main difference compared to our transformations is that the GRIN code is a low level code compared to both the Core and the STG language used by ghc (they are both essentially 2:nd order λ-calculus, the STG language is on a slightly lower level). A typical example of the difference is that we, in GRIN, can "inspect" a node value (closure) without having to force (evaluate) it, something which is not expressible in the STG language.

It may be an illusion, but the monadic presentation of GRIN code gives it a very "functional flavour", and hence a nice framework for doing analysis and transformations.

The relationship between GRIN and *Continuation Passing Style* (CPS) [AJ89], can be compared to programming using either monads or continuations, i.e., it is probably mainly a matter of taste. One might also compare GRIN to *Static Single Assignment* (SSA) code [CFR+91], which has received recent popularity for implementing imperative languages (mainly Fortran). In GRIN, we will get single assignment "for free", i.e. all variables in a GRIN program are only assigned once, yet another example of the "functional flavour" discussed before.

Heap points-to analysis. A great deal of work has been done on points-to analysis of more conventional languages – see, for example, the overview in [SH97]. However, with the notable exception below, none other seem to have addressed the problem in the context of lazy graph reduction.

The work by Karl-Filip Faxén [Fax95,Fax96] is quite similar in scope to ours, and he addresses many of the same problems as we do. Central to his work is a type based program analysis, called *flow inference*, which analyses programs expressed in his intermediate language, *Fleet* (Functional Language with Explicit Evals and Thunks). As the name suggests, he analyses programs on a higher level than our GRIN code. His flow inference derives information quite similar to ours, and he uses the information to eliminate `eval`s and thunks, to do unboxing, and update elimination.

Register allocation. To our knowledge, interprocedural register allocation has not been applied previously to code generated from a lazy functional language. It has been applied to other kinds of languages though, e.g. to Lisp [SH89] and to C [Cho88,Wal86].

14 Conclusions and further work

Our preliminary results look very promising, but there is a lot of implementation work that needs to be done before we can say if our back end really can be made practical. We can not yet say how our interprocedural approach will scale up to large programs. Two possible problem areas are the heap points-to analysis and the interprocedural register allocation. Although there are various methods for trading exactness for speed in both these cases, it is difficult to predict exactly how the code quality will be affected by less precise program information.

The GRIN back end described in this paper constitutes quite a heavy basic machinery. But once that has come off the ground, many other opportunities for further optimisations present themselves:

So far, the only use of inlining in our GRIN transformations are to unfold calls to `eval` and `apply`, but we plan to experiment with more aggressive methods. Inlining of conventional calls, together with simplification of the resulting GRIN code, might effectively give a compile-time version of the vectored return mechanism of the STG machine [Joh91].

As an area for further work, we would like to investigate which of the transformations usually done closer to the front end, e.g., ghc's (or hbcc's) "functional" transformations, that could be profitably done on the GRIN level: for example unboxing, deforestation, firstification, possibly even strictness analysis!

The simplifier in ghc is a kind of "transformation engine" that will apply (and repeat) transformations rather automatically. We plan to implement a similar machinery in our back end.

Doing aggressive optimisations like the ones described here might very well turn out to be impractical to do on large entire programs. We might consider a profiling based approach, where the optimisation effort is spent where it really matters for the overall execution speed of the program.

Acknowledgements. The second author, Johnsson, visited the Glasgow Functional Programming Group 1989-90, on an SERC fellowship: the first ideas [Joh91] concerning the GRIN approach occured in that inspiring environment.

References

[AJ89] A.W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[App92] A. W. Appel. *Compiling With Continuations*. Cambridge University Press, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Addison-Wesley Publishing Company, Reading, Mass., 1986.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 275–284, Portland, OR, June 1989.

[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. URL: `http://www.cs.chalmers.se/~boquist/fpca95.ps`.

[Boq95b] Urban Boquist. Interprocedural Register Allocation for Lazy Functional Languages. Licentiate Thesis, Chalmers University of Technology, Mars 1995. URL: `http://www.cs.chalmers.se/~boquist/lic.ps`.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[Cho88] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[CKT86] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimizations in the R(n) programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–523, October 1986.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow-inference. In A. Mycroft, editor, *Static Analysis Symposium (SAS)*, volume 883 of *LNCS*. Springer Verlag, September 1995. URL: `http://www.it.kth.se/~kff/fvSAS.ps`.

[Fax96] Karl-Filip Faxén. Flow Inference, Code generation, and Garbage Collection for Lazy Functional Languages. Licentiate Thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, January 1996. URL: `http://www.it.kth.se/~kff/TRITA-IT-9601.ps`.

[FW87] J. Fairbairn and S. C. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

[GM86] P.B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86*

Symposium on Compiler Construction, volume 21, pages 11–16, Palo Alto, CA, June 1986.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984. Available from `http://www.cs.chalmers.se/~johnsson`.

[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. Available from `http://www.cs.chalmers.se/~johnsson`.

[KH89] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, TX, January 1989.

[PJ92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[PJ96] Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proceedings of the European Symposium on Programming*, Linköping, April 1996.

[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.

[PJPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the International Conference on Functional Programming*, Philadelphia, 1996.

[SH89] Peter A. Steenkiste and John L. Hennessy. A Simple Interprocedural Register Allocation and Its Effectiveness for LISP. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[SH97] Marc Shapiro and Susan Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Conference Record of POPL'97: 24nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[SNvGP91] Sjaak Smetsers, Erik Nöcker, John van Groningen, and Rinus Plasmeyer. Generating efficient code for lazy functional languages. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts, July 1991.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.

[Wal86] David W. Wall. Global Register Allocation at Link Time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, New York, 1986.