Interprocedural Register Allocation for Lazy Functional Languages

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

Abstract

The aim of this paper is two-fold; first, we develop an interprocedural register allocation algorithm, an extended variant of Briggs' optimistic graph colouring. We use interprocedural coalescing as an elegant way to achieve a custom-made calling convention for each function. We add a restricted, and cheap, form of live range splitting in a way that is particularly useful for call intensive languages.

Second, we apply our interprocedural register allocation algorithm to code generated from a lazy functional language. In doing this we use a monadic intermediate code, well suited for analysis and program transformation. We use program transformation techniques and the result of an abstract interpretation analysis on the intermediate code to eliminate all unknown control flow in the program. This will improve the chances of the register allocator to produce good results.

Preliminary measurements show that we are able to eliminate up to 95% of all stack references compared to the Chalmers hbc compiler, and to a large extent even further reduce the total number of memory references.

1 Introduction

Lazy functional languages have always posed special challenges to compiler writers, the main reason being, of course, that the languages are very abstract and "far from the machine". One such challenge has been making good use of registers. Unfortunately, implementations of lazy functional languages have often been very poor at utilising registers. This has become more and more critical with recent trends in computer architecture emphasising so called RISC processors (Reduced Instruction Set Computers). These architectures typically provide a large register set that is much faster to access than memory, making it essential for an implementation to use the registers in an efficient way.

In fact, Hennessy and Patterson viewed register allocation as the technique "which adds the largest single improvement" among various compiler optimisations [19].

In this paper we will show what is required in a back end for a lazy functional language to get good register utilisation. As we will later discuss, lazy functional languages have certain characteristics that often make traditional optimisation and register allocation methods produce far from satisfactory results. Even though register allocation is normally done on a low level representation of the code, we will have to attack the problem at different levels to make the register allocation succeed well in our context.

The main contributions of this paper are:

- The intermediate code is functional in nature and uses monads to express the sequential properties of the code. Since the code is functional, it is well suited for analysis and program transformation. An abstract interpretation analysis is used to eliminate all *unknown control flow*.
- In the area of register allocation, we show how a Chaitin-style graph colouring algorithm can be extended to work interprocedurally. We add a restricted form of *live range splitting* without incuring too high additional cost. We also use *interprocedural coalescing* as an elegant way to achieve a custom-made calling convention for each procedure.
- Rather than using "standard techniques" to speed up graph reduction, like *strictness* and *boxing analysis*, we show how interprocedural register allocation can make the actual *graph reduction machinery* more efficient. The main reason for using interprocedural methods is that lazy functional languages are extremely *call intensive* (in some sense even more so than strict functional languages).

The organisation of the rest of this paper is as follows. In section 2 we discuss previous work. We continue in section 3 motivating why we believe interprocedural register allocation to be especially important for lazy functional languages. In section 4 we explain our intermediate code and in section 5 describe the register allocation algorithm. We verify our claims with measurements in section 6. Finally, we give conclusions and future work.

2 Previous work

2.1 Register allocation

Graph colouring has since long been considered a clever and yet simple abstraction to use for register allocation. It was first implemented in the classical "Yorktown allocator", developed by Chaitin and his colleagues at IBM [10, 11]. Beside Chaitin's the main influential work is Chow and Hennessy's *priority-based colouring* [13]. Starting from either

Chaitin or from Chow and Hennessy, many *global*[1] register allocators have been developed (see [6] for an overview).

However, few interprocedural allocators have been presented, mainly because most previous work have been targeted towards imperative languages and therefore naturally interested in making the allocation succeed well on loops (for which global allocation is enough). The popularity of unrestricted separate compilation is probably partly responsible for this as well. Recently, however, it has been more accepted that interprocedural register allocation is needed, mainly for two reasons: Firstly, to take advantage of the large register sets that modern RISC processors have. Secondly, to reduce the procedure call and return overhead.

Wall [33] does program-wide register allocation at link time. Estimated usage frequencies are used to decide what variables should be allocated to registers. An observation used by Wall, and by most other interprocedural allocators, is that local variables of procedures that cannot be active at the same time can be allocated to the same registers.

Chow [14] and Steenkiste and Hennessy [31] present methods where the allocation is done on one procedure at a time, in contrast to Wall's method, but interprocedural information is used to reduce the procedure call and return overhead. By compiling the procedures according to a bottom-up ordering of the procedure call graph, and avoid using the registers used by descendants in the call graph, there is no need to save and restore registers around procedure calls. A problem with the bottom-up methods is that they risk running out of registers high up in the call graph. According to Steenkiste this is normally not a problem since his Lisp programs "normally spend most of their time in the bottom of the call graph". Recursion is handled by taking the strongly connected components of the procedure call graph.

Santhanam and Odnert [28] develops a two-phase compilation method that results in program-wide register allocation, trying to allocate global variables better than Wall. They also move spill code out of frequently executed regions.

Our approach is similar to Wall's in that it is program-wide. However, instead of using a method based on usage frequencies, we use optimistic graph colouring to allocate registers. This also enables the use of interprocedural coalescing, which will help us construct a completely custom-made calling convention for each function (see section 5.4), and even further reduce the procedure call and return overhead.

Live range splitting

Live range splitting is the fundamental colouring heuristic in Chow and Hennessy's priority-based colouring. However, for this to be possible they have to sacrifice much of the accuracy of Chaitin's allocator. In a Chaitin-style allocator there is no easy way to add splitting, mainly because the cost of keeping the *interference* graph up-to-date as live ranges are split is too high.

Unfortunately, it turns out that in our interprocedural setting we need both live range splitting and at the same time the greater accuracy of a Chaitin-style framework. To combine these seemingly contradictory goals we have introduced a restricted form of live range splitting that can be added to a Chaitin-style framework without too high cost. Briggs [7] discuss a similar problem, although in an intraprocedural setting. His solution is to aggressively split live ranges at certain strategic points before the interference graph is built. Our solution instead restricts splitting to what we call interprocedural live ranges (arguments and results from function calls) and we will add splitting to the allocator in much the same way as *spilling* normally is handled. This will be discussed in greater detail below (see section 5.6).

[1] By global we mean optimisations "across basic block boundaries" but not using information from other procedures. To avoid confusion we will try to use the word *intraprocedural* instead of global in the rest of this paper, also emphasising the distinction compared to *interprocedural*. By interprocedural we mean using information from other procedures, possibly (but not necessarily) compiling the complete program at once.

2.2 Register allocation for lazy functional languages

Not much work have been done on advanced register allocation specialised to lazy functional languages.

Native code

The well known Chalmers Lazy-ML compiler by Augustsson and Johnsson [21, 5] uses a cleverly constructed circular code generator [20] and the resulting register allocation is a kind of *register tracking* algorithm inside each basic block, i.e., it is *local*. Smetsers *et al.* [30], compiling the language Clean, go one step further in using an ad-hoc global (intraprocedural) algorithm, for example in the m68k port [17]. The Clean global register allocator tries to "cache" the top elements of the abstract stacks over basic block boundaries.

The $\langle \nu, G \rangle$-machine [4] once used a global graph colouring algorithm, resulting in a 25% speedup compared to a simple code generation algorithm [Augustsson, personal communication].

Generate C

A popular approach recently has been to compile the lazy functional language into some form of the C language, and thereby take advantage of the great deal of effort put into writing register allocators and other low level optimisations in standard C compilers, for example the GNU C compiler. Gcc uses a global graph colouring algorithm based on usage priorities.

The FAST compiler [18], by Hartel, Glaser and Wild compiles into a very "direct" form of C, every function definition in the original source has a clear correspondence in the resulting C code (called Functional C). Most of the arguments remain the same since the compiler includes quite advanced strictness and boxing analysis. A serious drawback is that standard garbage collection techniques do not work. This situation is remedied by Langendoen and Hartel in the FCG code generator [25]. Here, the Functional C is further compiled, and an explicit stack added to support the garbage collector. The result, Koala code, is very much like "standard" abstract machine code, for example G-machine code, and the C compiler is used as a portable assembler.

The Glasgow Haskell compiler, by Peyton Jones *et al.*, also has C as its primary target. They describe many optimisations and tricks to get the mapping of the abstract machine onto C as efficient as possible [27].

However, all the "generate C" approaches suffer from not having the "full control" that a native code generator can give and they have to tweak the C compiler in various ways to get it to better understand particular properties of lazy functional programs, like tail calls. It is often difficult to pass arguments in registers when generating C.

Perhaps the most serious problem, however, is that the global register allocation done by the C compiler, is not particularly well suited to handle the high function call frequency in code generated from a lazy functional language. We will discuss this in more detail below (section 3).

Strict functional languages

To our knowledge, interprocedural register allocation methods have not been applied to lazy functional languages before. If we look at the strict world, Steenkiste, as described above, have used bottom-up interprocedural register allocation for Lisp.

Register allocation in general seems more well examined for strict functional languages than for lazy ones, see for example the SPUR Lisp compiler [26], the Orbit Scheme compiler [23] and all the work around the SML of New Jersey compiler [2, 3] and more recently in [16].

3 Why interprocedural?

The ultimate goal of interprocedural register allocation is to *minimise function call penalty* [14]. Because of this we might expect it to work equally well for all call intensive languages, both strict and non-strict functional languages. However, lazy functional languages have certain features that we believe make them particularly amenable to interprocedural register allocation:

3.1 No inner loops

For conventional (imperative) languages, the most important parts to optimise are the so called *inner loops*. For pure functional languages there are no loops, instead all looping is done using recursion. For lazy functional languages it gets even worse, many of the recursive "loops" produce so called lazy data structures, e.g. a lazy list. This means that not all loop iterations will happen "at the same time". Instead, a loop iteration will take place each time a cons-cell is demanded from the resulting list. If we aim to keep a value in a register over the entire loop in this case, we will probably have to do so over a large portion of the procedure call graph, and hence we need interprocedural register allocation.

One special form of recursion that an intraprocedural register allocator could take advantage of is tail calls (to the same function), and thus forming an intraprocedural loop. Unfortunately, most recursion is not that simple.

3.2 Unknown calls

Maybe the greatest complication with lazy functional languages is that the code gets scattered with calls to `EVAL` (or its equivalent depending on the abstract machine used), the procedure in the runtime system that is used to evaluate a suspended computation (closure) to weak head normal form. This means that function calls appear even more often, and in different places, in the actual code than one might think by looking at the source code. A call to `EVAL` in general also means an *unknown* call which complicates matters even more. The fact that we have unknown calls is a problem for our interprocedural register allocator, so we will have to take care of making all calls known at the intermediate code level (see section 4.3). Once this is done, the allocator will help reduce the call penalty for all places where something needs to be evaluated.

Unknown calls occur also in strict functional languages, in the presence of higher order functions, but not as often as in a lazy language where calls to `EVAL` seem to occur almost everywhere in the code. To some extent this can be improved by strictness analysis.

3.3 Parameter passing

Passing parameters in registers is generally considered to be important for call intensive languages. In its simplest form, arguments are copied to the correct parameter registers just before a call. This may save some memory accesses compared to passing parameters on the runtime stack, but if we want parameter passing to succeed really well it is important that the arguments are actually computed in the correct registers (sometimes called *targeting*). This may be very hard to accomplish if the code is scattered with calls to `EVAL`. A further complication is that many (perhaps most) function calls are not direct calls to a known function, but instead indirect via `EVAL`.

As we will show later, interprocedural register allocation helps us avoid having a fixed calling convention, i.e., dedicating certain machine registers for parameters. Instead we will use ordinary *virtual registers* to pass arguments and then let the graph colouring assign colours (machine registers) to the argument virtual registers. The key to all this is the interprocedural coalescing in the register allocator (see section 5.4).

Similar problems arise for function return values and we handle them analogously, leaving it to the graph colouring to decide what registers to use for each function. In general we will use several registers to return a result (a complete node).

4 GRIN - the intermediate code

Before we explain our register allocation algorithm we will describe the intermediate code, GRIN. For the moment we will assume that we are compiling a first order lazy functional language in a pure interprocedural compilation environment, i.e., we have the complete program available at compile time. We will later discuss how to get rid of these restrictions.

4.1 An example

We use an intermediate code that, in its essence, is very much like the well known G-machine code [21]. The main difference is that it uses explicit names for function arguments and local variables (temporaries) instead of "abstract machine stack-locations". The ultimate aim is of course to allocate as many as possible of these names to registers. The code is functional and uses a state monad to express sequentiality. Instead of giving a formal semantics for GRIN we will show an example of how a simple program (see Figure 1) can be translated, and explain GRIN constructs as they appear. For more details the reader is referred to [6].

GRIN for the sieve program

In [22], an intermediate code called GRIN, *Graph Reduction Intermediate Notation*, was introduced. It was basically an imperative, procedural version of G-machine code, but used local variable names instead of abstract stack locations. As an example, EVAL was seen as an ordinary procedure that, as a side effect, turned the graph to which its argument pointed to, into weak head normal form.

To be able to reason more easily about intermediate code we have developed a more functional version of GRIN. To handle the graph manipulation we introduce a "graph reduction monad". For general descriptions on how to use monads, see [32]. Our state monad have operations to load, store and update nodes in the heap, imitating what would happen in a real implementation.

```
main = sieve (upto 2 1000)

sieve l = case l of
              []     -> []
              (x:xs) -> x : sieve (filter x xs)

filter y l = case l of
                 []     -> []
                 (x:xs) -> if x `mod` y == 0
                           then filter y xs
                           else x : filter y xs

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

Figure 1: The sieve program

For the sieve program, the `main` function will build an initial piece of graph and then call `PRINT`, to start the graph reduction:

```
main  =
    store (Cint 2) ; \a ->
    store (Cint 1000) ; \b ->
    store (Fupto a b) ; \c ->
    store (Fsieve c) ; \d ->
    PRINT d
```

The result of every operation is assigned to a new variable name using the monad bind operation, written as `;`. In this way we can express the sequential nature of the code while still retaining a completely functional program! It is in fact even possible to run our GRIN code as a Haskell program, with minor syntactic changes and the definition of a suitable monad. The `store` monad operation builds a new node in the heap and returns a pointer to it. A node is always distinguished by a tag. It could be either a value node (constructor application) or a suspended function application (closure). By convention, function tags start with an "`F`" and constructor tags with a "`C`". For simplicity we use one tag for each possible closure but this is not necessary, we could equally well represent an application as "`App fun args`". What is important, though, is that we in some way can find what function an application represents. For the moment, all function and constructor tags will be fully saturated, since we are assuming a first order language. Note that we have not fixed any concrete node representation or said "how many words a node is".

There are three kinds of variables: ordinary node pointer variables, variables holding basic values (ending in a `'`) and so called *vector variables* (ending in a `_`). Vector variables can hold more than one ordinary value, for example a complete node. Whereas pointers and basic values are meant to be possible to store in a single register, a vector variable might need several registers.

The function `PRINT`, which really is what drives evaluation of the program forward, is normally considered to be a part of runtime system. In GRIN, however, we can write it directly. Here is a simple version that prints a list of integers:

```
PRINT p =
    EVAL p ; \v_ ->
    case v_ of
       Cnil       -> unit Cnil
       Ccons x xs -> EVAL x ; \(Cint x') ->
                     print_int x' ; \() ->
                     PRINT xs
```

The `PRINT` function starts by evaluating its argument using the general `EVAL` procedure, which is used to evaluate an arbitrary closure to weak head normal form. The result, a complete node, is bound to a vector variable. The number of "parts" in a vector variable need not be fixed, in this case it can be one, a single `Cnil`, or it can be three, a `Ccons` tag and two pointer arguments. If `Cnil` is found, `PRINT` is done and returns using the `unit` monad operation. In the `Ccons` branch, when calling `EVAL x` we know that the returned value must be a `Cint` node, so we can use a pattern matching lambda binding.

The monad operation `print_int` represents the side-effect of printing a number. Since GRIN is completely functional this is done by altering the monad state (`print_int` returns the unit element). Finally `PRINT` tail calls itself recursively to print the rest of the list.

The "ordinary" functions `upto`, `filter` and `sieve` will behave much like they would in standard G-machine code. First `sieve`:

```
sieve l =
    EVAL l ; \v_ ->
    case v_ of
        Cnil       -> unit Cnil
        Ccons x xs -> store (Ffilter x xs) ; \a ->
                      store (Fsieve a) ; \b ->
                      unit (Ccons x b)
```

In the `Ccons` branch `sieve` will build a `Ccons` node with a tail that is the unevaluated function application "`sieve (filter x xs)`".

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

The `upto` function uses some basic operations that is evaluated immediately and returns a basic value (`>` and `+`). In our monad notation this is done using a `unit` operation binding its result to a new variable name. In an actual implementation this would correspond to an operation done with one or a few machine instructions producing its result in a register. The only operation that builds new graph is `store`.

The `filter` function contains nothing new.

The `EVAL` procedure, just like `PRINT`, is normally considered a part of the runtime system, but we can write it in GRIN. It simply checks the tag of its argument and, if it is a closure, calls the corresponding function. The `fetch` monad operation loads a complete node from the heap into registers (a vector variable).

```
EVAL p =
    fetch p ; \v_ ->
    (case v_ of
         Cint i'     -> unit (Cint i')
         Cnil        -> unit Cnil
         Ccons x xs  -> unit (Ccons x xs)
         Fupto m n   -> upto m n
         Ffilter y l -> filter y l
```

```
      Fsieve k    -> sieve k
) ; \u_ ->
update p u_ ; \() ->
unit u_
```

Note how the result of the case expression, either `unit` of an already evaluated constructor or the result of a function call, is bound to the vector variable `u_` The original redex, pointed to by `p`, is then updated using the monad operation `update`. Finally the value, still in `u_` is returned.

In GRIN, all functions return values using a vector variable. This captures our intentions of being able to return values from functions using more than one register.

In principle, the case expression in `EVAL` has to contain all possible nodes (values and closures) that occur in the program. We will later show how to eliminate this potential problem by removing `EVAL` altogether.

4.2 Control flow

When doing aggressive low level optimisations it is important that the compiler can "understand" the control flow. However, for the translated sieve GRIN program, the control flow is not very useful. To see this we look at its *call graph*, in Figure 2 (here nodes represent procedures and edges mean "might call", i.e., a call to the procedure appears in the code).

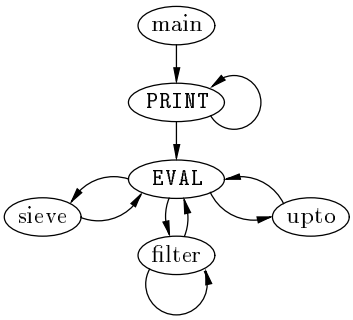

Figure 2: The lazy call graph

The important thing to note is how `EVAL` gets involved in most calls. For example, the arc from `sieve` to `EVAL` means that `sieve` might need to evaluate a (possible) closure and the arc from `EVAL` to `sieve` that a `sieve` closure might be evaluated somewhere (by someone). Even though the actual dynamic, i.e., runtime, control flow is much more restricted than this, the compiler cannot know that. For our purpose, interprocedural register allocation, the graph above is a problem since local variables in a procedure normally will interfere with all other locals in procedures that can be active at the same time. This will manifest itself in the data flow analysis done during register allocation (see section 5.3). The data flow information will flow "too much", giving rise to sub-optimal results (unnecessary conflicts). It is therefore a bad thing if many procedures together with `EVAL` become a large mutually recursive procedure group.

4.3 Unfolding `EVAL`

Noting that the problem stems from calling a general `EVAL` procedure, we will eliminate all those calls by specialising each call to `EVAL`. If we simply unfolded (inlined) every call to `EVAL` we would risk getting code explosion, since the case expression in our `EVAL` procedure in principle needs to include every possible value and closure in the program. Instead we will use Johnsson's *constructor analysis* [22]. In that paper he proposed inlining as a general method to enlarge procedures to get better results from register allocation. He also wanted to do more general fold-unfold transformations [9] to improve the efficiency of the code. The main part of the paper gave an analysis based on abstract interpretation to find, for each pointer in the intermediate code, a safe approximation to what kinds of nodes it might point to in the heap. By reducing the size of the "`EVAL` case expression" this suddenly makes unfolding all calls to `EVAL` feasible.

As an example, for the call "`EVAL p`" in the `PRINT` procedure above, the analysis will show that `p` must point to a `sieve` closure. Using this we can unfold `EVAL` and at the same time specialise it:

```
PRINT p =
    fetch p ; \v_ ->
    (case v_ of
        Fsieve k -> sieve k
    ) ; \u_ ->
    update p u_ ; \() ->
    unit u_ ; <the rest of PRINT>
```

The resulting code will be further optimised by later transformations. As we unfold calls to `EVAL` we also do suitable renaming to make sure that we do not get any name capture problems.

After unfolding all calls to `EVAL`, the `EVAL` procedure will completely disappear. Assuming that the constructor analysis succeeds as well as possible, which it will easily do for this simple program, we will get the improved call graph shown in Figure 3.

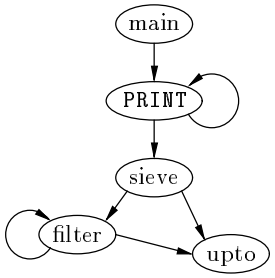

Figure 3: Improved lazy call graph

Hopefully, during later optimisations we should be able to benefit from the knowledge that there are no unknown calls anymore, the `upto` procedure does not do any calls at all, ie it is a leaf procedure, and the fact that `sieve` is not recursive at all but might call either `filter` or `upto`.

Higher order functions

This example used only first order functions. However, according to the author of the constructor analysis, higher order functions should pose no special problem. At this low level, partial applications are simply a tag followed by arguments just like any other value. In a way similar to the `EVAL` procedure it is possible to construct a number of different `APPLY` procedures, for different number of additional arguments. We will also need a number of different tags for each partially applied function, one for each possible arity.

Related approaches

The constructor analysis is closely related to the work by Chase, Wegman and Zadeck [12], where heap allocated structures are analysed. The information is intended to be used to guide later optimisations.

The constructor analysis and the unfolding of `EVAL` taken together is a way of determining the control flow of the program (or at least to make it more precise). In that sense it is similar to Olin Shivers *control flow analysis* [29] for Scheme. Scheme is a strict language, but when higher order functions are used, similar problems arise as in lazy languages.

4.4 Code Generation

After a series of simple transformations [6], the intermediate code is translated into assembler code for a hypothetical RISC machine assuming an infinite number of available virtual registers (live ranges). Apart from the standard "book keeping" that needs to be done in a code generator the translation itself is rather straightforward, since the final GRIN contains only quite simple operations.

After the code generation, and some initial RISC optimisations, the code is fed into the register allocator. We will discuss further properties of the code in the description of the register allocator.

5 The register allocation algorithm

The basic idea of our interprocedural register allocation algorithm is to build an *interference graph* (conflict graph) for the complete program and then colour it. We initially assume that all values are allocated to registers (called *virtual registers*). If the allocator fails to find a colour for some value (*live range*), it will be *spilled*, i.e., kept in the stack frame and loaded/stored when there is need to. In addition to spilling, we will occasionally also *split* live ranges by inserting register-to-register copy instructions. We will later discuss the practicality of this "whole program" approach and alternatives to it.

5.1 Overview

The starting point of our allocator is the *optimistic* version of Chaitin's graph colouring algorithm described by Briggs *et al.* [8, 7]. The main differences between Briggs' allocator and ours are:

- It is *interprocedural*, both Chaitin and Briggs discuss only global register allocation.
- We add *interprocedural coalescing*, which is very effective in handling parameter passing and functions returning results in several registers.
- We introduce a restricted form of (the potentially very expensive operation) *live range splitting* in a way that adds no significant extra cost to the colouring.

We have chosen a Chaitin-style allocator as a starting point because of its greater accuracy compared to the other main alternative; Chow and Hennessy's priority-based colouring [13], even though the latter has live range splitting as one of its basic operations. As a consequence of the lower accuracy, *coalescing* (see section 5.4) is not possible in Chow and Hennessy's framework. Instead they use *copy propagation* earlier during the compilation, but that is insufficient for our purposes.

In principle, our allocator could be used for other languages than lazy functional ones, but in some sense it is specialised to the particular properties found in code generated from lazy functional languages. First, the high function call intensity makes our custom-made calling conventions really important. It is vital that function calls are made as cheap as possible. Second, we think intraprocedural splitting is less important since local variables in a procedure are seldom used many times and are never used inside loops (there are no intraprocedural loops). The exception to this is tail recursion, but in that case, splitting would probably not help at all. In contrast, for a language like Fortran, not having splitting inside procedures could be quite devastating.

The basic structure of our allocator, the so called *build-colour cycle*, can be seen in Figure 4. Its different phases are:

build – performs an interprocedural, live variable data flow analysis and builds the interference graph.

coalesce – low level copy propagation, eliminates unnecessary copies by combining live ranges.

spill costs – approximates the spill cost for each live range.

simplify – the main colouring heuristics, simplifies the interference graph, possibly choosing spill candidates.

select – assigns colours to the nodes in the reverse order as they were chosen by simplify.

spill – inserts spill code for spilled live ranges. This means creating a number of very short live ranges instead of a few long ones.

split – inserts code to split a long live range into shorter ones.

5.2 Structure of emitted code

The register allocator works on low level assembler code for a hypothetical RISC machine.

Each procedure is represented as a flow graph of basic blocks [1, pages 528-535]. The intraprocedural flow graphs are always DAGs (directed acyclic graphs). The graphs have no cycles since the code is generated from a functional language (at this stage tail recursive calls have not yet turned into loops). On the interprocedural level we link together the intraprocedural flow graphs using *call* and *return edges*. We call the resulting flow graph, for the complete program, the *Interprocedural Control Flow Graph* (ICFG) following Landi and Ryder [24].

It is interesting to note the distinction between *live ranges* (or virtual registers) and *values*. A new value can be seen as arising for each definition in the code. For imperative languages we usually say that a live range consists of several values connected by *common uses*. Given the "single assignment nature" of functional languages we might expect the two notions *value* and *live range* to be (almost) identical, but unfortunately that is not the case. Firstly, in our interprocedural setting, a live range used as a parameter register for a function can be defined in several places (all the call sites to that function). A similar phenomenon occurs for return values. Secondly, when doing our intermediate code transformations to eliminate unknown control flow (unfolding calls to `EVAL`) we introduce several definitions of the same live range, in different alternatives of the same case expression. The control flow graph joins the different alternatives of this case expression (this is one of the things

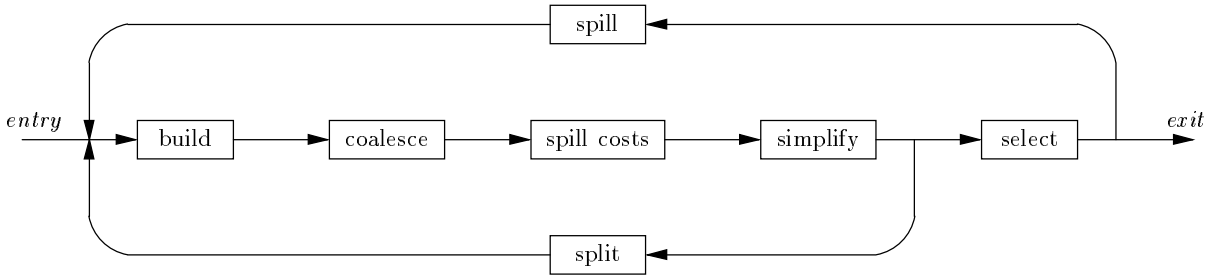

Figure 4: The register allocator

that makes the intraprocedural flow graphs into DAGs and not just trees). Furthermore, **coalesce** (see section 5.4) will completely break most remaining single assignment properties as it combines live ranges that do not conflict with each other. So, in general, a live range will represent several values and we are no better off than if we had been compiling an imperative language.

Recursion

Although interprocedural register allocation will often avoid the saving and restoring of local variables around function calls that a global allocator will have to do, this is not always possible. When a local variable of a procedure is *live*[2] across a recursive function call, i.e., its value may be used after the call, it will have to be saved to memory before the call and then restored after the call. If we did not do this, we would risk that a recursive invocation of the same procedure clobbered the register. In general we will have to find the strongly connected components (SCCs) of the procedure call graph to decide if a call can be recursive. We need only save and restore local variables for calls inside the same SCC. There are some variations on where the save and restore instructions can be placed. In Figure 5 we show two different ways, as used by Steenkiste [31] and Wall [33] respectively. Each node represents a procedure and is marked with its register usage. Edges represent calls and are marked with the register save operations done before the call.

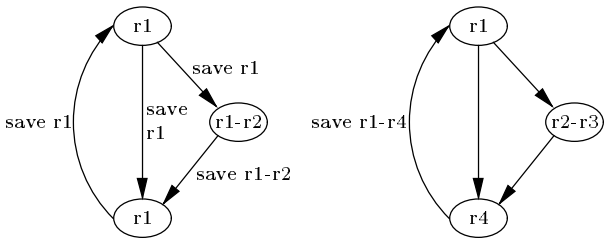

Figure 5: Different ways to handle recursion, Steenkiste vs. Wall

In the first solution, each procedure's live variables are saved *incrementally* for each function call. In the second, the registers for the complete SCC are saved on the *back edges* of the call graph. A back edge is an edge in the flow graph whose head *dominates* its tail [1, pages 603-604]. For Steenkiste's bottom-up allocation, the incremental method is natural since procedures in the same SCC can reuse registers. Because of this, more registers will be free higher up in the call graph. Wall simplifies the allocation process by removing all back edges from the call graph before the allocation is done. After allocation he inserts saves and restores only at the back edges. In our allocator, we have chosen the incremental method because of its above-mentioned advantages, although the choice is not very critical compared to in Steenkiste's algorithm.

[2] 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.

5.3 Build

Our **build** phase uses an iterative *data flow analysis* to find, for each point in the program, the set of *live variables* at that point. In our case, this will mean an *interprocedural live variables analysis*. Using that information, we can build the interference graph. To compute the live variables information we solve a slight variation of the following global data flow equations. A pair of set equations for each basic block b in the interprocedural control flow graph:

$$\begin{cases} Live\_in(b) = Gen(b) \cup (Live\_out(b) \setminus Kill(b)) \\ Live\_out(b) = \bigcup_{x \in Succ(b)} Live\_in(x) \end{cases}$$

where $Live\_in(b)$ is the set of all live ranges live at entry to basic block b, $Live\_out(b)$ means live at exit from basic block b, $Gen(b)$ are the live ranges generated in basic block b ,i.e., used before they are defined, and finally $Kill(b)$ are the live ranges killed (defined) in basic block b. $Succ$ is the successor relation in the ICFG, i.e., there can be both intraprocedural and interprocedural successors (call/return edges).

In the $Live\_out$ equation, for those successor edges that are interprocedural call edges, we introduce an additional intersection with the incoming parameter registers. The reason for this is that live variables otherwise would flow unnecessarily along *unrealisable execution paths*. This problem is usually called the *calling context problem* [24].

The data flow equations are solved by iteration. We use a reverse depth first ordering of the basic blocks in the flow graph [1, page 660] to speed up the convergence.

Once we have the live variables information we can build the interference graph in a single backwards walk of the code. In principle, for each instruction, all defined live ranges will conflict with the live set at that instruction. Doing this we are careful not to introduce any conflicts for copy instructions, as this would stand in the way of **coalesce** [10].

5.4 Coalesce

Normally the **coalesce** phase is used as a "cleanup" after a code generator or optimiser that has been allowed to insert lots of redundant copies. Our main use of **coalesce** will

be as an elegant way to achieve a custom-made calling convention for each procedure (for parameter as well as return registers). When generating code for a function call to the function `f`, assuming the actual arguments can be found in the virtual registers `a1` and `a2`, we will generate the code:

```
copy a1 -> r1
copy a2 -> r2
call f
```

where `r1` and `r2` are the virtual registers in which f expects its arguments. In the same style, the first thing `f` will do is:

```
f:
    copy r1 -> t1
    copy r2 -> t2
```

where `t1` and `t2` are the virtual registers that will be used when `f` needs its arguments. Introducing a lot of copies for each function call might seem strange, but the point is that we want the register allocator to have "maximal freedom" to choose the registers used to pass parameters. Therefore, we want as few constraints as possible to prevent different functions and call sites from negatively interfering with each other. Hopefully, **coalesce** will remove most of the copy instructions. The ones that it leaves behind will actually be useful.

Coalesce examines each copy instruction, and checks if the operands conflict with each other. If they do not, **coalesce** will combine the two live ranges and delete the copy instruction. The two nodes in the interference graph will be merged into one, taking the union of all neighbours. For the copy `a1 -> r1` above this would mean computing the argument into the correct register (targeting). For the copy `r1 -> t1` it would mean that `f` will use its argument in the register where the caller put it. The order for coalesces may be important, and for best results, "high priority" coalesces should be made before others.

Parameter coalescing can also be used in global register allocators [7], but this suffers from not having the same freedom as we have in our setting since the coalescing often can be prohibited by the fact that a certain machine register is dedicated to be, say the first parameter register.

Returning values in registers from functions is handled in exactly the same way as parameters are, introducing extra copies that later are removed by **coalesce**. Normally we will return complete nodes using several registers, not just a pointer to a node in the heap.

When **coalesce** combines live ranges it also reduces the size of the interference graph. As live ranges are split (see section 5.9) it is necessary to restrict **coalesce** slightly.

5.5 Spill Costs

The spill costs should be an approximation, for each live range, of the savings in execution time (instruction cycles) of keeping the live range in a register instead of in memory. For a RISC machine, this approximately means counting the number of instructions where the live range occurs. However, to give good spill cost approximations it is also important to try to mimic the dynamic execution behaviour of the program, i.e., how many times a certain instruction will be executed. For imperative languages, a method that works well in practice is to assume that all loops are run a small constant number of iterations (typically 10). A variable that is used inside k nested loops (each of which do n iterations) will have its cost *weighted* by n^k compared to a variable that is used outside the loop.

In our allocator we analogously assume that every function is called n times. By looking at the strongly connected components of the call graph, after all unknown control flow is eliminated, we can find the *depth* of each procedure (call nesting). The weight for a certain use of a live range is then calculated as n^d where d is the depth of the procedure where the use occurs. This will result in larger costs for procedures in the bottom of the call graph. It is in line with Steenkiste and Hennessy's observation that "programs spend most of their time in the bottom of the call graph" [31] (for Lisp programs).

5.6 Simplify

Our **simplify** phase is a variant of Briggs' optimistic method to colour the interference graph [8]. The main difference is that a node that is uncolourable may not only be chosen as a *spill candidate* but may also be *split*. The reason for this is that in our interprocedural setting, a complication arises that is not found in intraprocedural (global) allocators: we get live ranges that span several procedures, i.e., they are used in more than one procedure. The question is: if such a live range is chosen as the most beneficial to spill, how should it be spilled? Since the live range can be used in several procedures, there is no single *home location* in the stack frame of a single procedure where the live range can be spilled. The number of possible procedure invocations the live range can be referenced from may be unbounded in the presence of recursion, so we can not spill it to the heap.

Restricted splitting

Our solution to this problem notes that the only way that a live range can be "transferred" between procedures is when passed as an argument to a function or returned as a result from a function. The **coalesce** phase may have constructed very long live ranges, ranging over several procedures, but the live ranges will always be *connected* in the sense that there are no "holes" in the live range when looked at in the interprocedural control flow graph. Instead of spilling such a live range we *split* it around call sites, i.e., insert copy instructions, undoing some of the combinations done by **coalesce**. The splitting transforms a long interprocedural live range into a number of new live ranges. These can be of two kinds: first, completely intraprocedural, one for each procedure where the original live range appeared. Second, for call sites where the original live range was used, we create a new live range that is used only at that call. Compare with the live range `r1` in the example of section 5.4. Of these two kinds of new live ranges, the second are so short that they do not influence the register pressure in particular. The first kind can, if needed, be spilled as any other intraprocedural live range during the next run of **simplify**.

In some sense this is a completely natural solution. A very long lived live range should not be spilled in the normal sense since the high register pressure that makes it uncolourable probably is that high only in a small portion of the total lifetime of the live range. Spilling it all the time would result in an unnecessarily high cost. This is in fact one of the main disadvantages with standard Chaitin-style colouring and it becomes even more apparent in our interprocedural setting where live ranges may become very long. In our framework, a long-lived live range is often held in a register for some regions of the program (where it is heavily used) and in memory in others.

Aggressive vs. restricted splitting

In general, live range splitting is considered too expensive to add in a Chaitin-style colouring framework, mainly because the cost of keeping the interference graph up-to-date as live ranges are split can be very high. Briggs discusses this problem in some detail [7] and his solution is to aggressively split live ranges once before the interference graph is built for the first time. The splitting is done at certain *strategic points*, for example before the entry to a loop, trying to maximise the chances of a live range used inside the loop to be allocated to a register. Briggs' colouring algorithm has to go through quite a large reorganisation to support this addition, for example **coalesce** has to be restricted to not combine live ranges that were just split. He also has to go through a lot of effort to later "undo" as many of the unnecessary splits as possible. He uses a *biased* form of colouring that tries to give related live ranges (originating from the same split) the same colour, so that the copy instruction can be deleted.

The advantage of our solution compared to Briggs' is that our splitting is very restricted, it can only happen to interprocedural live ranges. Therefore it is quite cheap to simply add it to the allocator in much the same way as spilling is handled. When **simplify** finds that one or more interprocedural live ranges would be the most beneficial to spill (according to the calculated spill costs), it will instead continue to the **split** phase, to do the actual split code insertion. After that the interference graph will be rebuilt (see Figure 4). Experiments suggest that **simplify** often finds several live ranges to split in "one go", even further making the cost of splitting feasible.

5.7 Select

The **select** phase will only be entered if no live range splits occured during **simplify**. It assigns colours to nodes (spill candidates) in the reverse order as determined by **simplify**. When a spill candidate is about to be coloured it may very well receive a colour even if it has N or more neighbours[3]. This happens if some of its neighbours have received the same colours. In these cases, the *optimistic* colouring has paid off. If **select** fails to find a colour for a spill candidate, it will be marked as a definite spill. After **select**, if any live ranges were marked we insert spill code (in **spill**) and rebuild the interference graph, otherwise the colouring is complete.

5.8 Spill

The **spill** phase traverses the program and transforms spilled live ranges from being held in a register to being kept in memory. This basically means creating home locations, in the procedure stack frame, for all spilled live ranges. We insert a load from the home location before each use of a spilled live range and a store to the home location after each definition of a spilled live range (since we assume a RISC like architecture). We do some optimisations to this simple scheme, a live range should not be reloaded twice if two uses are *close*. We use Chaitin's definition of close, i.e., two uses of a live range are close if no other live range goes dead (has its last use) in-between. This can be read as "nothing interesting happens to the register pressure between two uses that are close".

[3] Whenever we use the letter N it will mean the available number of colours (machine registers).

5.9 Split

The **split** phase traverses the program and introduces copy instructions at function boundaries and around call sites for those live ranges that should be split. For simplicity, all occurrences inside one procedure of a live range to be split are replaced by a single new live range.

After a live range has been split we will restrict **coalesce** on the resulting live ranges during the subsequent runs of the build-colour cycle. A part of a split live range is not allowed to be coalesced if the resulting live range would get N or more neighbours of degree N or higher, and therefore would risk getting spilled. Briggs calls this *conservative coalescing* [7].

6 Measurements

To support our experiments we have written a back end for a lazy functional language generating code for the Sparc processor.

We will here concentrate on the behaviour of the register allocator trying to isolate the results of our interprocedural register allocation algorithm, rather than measuring overall speed of the programs. To do this we have used the *Spix tools* [15] to measure dynamic instruction counts for different kinds of instructions.

6.1 Stack references

The most important task of a register allocator is to transform memory references to register references. This typically means keeping the values of local variables and other temporaries in registers instead of in the procedure stack frame. Therefore, counting *stack references* should be a good measure on how well the register allocator has succeeded [31].

To see what can be gained by doing interprocedural register allocation we have compared the dynamic number of stack references for the Chalmers hbc compiler and our own allocator. For hbc we measure the code it generates for the Sparc processor with best (low level) optimisations turned on[4]. For our allocator (called GRIN in the figures) we first assume it has been given a sufficient number of registers so that it needs no spill code. Below we will see how it behaves when we force it to spill.

We have two small test programs. Due to the absence of a real front end we have not yet been able to run our allocator on larger examples. Hopefully, the results for our test programs will be fairly representative:

queens Solving the n-queens puzzle, for n=10. The code size for the optimised program is 1.4 Kb.

sieve The sieve of Eratosthenes prime number filter. The result is the sum of all prime numbers less than 10 000. Code size 0.7 Kb.

We have been very careful to see to that the comparisons become as fair as possible when it comes to counting stack references. The programs are written in such a way that, for example, the strictness is exactly the same for the different compilers. In fact, there is very little strictness to exploit in these programs, they are "pure graph reduction" programs. The heap consumption is also roughly the same for the different implementations, so we can assume that the same "graph reduction in the heap" is taking place. What differs is only the *graph reduction machinery*.

[4] Using hbc version 0.999.7 with the flags "`-O -msparc8`".

queens		hbc	GRIN
Total	instrs	174,338,932	52,920,599
	load	58,197,896	12,593,346
	store	30,329,892	10,272,698
Stack	load	29,267,187	1,011,848
	store	15,361,397	1,011,847

sieve		hbc	GRIN
Total	instrs	72,852,729	28,862,038
	load	21,152,009	6,221,826
	store	13,316,775	7,006,997
Stack	load	10,190,813	2,308,504
	store	7,031,775	2,308,503

Figure 6: Instruction counts

In figure 6 we compare the total number of instructions, the total number of loads and stores and the number of loads and stores to stack[5] for the two implementations (all are dynamic instruction counts). We can see how our allocator is very effective in reducing the number of stack references compared to hbc, 95% for the queens programs and 75% for sieve. The stack references in GRIN consists of saving and restoring the return address (and a few local variables that are live around possibly recursive calls). See below for what happens when the allocator starts to spill.

It should be noted that the big difference in total number of memory references between hbc and GRIN does not come from hbc doing "more graph reduction" in the heap. Instead it probably comes from differences in the implementation of hbc's abstract machine compared to our implementation. In hbc a load from the pointer stack is very often followed by one or more loads, first loading a tag from the heap, then another load using the tag as an address to a *dispatch table*. Of these loads, only the first would show up as a "stack load". Therefore one might argue that our interprocedural register allocation can eliminate more than what could be called "classical stack references". A further reason for the big difference is that values from the heap needed several times have been allocated to registers with our allocator, but reloaded with hbc.

6.2 Spill code

When using our register allocator on real programs, i.e., larger programs, it is important that it is able to handle situations where there are too few available registers without loosing too much in performance. The only way to test this is really to run the allocator on large programs. Due to the absence of a real front end we have not yet done this. Anyway, we believe that the results for our test programs are fairly representative. In figure 7 the number of spill instructions are given as we decrease the number of available registers (the first row is without spill). Numbers are relative to the preceeding row. The absolute numbers of stack references might seem to increase quite fast as we lower the available number of registers, but looking at the total number of executed instructions we see that this does not increase at all that fast. For the queens program the difference is only 6% when going from 14 to 8 registers, for sieve there is hardly no difference at all. In fact, the queens program even got faster when going from 14 to 13 registers! The reason for this was that not exactly the same coalescing occured in the two programs, due to a live range being spilled in the latter and thus slightly changing the code. This suggests that our register allocator is not yet perfectly tuned.

[5] For hbc we sum the references to its two stacks.

queens	Stack		Total
	load	store	instructions
14 reg	1,011,848	1,011,847	52,920,599
13	+2	+1	52,572,452
12	+1,450	+1,449	52,574,672
11	+70,353	+35,538	52,750,871
10	+105,891	+35,538	52,787,132
9	+696,300	+382,965	53,553,785
8	+731,115	+1,077,819	56,404,999

sieve	Stack		Total
	load	store	instructions
10 reg	2,308,504	2,308,503	28,862,035
9	+2	+1	28,862,038
8	+2,460	+2,459	28,866,958
7	+2,460	+1,230	28,869,418
6	+785,401	+9,999	28,878,188

Figure 7: Spilling

6.3 Execution time

The total number of instructions executed (Figure 6) gives a hint on how fast our code is compared to code generated by hbc. However, for modern computer architectures, things like super-scalar features and different cache organisations make it very different to draw conclusions about execution times by just looking at instruction counts.

For our test programs, though, the speed difference compared to hbc seems to be about the same as indicated by the instruction counts, i.e around 3 times. More measurements on larger programs are needed to say anything definite about this.

7 Conclusions

7.1 Graph reduction

We have shown how interprocedural register allocation can be used to speed up graph reduction. We have not done this the "standard way", i.e., by doing less graph reduction and instead more direct computations. We still do the same graph "shuffling" in the heap. In principle, the things that we do different from a typical G-machine implementation, like hbc, are:

- instead of putting values on stacks (pointers, values and return addresses) as control flow changes, we keep everything in registers. When this is not possible, due to recursion or too high register pressure, we spill to the standard system stack. We try to spill values that are the most beneficial, i.e., both cheap to spill and which decrease register pressure.

- as a special case of the above, we very efficiently pass arguments to functions in registers and also return results from functions using several registers.
- we have no general `EVAL` procedure. In our implementation all function calls are to known functions. This enables optimisations (like register allocation) to succeed better. It is hopefully also better for the architecture.

With this in mind we could call our work "fast graph reduction machinery". We also believe the intermediate code GRIN together with our interprocedural register allocation to be a good starting point for further exploiting low level optimisations. Things like unboxing, strictness, more aggressive inlining and update avoidance can hopefully enable our register allocator to do a still better job. We plan to explore this route in the future.

7.2 Register allocation

In the actual register allocation algorithm we have shown how a Chaitin/Briggs-style colouring algorithm can be extended to an interprocedural algorithm. We have added a restricted form of live range splitting without introducing the high extra cost that normally is associated with splitting in Chaitin-style colouring. We have used an interprocedural kind of coalescing that succeeds very well in achieving a custom-made calling convention.

This taken together have resulted in a register allocation algorithm that is particularly well suited for lazy (and other call intensive) languages.

8 Future work

8.1 Limitations

Currently we have no garbage collector. This should however be fairly straightforward to implement. The complication is that it can be hard to find all *roots*, i.e., pointers to live data in the heap, if pointers reside in registers instead of on the stack. The normal way to garbage collect at the beginning of a procedure will not help since in our implementation there are no "safe places" where every live value can be found on the stack. A solution to this could be to use special purpose code to do garbage collection for each procedure. It would be an interesting experiment to be able to write the garbage collector in GRIN just as we can express other parts of the runtime system in GRIN, for example `EVAL` (see section 4.1). There is also a problem of knowing which registers contain pointers and which contain non-pointers, this can hopefully be handled with some kind of descriptors.

A further limitation so far has been that we have no higher order functions, but as mentioned earlier (section 4.3), there should be no fundamental problems in extending this to handle higher order functions.

8.2 Practicality

When discussing the practicality of our approach, there are two main issues: the constructor analysis and the register allocation algorithm.

Unfolding EVAL

It should first be noted that the constructor analysis and the unfolding of `EVAL` is not strictly necessary to make our approach work. However, the more unknown control flow that we can eliminate using the constructor analysis, the better the register allocation will succeed. Currently, we completely unfold all calls to `EVAL`, but we plan to experiment with only "partial unfoldings", having a "default branch" in the case expression. The use of profiling information could also be used to guide the unfolding, keeping only the most common branches of the `EVAL` case expression. Johnsson is currently working on methods to make the constructor analysis practical for larger programs.

Register allocation

Currently we do graph colouring on the complete program at once. This may prove to be impractical when it comes to really large programs. To solve this we can use techniques from the methods discussed earlier, i.e., either apply per-procedure register allocation as a bottom-up traversal of the procedure call graph, or we could keep the program-wide allocation, but use a cheaper method than "full graph colouring".

Issues like separate compilation can be attacked in several ways. One way is to avoid doing optimisations during program development. As the development is done, a full optimisation can be applied to the complete program (possibly at link time like in Wall [33]). Another way, if separate compilation really is desired (for programmer or practical concerns), is to use some kind of interprocedural compilation/optimisation environment that keeps track of what needs to be re-optimised etc.

Another idea could be to use something in the style of Chow's open and closed procedures [14], i.e., only apply full optimisation for calls inside the same module and use fixed conventions for all exported functions. Unfortunately this is more complicated in a lazy than in a strict language, since functions local to a module can still "escape" the module as closures are built. This can also happen in strict languages, but is probably not as common.

9 Acknowledgements

I would like to thank my advisor Thomas Johnsson without whom much of this probably never would have happened.

References

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

[2] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Proceedings of the '87 Functional Programming Languages and Computer Architecture*, pages 301–324. Springer Verlag, LNCS 274, 1987.

[3] Andrew W. Appel and Zhong Shao. Callee-save Registers in Continuation-passing Style. *Lisp and Symbolic Computation*, 5:189–219, 1992.

[4] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle \nu, G \rangle$-machine. In *Proceedings of the 1989 Conference on Functional Languages and Computer Architecture*, pages 202–213, London, England, 1989.

[5] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

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

[7] Preston Briggs. Register allocation via graph coloring. PhD Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.

[8] 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.

[9] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24:44–67, 1977.

[10] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[11] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 201–207, June 1982.

[12] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 296–310, White Plains, NY, June 1990.

[13] F. Chow and J. Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232, Montreal, 1984.

[14] 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.

[15] Bob Cmelik. *SpixTools User's Manual*. Sun Microsystems Laboratories, Inc., Mountain View, CA 94043, 1992.

[16] Lal George, Florent Guillame, and John H. Reppy. A Portable and Optimizing Back End for the SML/NJ Compiler. In *Compiler Construction, 5th International Conference, CC'94*, Edinburgh, 1994. Springer-Verlag, LNCS 786.

[17] John van Groningen. Implementing the ABC-machine on MC680x0 based architectures. Masters thesis, Department of Informatics, University of Nijmegen, November 1990.

[18] P. Hartel, H. Glaser, and J. Wild. On the benefits of different analyses in the compilation of a lazy functional language. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, Southampton, UK, 1991.

[19] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Palo Alto, 1990.

[20] T. Johnsson. Code Generation for Lazy Evaluation. Technical Report Memo 22, Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1981.

[21] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[22] 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.

[23] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: an optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, volume 21, pages 219–233, Palo Alto, CA, June 1986.

[24] W. Landi and B. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

[25] Koen Langendoen and Pieter Hartel. FCG: a code generator for lazy functional languages. In *Compiler Construction, 4th International Conference, CC'92*, Paderborn, 1992. Springer-Verlag, LNCS 641.

[26] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, volume 21, pages 255–263, Palo Alto, CA, June 1986.

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

[28] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 28–39, White Plains, NY, June 1990.

[29] Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 164–174, Atlanta, GA, June 1988.

[30] 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.

[31] 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.

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

[33] 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.