Thesis for the Degree of Doctor of Technology

Code Optimisation Techniques for Lazy Functional Languages

Urban Boquist

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, March 1999

A dissertation for the Ph.D. Degree in
Computing Science at Chalmers University of Technology

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

ISBN 91-7197-792-9
ISSN 0346-718X
Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie 1495

CTH, Göteborg, 1999

Abstract

This thesis describes a complete compiler back-end for lazy functional languages, which uses various interprocedural optimisations to produce highly optimised code. The most important contributions of this work are the following.

A novel intermediate language, called GRIN (Graph Reduction Intermediate Notation), around which the first part of the back-end is built. The GRIN language has a very "functional flavour", making it well suited for analysis and program transformation, but at the same time provides the "low level" machinery needed to express many concrete implementation details.

We apply a program-wide control flow analysis, also called a heap points-to analysis, to the GRIN program. The result of the analysis is used to eliminate unknown control flow in the program, i.e., function calls where the call target is unknown at compile time (due to forcing of closures).

We present a large number of GRIN source-to-source program transformations that are applied to the program. Most transformations are very simple but when taken together, and applied repeatedly, produce greatly simplified and optimised code. Several non-standard transformations are made possible by the low level control offered by the GRIN language (when compared to a more high level intermediate language, e.g., the STG language).

Eventually, the GRIN code is translated into RISC machine code. We develop an interprocedural register allocation algorithm, with a main purpose of decreasing the function call and return overhead. The register allocation algorithm includes a new technique to optimise the placement of register save and restore instructions, related to Chow's shrink-wrapping, and extends traditional Chaitin-style graph colouring with interprocedural coalescing and a restricted form of live range splitting. The algorithm produces a tailor-made calling convention for each function (the registers used to pass function arguments and results).

A combined compile time and runtime approach is used to support garbage collection in the presence of aggressive optimisations (most notably our register allocation), without imposing any mutator overhead. The method includes a runtime call stack traversal and interpretation of registers and stack frames using pre-computed descriptor tables.

Experiments sofar have been very promising. For a set of small to medium-sized Haskell programs taken from the nofib benchmark suite, the code produced by our back-end executes several times faster than the code produced by some other compilers (ghc and hbc).

Keywords and phrases: Lazy functional languages, compiler back-end, intermediate code, graph reduction, code optimisation, program transformation, interprocedural register allocation, graph colouring, garbage collection.

Acknowledgements

Many people have inspired and helped me during these years, and I wish to thank you all deeply, I hope you know who you are!

First I would like to express my gratitude towards Thomas Johnsson, who has been an excellent advisor. Without some of his ground breaking ideas much of the work in this thesis would probably not have happened, or at least been very different. His guidance during these years have been the perfect mix of interesting and stimulating discussions combined with leaving me alone when I had other things to do ...

Thomas and his alter ego, Lennart Augustsson, and their early implementation work was a great source of inspiration for me when I first started my PhD work. Since then I have come to know Lennart as "the man who knows everything", a great resource to have around. Thanks Lennart for answering all my questions about the NetBSD kernel.

Jan Sparud has been following me wherever I have gone for almost 10 years now. But I don't mind. On the contrary, Janne has become a really good friend and I wish to thank him for all the fun we have had together (unfortunately most of it has not been particularly related to our thesis work). However, my first stumbling steps into the wonderful world of graph reduction was taken together with him, a really enjoyable experience. Thanks for everything Janne, I really look forward to working more closely together with you again.

I would also like to thank Staffan Truvé, whom I first met as an advisor during a hectic undergraduate programming project. Staffan impressed me immensely, and he was probably the single most important reason for me deciding to become a PhD student. Thanks a lot for that.

Thanks also to the inseparable Magnus Carlsson and Thomas Hallgren, and the defector Niklas Röjemo. They all were like big brothers to me during the first years of my PhD studies, but then I outgrew them, of course. A special thanks to Niklas for all the nice Doom sessions.

Another special thanks goes to Andy Moran for reading and commenting on the semantics in this thesis, as well as his great job of always keeping me upto-date with the latest development in the computer games arena. Let me challenge you in a deathmatch soon!

A big thanks to Lars Pareto for being such a long time friend. Some of my most memorable moments have been spent with him (code names "Four Roses" and "XKenneth").

I would also like to thank my thesis opponent, Lal George, who have already given me valuable comments on this thesis.

I am very grateful to all past and present members of the multi group, and the department as a whole, for providing such a stimulating research environment.

The administrative staff does a great job of keeping the department and its computers running. Thank you all. In particular a big thanks to Christer Carlsson for taking care of all those necessary things, and to Görgen Olofsson

for being so kind all the time. Thanks also to Hasse who did such a great job of printing this thesis (I hope).

When I first started as a PhD student, it didn't take long until I noticed Annika Aasa, a lovely young researcher. After a while I gave her a rose, and she gave me glögg. After that she has been a constant source of joy in my life. Eventually, she also gave me a wonderful son, Gustav. During the last year of my thesis work Annika has done an incredible job of taking care of me and Gustav. I would never have been able to finish without her help. And in addition to that she also drew many of the pictures in this thesis. Thanks Annika, I owe you forever! Or as she so gently put it herself: "tacka bara Annika, inga andra behöver tack".

A final thanks to my mother and late father for their support, although they never really understood what I did. I would like to dedicate this thesis to you.

Contents

List of Figures

Part I

Introduction

Chapter 1

Background and motivation

The purpose of this thesis is to investigate methods to improve the efficiency of lazy functional languages, from a *compiler back-end* perspective, i.e., using optimisations done rather late during the compilation process, on low level code.

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

Note that this should be seen from the perspective where lazy functional languages really have a potential of being *faster* than imperative languages. Many of the complications that arise during the optimisation of imperative code, often related to *side-effects* in some way, simply do not arise when implementing a pure functional language. Unfortunately, lazy functional languages have some other "features" that severely complicate their implementation, and lead to problems that do not arise at all for imperative languages.

The rest of this introduction is organised as follows. We will first very briefly introduce lazy functional languages and motivate why they are so nice from the programmer's point of view. After that we will describe a basic principle behind their implementation, called *graph reduction*. We will also give a short introduction to *register allocation*, since that is one of the most important optimisations discussed in this thesis. We conclude the introductory part with an overview of the rest of this thesis, and a summary of its main contributions.

1.1 Lazy functional languages

Functional languages in general, and *lazy* functional languages in particular, have long been praised for their expressive power and their semantic simplicity.

In a pure functional language, all computation is based on a single concept:

> the *evaluation* of *expressions* resulting in *values*.

Concepts such as a *state*, *assignments* and *side-effects* which form the base of imperative programming, do not exist in a pure functional language. This is not surprising since functional languages originate from the mathematical tradition of λ-*calculus* [Chu41] and *combinatory logic* [CF58]. From the programmer's point of view this results in a more *declarative* style of programming which has many advantages. To motivate this, we will list some of the features that can be found in a modern lazy functional language, like Haskell [Hud92].[1]

The most important feature of a functional language in our opinion is that functions are *first class values*, and can be passed around (as function arguments and results), and stored in data structures exactly as other values. The use of such functions, called *higher order functions*, provides for easy reuse of previously written code, and can even be used to define completely new *control structures* by making suitable abstractions [Hug89]. The use of *algebraic datatypes* combined with *pattern matching* make it very easy to handle complex data structures in a natural way. The *strong type system* and automatic *type inference* provide increased safety from programming errors (the type system can often catch many "bugs" already at compile time). *Parametric polymorphism* encourages the programmer to write general functions that can later be reused. *Automatic memory management* via *garbage collection* offers an enormous relief for the programmer, not having to worry about the allocation and deallocation of memory (something which is often very error-prone).

The above could be said about many higher order languages, but there are also useful features that exist mainly in lazy languages (or at least are more natural to use in a lazy language). The *non-strict evaluation* in itself increases the expressiveness of the language, since the programmer does not have to worry about the "order" in which computations are done or about "unnecessary" computations. Only computations whose result affect the final result of the program will be performed, and the implementation will automatically select a working order (if possible).

As an example, consider a typical *search problem*. They are usually solved by constructing selected parts of a *search tree* and at the same time use some *pruning* heuristic to avoid building unnecessary parts of the tree. In a lazy language this can instead be solved in two separate steps: first build the entire tree (which may well be of *infinite* size) and then apply a pruning function to cut off uninteresting parts. As a result we get a clean separation of concerns, and a more modular program. The implementation will build only the necessary parts of the tree without any special treatment, due to the lazy evaluation (see [Hug89] for more discussion around the beauty of lazy evaluation).

[1] It is outside the scope of this thesis to explain in detail what each of these features mean, instead we refer to some standard text on functional programming [Tho96, Bir98].

With all these features, lazy functional languages are in general much more abstract and "far from" the machine than traditional imperative programming languages (which are normally not much more than an abstract description of the hardware itself). Needless to say, this also make lazy functional languages more difficult to implement, at least on conventional hardware.

However, even though the high level features require special solutions, it is still interesting to relate to implementation issues in the "imperative world" because over the years a lot more work has been put into optimising imperative languages, and we should try to reuse as much as possible of that work.

1.2 Graph reduction

In the previous section we discussed some advantages that lazy functional languages offer the programmer. In this section we will instead discuss some requirements that this puts on an implementation, in particular issues related to the lazy evaluation.

As a start, we will give a short introduction to the technique that we use to implement the laziness, called *graph reduction*. The basic method was first introduced by Wadsworth [Wad71], and has since been refined in various ways by many others (see section 2.6.1 on page 59 for a brief discussion of the history and previous work in this area).

1.2.1 Theory

Since lazy functional languages are based on the λ-calculus, the most important reduction rule is the β-reduction rule, where substitution is used to reduce a function application. In the style of operational semantics [Plo81] we could write:

$$\frac{e_1 \rightarrow^* \lambda x.e}{e_1\ e_2 \rightarrow e\,[{}^{e_2}/_{x}]}$$

i.e., if "e_1" reduces to a lambda abstraction "$\lambda x.e$" (in zero or more steps), then the application "$e_1\ e_2$" reduces (in a single step) to "e" with "e_2" substituted for all free occurrences of "x". We say that the application forms a *redex*, something that can be reduced. The above rule encodes *normal order reduction*, which is required by a lazy (or non-strict) language, in contrast to *applicative order reduction* that is used by *strict* functional languages. The effect of normal order reduction is to *delay* the evaluation of function arguments until their values are needed, i.e., we do not evaluate "e_2" above before the substitution. In addition to that, in a *lazy* language we also require the same term to be evaluated at most once, i.e., if "x" above occurs more than once inside "e" then the evaluation of "e_2" must be *shared* between all those occurrences. This is also called *call-by-need*, as opposed to call *call-by-name* which does not require sharing but still

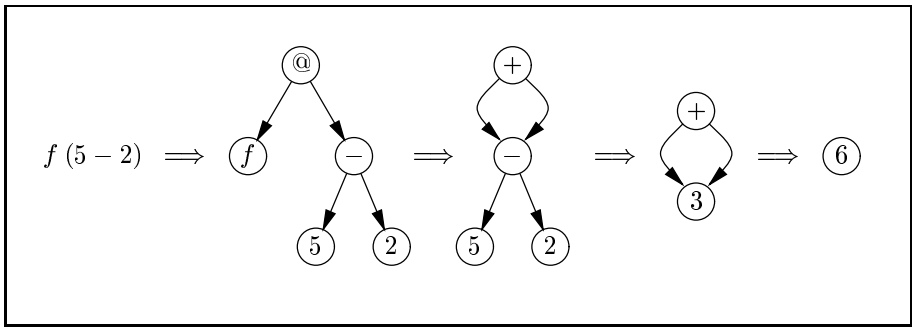

Figure 1.1: A simple example of graph reduction.

uses normal order reduction. Using the same terms, the strategy of a strict functional language would be called *call-by-value*, since all arguments will be fully evaluated (to values) prior to an application taking place (a function call).

The notion of *sharing* explains why we must use *graph reduction*, rather than *tree reduction*, to evaluate expressions in a lazy functional language. To illustrate this, consider a function "f" with the definition:

$$f = \lambda x.\,(x + x)$$

An example of how an application of "f" would be reduced in a lazy functional language is shown in figure 1.1. In the figure, "f" is applied to the expression "$5 - 2$" (the symbol "@" denotes application). When the application is reduced (the β-reduction rule shown above), the argument expression is substituted (by reference) for both occurrences of "x" inside the body of "f". Note how the argument expression is not evaluated until its value is needed by the addition, and how the result of the subtraction is shared (the computation "$5 - 2$" will only be performed once).

The successive expression graphs in the above example are all either *trees* or *directed acyclic graphs* (DAGs), but in general also *cyclic* graphs can appear during the evaluation.

The normal order reduction results in a *demand driven* evaluation, i.e., the order in which function applications are executed is fully determined by the *data dependencies* in the program and an initial demand for the final result of the program. Compare this to an imperative program where the execution is entirely driven by the control structures written by the programmer. We could also say that a lazy computation is *data driven* rather than *control driven*.

1.2.2 Implementation

Initial implementations of graph reduction were in the form of interpreters, and were not very fast. However, this changed dramatically with the arrival of

compiled graph reduction [Joh84]. The idea was that instead of having an interpreter perform the graph reduction, functions could be compiled to code that performed graph reduction when executed. This in itself did not change much, but suddenly it became possible to apply a number of optimisation techniques to the generated code, and this made a big difference. In particular, it became possible to sometimes generate code that did not perform the explicit graph reduction outlined above, but instead directly executed the operations involved. E.g., the function "f" in figure 1.1 need not build a graph for the addition, it can simply execute an addition instruction (once the values of the arguments have been found). However, when doing this we must be very careful not to evaluate "too much", we must still follow the non-strict evaluation order. So, in general the code produced will be a mix of "ordinary instructions" (for expressions that can be evaluated immediately, like the addition), and instructions to build and reduce graph (for expressions that must be built un-evaluated).

Since we can never entirely get rid of the graph manipulation, it is necessary that un-evaluated expressions (suspended function applications) can be represented in some way. We will use the term *closure* to denote an un-evaluated expression.[2] Correspondingly, there must be some way to turn a closure into a value, i.e., to *evaluate* the closure.[3] By *value* in this context we really mean a *weak head normal form* [PJ87], i.e., a term whose outermost "construction" can not be reduced any further (but can still contain redexes inside).

The typical example of closures and evaluation is the passing of function arguments. Since a *caller* (the function evaluating a function application) normally does not know if the *callee* (the function being applied) will need the values of its arguments, all arguments must be passed un-evaluated, i.e., as closures. Correspondingly, when a callee needs the value of some argument it must first make sure that it is evaluated, and not a closure. For this purpose, all implementations of graph reduction must include some kind of *evaluation procedure*, to turn a closure into a value.

Over the years a number of *abstract machines* and compilation techniques have been devised to efficiently implement graph reduction (an overview of this is given in section 2.6.1 on page 59). One of the most well known abstract machines is the *G-machine* [Joh84], and its associated abstract machine code, called *G-machine code*. To give a flavour of the code we will show how the function "f" used earlier could be written in G-machine code:

```
f :  PUSH 0
     EVAL
     PUSH 0
     ADD
     UPDATE 2
     RETURN 1
```

[2] Other commonly used terms for *closure* are *thunk*, *suspension* and *application node*.
[3] Other commonly used terms for *evaluate* are *force*, *enter* and *reduce*.

The G-machine is a stack based machine, so the two "PUSH" instructions will copy the top element on the stack (twice), to account for the two uses of "x" in the body of "f". The "EVAL" instruction is used to make sure that "x" is evaluated (if it was a closure). Before the function returns the "UPDATE" instruction will overwrite the original redex with the result of the function application (this is the sharing issue discussed above). Finally, the "RETURN" instruction pops one element off the stack and returns.

The evaluation procedure

The method used to force the evaluation of a closure is one of the most important parts of an implementation of graph reduction. Exactly how it is implemented depends of course on how the closures themselves are implemented. Most implementations will include some kind of "pointer" to the function in the representation of a closure, so that the correct function can easily be called when a closure needs to be evaluated. Note that the function to call is in general *unknown* at compile time. So, normally the evaluation of a closure will result in some kind of *indirect call* (through a pointer).

Unfortunately, the presence of such unknown function calls in the code makes some kinds of optimisations much more difficult, optimisations that rely on the *target* of function calls to be known at compile time. Because of this, our implementation will use a novel approach to the handling of unknown calls. We simply *forbid* all unknown calls (in the implementation, not in the original source code). Instead, we use a *control flow analysis* to approximate "possible targets" for all unknown calls resulting from the evaluation of closures, and then replace each indirect call with an explicit test combined with a number of calls to known functions. These ideas were originated by Thomas Johnsson [Joh91], and will be discussed extensively later in this thesis.

More complications

The handling of the lazy evaluation is perhaps the main obstacle when implementing a lazy functional language. But there are also other things that make an implementation more cumbersome compared to implementing an imperative language. We will give a few more examples here. One particular problem has to do with the creation of function closures when *free variables* are involved, i.e., variables that are referred to from within a lambda abstraction but are not parameters of it. The problem is basically that a closure in general can *survive* the function invocation that created the closure. This means that when a closure is evaluated and a reference is made to one of its free variables it is possible that the value of the variable no longer exists (because it was stored in an activation record that no longer exists). To solve this dilemma we must either arrange closures so that they can always reference the values of free variables or, which is our solution, eliminate free variables before building closures. This

can be done by turning all free variables into extra arguments of the closure being built, a technique called *lambda lifting* [Joh85]. In essence, the program is transformed into a set of *super combinator* definitions [Hug82].

Higher order functions also pose special problems for an implementation. In fact, some of the problems due to higher order functions are very similar to the laziness issues discussed above, and requires similar implementation techniques. This means that also a *strict* functional language will have to deal with closures (if the language allows the use of higher order functions).

The *polymorphic type system* combined with *garbage collection* requires some special solutions concerning data representation. Consider a function whose first argument is of a polymorphic type, i.e., at runtime any type can appear as that argument. If a garbage collection occurs while an invocation of the function is active, the garbage collector must be able to "check" if the first argument is a pointer or not (if it is a pointer the object it points to must also be examined). A number of solutions to this problem exists, each with both advantages and disadvantages, e.g., extensive *function specialisation* (not very common), *tagged pointers* (common among implementations of strict functional languages), and *uniform data representation* (common among implementations of lazy functional languages). The latter, which is also our main solution, requires all polymorphic function arguments to be represented in a uniform way, normally as *boxed* values. Unfortunately this means that even such "simple" values like integers must be represented as pointers to objects in memory. However, this concerns only values passed as polymorphic function arguments or built into polymorphic data structures. For values of known types it is possible to use other data representations, like *unboxed* values.

Opportunities

Above we discussed some things that make it *harder* to implement a lazy functional language, but there are also properties that make it much *simpler*!

Many of the nice features of a pure functional language is a direct consequence of the clean semantics. Perhaps the most notable property is the absence of *side-effects*, which is manifested in a number of ways. The notion of *referential transparency* enables the use of a number of *program transformations*, that can be used both in the front-end and in the back-end of a compiler. Another aspect is that there are no *global variables* whose value can suddenly change.

The optimisation of imperative languages can often be severely hindered by unknown *alias* situations (several pointers to the same memory location). Aliasing occurs also in an implementation of a lazy functional language, but in a much less intrusive way (in some sense the aliasing is under control by the compiler rather than by the programmer, which obviously makes it easier for the compiler to handle).

In a pure functional language there are no assignable variables, instead variables are no more than names on values, and they can never change. This is

sometimes referred to as *static single assignment* (SSA), and it is a property that makes many optimisations much simpler. In fact, a popular recent trend among implementors of imperative languages has been to first "transform" the imperative code into a functional representation, typically SSA, and then do optimisations on that. This if anything should be proof that purely functional code is well suited for optimisation.

1.3 Register allocation

Since one of the most important optimisations done in our back-end is *register allocation*, we will also give a short introduction to that area. A more detailed discussion of previous work is given in section 6.4 on page 238. We will also try to motivate why we believe it to be particularly important to do *interprocedural* register allocation when implementing lazy functional languages.

A common way to characterise register allocation methods (and other optimisations) is by considering how large parts of the program the register allocator will optimise at a single time. This yields three different kinds, in increasing power: *local*, *global* and *interprocedural* register allocation. The former two are also called *intraprocedural* methods.

1.3.1 Local register allocation

The simplest form of register allocation that can be done is termed local in the sense that it is only done on a small *expression tree* or inside a *basic block* at once. By a basic block we mean a piece of code with a single entry point and a single exit point, i.e., usually a very small piece of code (typically 5–10 instructions). Local register allocation is usually not done as a standalone optimisation, but instead built into the normal code generation. Often it is done as follows: when generating code, keep track of the register usage with some kind of *register descriptor*. When a new register is needed and there are no free ones, choose an occupied register, free it (by saving its value to memory), and use the register for the new value. A simple heuristic is used to select which register to free, e.g., to free the register whose *next use* is the most distant. When the end of a basic block is reached, all values in registers must be saved to memory. Since basic blocks are often very short, local methods can normally not keep values in registers for a particularly long time.

1.3.2 Global register allocation

By global register allocation we mean "across basic block boundaries", often on an entire procedure at once. To some extent, the simple method outlined above can also be used across basic block boundaries. When control paths join we somehow combine the different register descriptors. Unfortunately, such

a method will not be very powerful, e.g., it can not allocate a variable to a register over a loop. To see why, consider how a loop is compiled; typically it has a backward branch at the end, so when compiling the beginning of the loop, we should agree on register usage with the end of the loop. But, since we have not yet compiled the loop we know nothing about the register usage at the end of it and we have to make the most conservative assumption, that all variables are kept in memory, not in registers, at that point.

An alternative method is to use so-called *usage counts* [WJW+75]. Here, we approximate the savings of keeping each variable in a register instead of in memory and then choose the most beneficial ones to be allocated to registers. E.g., a variable that is used inside several nested loops is probably a good choice to keep in a register. This means that in addition to simply counting the number of *uses* for each variable, some notion of *weights* is needed to accurately approximate the benefits.

As the techniques become more advanced it is common to separate the problems of code generation and register allocation using the following assumption (this separation of concerns was advocated by the influential PL.8 compiler [AH82]):

> *During code generation (and optimisation) assume an infinite number of registers. Treat register allocation as a separate problem.*

We will use the term *virtual registers* to describe this infinite number of registers. The region of the code in which a virtual register contains a value, is called a *live range*. The task of the register allocator then becomes to decide, for each live range, if it is going to be mapped to a real machine register or if the value is going to be kept in memory instead (we say that the value is *spilled*).[4]

Graph colouring

A clever but yet simple abstraction that can be used for the problem of mapping virtual registers to machine registers is *graph colouring*. An *interference graph* (also known as conflict graph) is constructed where nodes symbolise virtual registers and an edge between two nodes mean that the two virtual registers are simultaneously *live*[5] and therefore can not be mapped to the same machine register.

The problem of register allocation has now been transformed into the problem of giving a colouring of the interference graph, using the same number of colours as there are available machine registers, such that no adjacent nodes receive the same colour.

[4] The register allocation problem itself is sometimes separated into two distinct subproblems, called register *allocation* and register *assignment* [ASU86, section 9.1]. However, that separation is not very meaningful in the kind of register allocation algorithms that we are interested in so we will not use that terminology.

[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.

Most formulations of graph colouring problems are NP-complete [GJ79], e.g., determining the *chromatic number* of a given graph (the minimum number of colours required to colour it). Another variant that is as hard, although it may seem slightly easier, is to determine if a given graph can be coloured using N colours (we use the letter N to denote the number of available colours, i.e., machine registers). Moreover, the problem size is often large (many variables), so finding an optimal solution is usually too expensive, we will have to satisfy with a sub-optimal one. Therefore, it is necessary to find good *heuristics* for the colouring.

Chaitin-style colouring

The first graph colouring register allocator, the classical "Yorktown allocator" was implemented by Gregory Chaitin and his colleagues at IBM and described by Chaitin *et al.* in [CAC+81] and Chaitin [Cha82]. Chaitin's innovation was to use a simple heuristic to reduce the complexity of the interference graph. The key observation was that a node in the interference graph with less than N neighbours is trivially colourable, if we can only colour the other nodes in the graph, since there must be at least one colour that is not taken by the neighbours. This observation leads to the following algorithm:

1. Remove all nodes with less than N neighbours from the graph (and all their interference edges), and try to colour the rest of the graph.
2. Repeat 1. as long as there are still nodes with less than N neighbours.
3. If the graph becomes empty we are done, and we can assign actual colours to all nodes by reinserting them into the graph in the *reverse order* (compared to how they were removed from the graph). For each node we select a colour that has not yet been assigned to any of its neighbours. Note that this is guaranteed to succeed due to the condition used in 1.
4. If the graph is not empty it must mean that all remaining nodes have at least N neighbours. When that happens a node is chosen to be *spilled*, i.e., the value is kept in memory instead of in a register. We remove the node from the graph and continue the colouring, i.e., go back to 1. When the graph is empty we rebuild it without the spilled nodes and start the colouring from the beginning. Hopefully the reduced complexity of the new graph will make the colouring succeed.

This simple heuristic was described as early as 1879, in the context of the geographical four colour problem [Kem79].

The main phases of Chaitin's original allocator, the so-called *build-colour* cycle, are shown in figure 1.2 on the facing page. We show this here because as one of the contributions of this thesis we will later show a slight variation to this. The *renumber* phase is used to find all live ranges. Note that a live range is

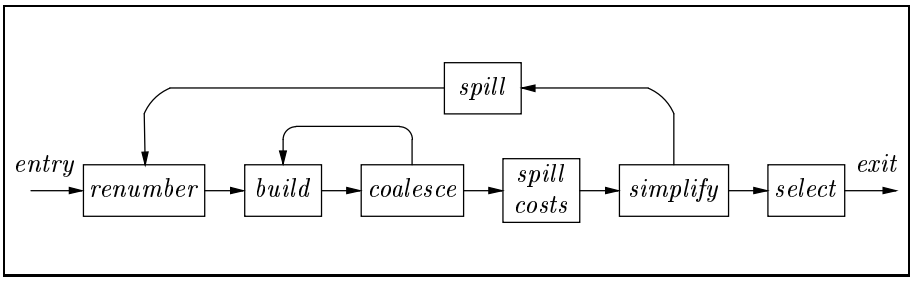

Figure 1.2: Chaitin's *build-colour* cycle.

not necessarily the same as a variable name because the programmer may have reused the same variable name for several completely distinct values. So, what we need is to identify so-called *def-use chains*. Chaitin referred to this as getting "the right number of names". The *build* phase builds the interference graph, by first calculating *liveness* data flow information. The *coalesce* phase eliminates unnecessary register to register copy instructions, a kind of *copy propagation*. The purpose of the *spill costs* phase is to estimate, for each live range, the cost of spilling that live range. The *simplify* phase encodes the colouring heuristic mentioned above. It will remove nodes from the graph until all nodes have either been found trivial or have been spilled. Depending on if any spills were found, we continue either to the *spill* phase, insert spill code and restart the allocation, or if no spills were found we continue to the *select* phase. The *select* phase will simply assign real colours to all nodes, in the reverse order as they were removed by *simplify*. Note that *select* is guaranteed to find a colour for each node due to the heuristic used by *simplify*.

Procedure calls

Although global allocators can be quite successful in allocating heavily used variables inside loops to registers, they are not particularly good at handling procedure calls. Normally a global allocator will have to save and later restore all live local variables around each call site. This can be alleviated somewhat by using a clever *linkage convention* (also called *calling convention*). Registers are classified as either *caller-saves* or *callee-saves*, depending on who is responsible for saving the register. A caller-saves register is not guaranteed to be left undisturbed by a function call, so if a value needs to survive a call, it must be saved by the caller. On the other hand, a callee-saves register is guaranteed to survive a function call, and hence must be saved by a callee before it can be used. A split convention, with some caller-saved and some callee-saved registers is usually the most effective, but is also more difficult to handle for the compiler than to use a single convention for all registers.

Even though a good linkage convention can somewhat reduce the procedure-call overhead, it is not enough for programs with very high procedure-call frequency. More aggressive methods are needed. Also, per-procedure allocation is often not able to use the large number of registers available with modern processors. This is especially true for functional languages, since procedures in functional programs tend to be much smaller than in imperative programs.

1.3.3 Interprocedural register allocation

In principle, any algorithm that uses information about the register usage of other procedures is an interprocedural algorithm. Of course, rules that are dictated by the standard calling convention does not count in this respect. This means that a global register allocator with a clever division of the registers into caller-saves and callee-saves classes does not qualify as an interprocedural algorithm (because the division of the registers into classes is fixed, it does not depend on the procedures involved). Most previous attempts at real interprocedural register allocation instead falls into the following two categories:

- *per-procedure* register allocation, but using interprocedural (program-wide) information to reduce the call and return overhead,
- *program-wide* register allocation, done for all variables in the program at once.

Some "key problems" for interprocedural register allocators are: recursion, indirect procedure calls, global variables, efficiency (of the allocation) and the interaction with separate compilation.

Per-procedure bottom-up register allocation

A technique called *bottom-up* register allocation is an example of the former of the two above classes, i.e., the allocation is still done on one procedure at a time. The key observation is that procedures that cannot be active at the same time can use the same registers for local variables. Therefore, if we allocate registers for one procedure at a time in a bottom-up traversal of the *procedure call graph*, and avoid using registers used by descendant procedures we do not need to save and restore registers around procedure calls. The main motivation for a bottom-up algorithm is the observation that many programs "spend most of their time in the bottom of the call graph" (observed for Lisp programs in [SH89]).

Recursive calls can be handled by taking the *strongly connected components* [AHU85, section 5.5] of the procedure call graph. For calls inside the same component, i.e., possibly recursive calls, all local variables that are live across the call must be saved and restored around the call site. For calls outside a component, the normal bottom-up allocation is used.

For calls where the callee are either unknown or not available during the allocation, e.g., indirect calls or calls to procedures in other modules, a standard linkage convention is used, guaranteeing no more than a global allocator with caller-saves and callee-saves registers. This means that all live registers for all *ancestors* in the call graph must be saved before doing such a call. Note that such liveness information may not be available if the allocation is required to be done in a single bottom-up traversal of the procedure call graph, which is often the case, so we may need to approximate what registers to save.

Normally, bottom-up allocation can not be applied to global variables since that does not really fit well into their per-procedure compilation strategy.

Program-wide register allocation

Compilers with real program-wide allocators are rare, mostly because of practical problems, e.g., the interaction with separate compilation. Wall describes an algorithm for register allocation on all variables in the program at once [Wal86]. The real register allocation is done at *link-time*, and hence the allocation at compile time is very limited, mostly consisting of annotations describing how the code should be changed if a variable gets allocated to a register. Estimated *usage frequencies* are used to decide what variables should be allocated to registers. Similarly to bottom-up allocators, Wall uses the observation that procedures not active at the same time can use the same registers for their local variables. Since Wall has program-wide information available he is also able to allocate heavily used global variables to registers.

There are also other approaches to interprocedural register allocation that do not really fit into the classification above. See section 6.4 on page 238 for a more detailed discussion of this and other related work.

1.3.4 Lazy functional languages and register allocation

One of the main claims of this thesis is that *interprocedural* register allocation is particularly beneficial for lazy functional languages. There is really only one reason for this, the high *function call intensity* of lazy functional languages, which makes it essential that function calls are cheap. But this is exactly the purpose of interprocedural register allocation, to *minimise the function call penalty* [Cho88].

For comparison, Steenkiste measured a set of Lisp programs and found that on average only 11 instructions were executed between two consecutive function calls or returns [SH89]. We have no reason to believe the situation to be any better in our context. In fact, the situation may be even more pressing in a lazy language due to the large amount of function calls that result from the evaluation of closures, in addition to all "normal" function calls. With this in mind, it is not difficult to see that a global register allocation algorithm will fall short, and that some interprocedural approach is needed. The function call

overhead needs to be reduced in two ways. First, we need to avoid the saving and restoring of live registers around function calls that a global allocator normally must do, and second, we must also minimise the overhead of handling function arguments and results (which preferably should be passed in registers).

From a more pragmatic angle, it is interesting to examine the concept of *loops* in imperative and functional languages. For imperative languages, the most important parts to optimise are the so-called *inner loops*. For pure functional languages there are normally no explicit looping construct in the language.[6] Instead all "looping" is done using recursion, i.e., function calls. For *lazy* functional languages it gets even worse, many of the recursive loops produce so-called lazy data structures, e.g., a lazily constructed list. This means that not all loop iterations will happen at the same time. Instead, one "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.

All these issues will be discussed extensively throughout this thesis.

1.4 Overview of the thesis

This section will give an overview of the organisation of the rest of this thesis. Since the thesis in large follows the organisation of the compiler back-end we have implemented, we will start by explaining the main infrastructure of the back-end (and the compiler of which the back-end is a part).

1.4.1 Compiler framework

To support our research we have implemented a complete compiler back-end for a lazy functional language, called *the GRIN back-end*. All optimisations discussed in this thesis have been implemented in the GRIN back-end (with a few minor exceptions). To make the experimentation easier we have connected our back-end to an already existing Haskell front-end. The main organisation of this combined compiler is shown in figure 1.3 on the facing page.

The front-end uses standard separate compilation and is modified to output the code resulting from each Haskell module to an intermediate file. The back-end (which is a stand-alone program) will collect this code (called λ in the figure) from all modules that are part of the program, and optimise the entire program at once. In other words, the combined compiler offers separate compilation in the front-end and program-wide interprocedural optimisation in the back-end.

[6] The special case of tail recursive calls (to the same function) corresponds to a real loop, but unfortunately most recursion is not that simple.

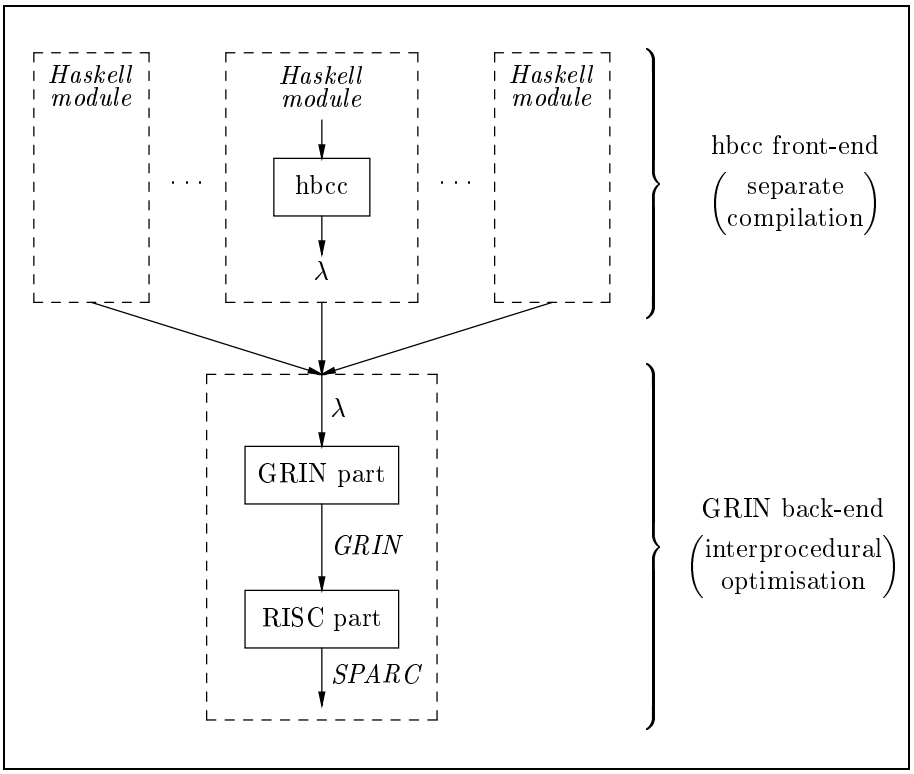

Figure 1.3: Overview of the compiler.

The hbcc front-end

The Haskell front-end we use is written by Lennart Augustsson [unpublished]. It was written in an effort to create the next generation of the well known Chalmers Lazy ML and Haskell compiler [AJ89c], but the effort was discontinued before the compiler was finished. The front-end is quite usable though, and a fair characterisation of it would probably be "a state of the art a couple of years ago" Haskell front-end.

Using hbcc we get well optimised code in a "low level functional" language comparable to the *Core* language [PJ96] used by the Glasgow Haskell Compiler. The code is lambda lifted, i.e., the program consists of a set of super combinator definitions, and all "high level" Haskell constructs have been transformed away. E.g., all uses of Haskell style overloading, via type classes, have been transformed into explicit dictionary passing [WB89, Aug93].

Front-end issues are not really part of this thesis, and from now on we will not discuss hbcc any more.

The GRIN back-end

The back-end consists of two major parts. The first part is built around an intermediate code, called GRIN, and will gradually transform and optimise the code into a very simple form (but still GRIN). After that the code is translated into machine code for a hypothetical RISC machine. The second part of the back-end operates on this RISC code and applies a number of low-level optimisations, in particular an interprocedural register allocation algorithm.

1.4.2 The GRIN part of the thesis

Chapter 2. The GRIN language is introduced in chapter 2. The name GRIN is short for *Graph Reduction Intermediate Notation*, and it is the intermediate language that we use to implement graph reduction. The GRIN language is intended as a further development of the well known G-machine [Joh84], providing a more functional intermediate code, better suited for analysis and program transformation, while still retaining a low level control that can be utilised for various optimisations. A built-in monad is used to express sequencing and side-effects.

We define the language by giving its syntax and semantics, and continue with an explanation of how the language can be used to implement a lazy functional language (via graph reduction). E.g., we explain how to handle closures and evaluation, sharing and updates, and higher order functions. We also give a translation from the simple functional language that is the input to the back-end (called λ in figure 1.3), to GRIN.

Chapter 3. This chapter explains a *control flow analysis* done on GRIN code, which forms a large part of the foundations for our work on GRIN. The intention is to approximate the evaluation order of the program in a way that allows us to later eliminate all "unknown function calls" from the program (as discussed earlier in this introduction). This analysis as well as the original ideas of eliminating unknown function calls is due to Thomas Johnsson [Joh91].

Chapter 4. This contains the major part of the GRIN description. We describe a large number of GRIN source-to-source program transformations, used to simplify and optimise the code. The transformations are divided into two categories due to their slightly different nature: *simplifying* and *optimising* transformations, and we describe each category separately.

The simplifying transformations taken together create an overall "compilation by transformation" strategy. The program is gradually transformed from "high level" GRIN to "low level" GRIN code. Each simplifying transformation is only applied once.

The optimising transformations on the other hand are simply intended to make the code faster. Many of these are similar to optimisations found in

other implementations (of both functional and other languages), whereas other transformations are more novel, and made possible due to special properties of the GRIN language. The optimising transformations are repeatedly applied until the program does not change any more.

The single most important of all GRIN transformations is the elimination of unknown control flow using the result of the control flow analysis discussed in chapter 3. This transformation is called *eval inlining*.

A complete GRIN transformation example is given in appendix A.

1.4.3 The RISC part of the thesis

Chapter 5. After all GRIN transformations the next step is to translate to RISC code, a machine code for a hypothetical RISC processor. Our RISC code is very similar to SPARC code, so the chapter also includes a short introduction to the SPARC architecture. To be able to illustrate all aspects of the RISC code generation we must also explain a set of low level conventions, commonly known as runtime system issues. This includes things such as rules for register usage, heap and stack layout, etc.

Chapter 6. We describe two register allocation algorithms, which we call the *base* algorithm and the *extended* algorithm. The base algorithm, which is what is implemented in the back-end, is a program-wide graph colouring algorithm. The algorithm is designed specifically to reduce function call and return overhead.

The *extended* algorithm is a detailed proposal of making a more practically useful allocator, by combining the base algorithm with a *bottom-up* register allocation algorithm (discussed earlier in this introduction).

Chapter 7. This chapter discusses optimisations done on the RISC level, except for the register allocation. These are mostly standard low-level optimisations, and include both optimisations that are particularly important for functional languages, like the optimisation of heap allocation and tailcalls, and more general optimisations, like instruction scheduling.

Chapter 8. We devote a complete chapter to the discussion of issues related to garbage collection. The problem is that supporting garbage collection becomes non-trivial due to the kind of aggressive optimisations done by the GRIN back-end, in particular the register allocation. The solution we describe involves a combination of compile time and runtime support. The compiler uses various analysis at a few strategic points during the compilation to make sure that enough information about pointers and non-pointers is found (such information is needed for accurate garbage collection). The information is then communicated to the runtime garbage collector via small descriptor tables that are put

in the final code. These tables are used at runtime to traverse the system call stack.

Chapter 9. To prove the validity of our approach we include a chapter with experimental results. We compare the code produced by the GRIN back-end with that produced by some other Haskell compilers, and show that our code executes several times faster. We also examine how various GRIN transformations affect performance, and make some experiments with the register allocator.

Chapter 10. The final chapter summarises our conclusions, and discusses some issues that need further research.

Appendix A. A complete GRIN transformation example, for a small example program is given in appendix A.

1.5 Summary of research contributions

The main contributions of this thesis are the following:

1. The GRIN *language*. As an intermediate language in a compiler for a lazy functional language it provides a very convenient "middle way" between a high level functional code and a low level imperative code. This makes GRIN well suited for analysis and program transformation, while still providing the low level machinery necessary to express many concrete implementation details.

2. The use of a *control flow analysis* together with a simple GRIN program transformation to approximate the control flow of the program in a way that makes it possible to eliminate all function calls to (compile time) *unknown functions*. Note that both the analysis and the original idea to eliminate unknown function calls are due to Thomas Johnsson [Joh91], we have merely made practical use of this in a compiler.

3. The *program transformation* strategy consisting of a large number of GRIN source-to-source program transformations. Most transformations are very simple, but when taken together, and applied repeatedly, they produce greatly simplified and optimised code. Some of the transformations are particularly interesting, most notably the *generalised unboxing*, the *late inlining*, the *case hoisting* and the *arity raising*. Taken together these transformations can sometimes give effects similar to *deforestation* [Wad88]. For an example of this see appendix A.

4. The *interprocedural register allocation algorithm*. It is used to reduce function call and return overhead in two ways. First, it avoids much of the saving and restoring of registers that is normally required around function calls, and second, it enables a very efficient use of registers to pass function arguments and results, constructing a tailor-made calling convention for each function. We develop a technique similar to *shrink-wrapping* to optimise the placement of register save and restore instructions, and extends previous graph colouring algorithms with *interprocedural coalescing* and a restricted form of *live range splitting*.

5. The combined compile time and runtime approach to support garbage collection in the presence of aggressive optimisations (most notably our register allocation), without imposing any mutator overhead. The method includes a runtime call stack traversal and interpretation of registers and stack frames using pre-computed descriptor tables.

6. All the above taken together to create a complete back-end for a lazy functional language, providing support for program-wide interprocedural optimisation, and producing code that executes several times faster than code produced by other compilers.

Part II

GRIN
the intermediate code

Chapter 2

Graph Reduction Intermediate Notation

In this chapter we will introduce the GRIN language, the language used as an intermediate code in the GRIN back-end. We define the language by giving its syntax (section 2.3 on page 27) and an operational semantics (section 2.4 on page 35). To describe how GRIN can be used to implement graph reduction we continue with a section "Compiling to GRIN" (section 2.5 on page 43), where we also show examples of GRIN programs. The main features of GRIN can be summarised as:

- it is a functional language, well suited for program transformation,
- it offers low level control over all data (closures are "normal" values),
- it offers "storage control", i.e., memory vs. registers,
- a built-in monad captures side-effects and sequencing.

2.1 Background

The name GRIN itself is short for *Graph Reduction Intermediate Notation* and was introduced by Thomas Johnsson in [Joh91]. When first presented, GRIN was very much like the well known G-machine code [Joh84]. It was an imperative language where statements inside procedure bodies were essentially three address code [ASU86, section 8.1]. GRIN was mainly intended to implement languages based on graph reduction (like Haskell), but was in fact quite a general form of intermediate code. It could be a suitable intermediate language for many "heap based" languages (like Lisp or Standard ML).

The main difference between GRIN and the original G-machine code was that GRIN used explicit names for arguments and local variables (temporaries)

instead of implicit "stack offsets" in the abstract machine stack. In other words, GRIN had taken the step from a stack based execution model to a more conventional register based model. One of the reasons for this change was that Johnsson wanted to try more advanced register allocation techniques than had previously been used on G-machine code [Joh86]. This made it natural to introduce explicit names, to make register candidates "visible".

A novel invention of GRIN was that it was possible to express parts of the runtime system in the GRIN language itself, and thereby making these parts available to the GRIN optimiser. The example given was EVAL, which in the G-machine is the built-in instruction used to force the evaluation of a suspended computation, i.e., to turn a closure into a value (or really, a *weak head normal form*). This will be discussed in detail later (see section 2.5.2 on page 44).

2.2 The GRIN language

To make it easier to reason about intermediate code and to apply program transformation techniques, we have developed a "more functional" version of GRIN. To summarise: our GRIN programs are *state monadic*, *first order*, *strict*, *untyped*, *functional programs*. Combining the two notions of monadic and first order may seem a bit contradictory, since monads are very closely coupled to higher order functions, but it still makes sense in GRIN (the GRIN language itself is first order, but it has a built-in monad that can only be used in a very restricted way, see below).

When compiling lazy functional languages, sooner or later one has to confront the issue of updating due to *call-by-need*. We have chosen to make this explicit in the GRIN language, and as a consequence, GRIN is in some sense not a pure functional language. The update operation will have to actually overwrite an already existing heap cell with a new value. However, it turns out that we can hide this, and all other non-pure features, inside a *state monad*, and thereby make GRIN at least "on the surface" a pure functional language. For general descriptions on the use of *monads*, see [Wad92] and [PJW93].

Another important feature that we wanted to keep from the original imperative GRIN code was the explicit sequencing of operations. As it happens, by introducing a state monad to hide side-effects, we also solve the sequencing problem, since state monads have sequencing built-in.

2.2.1 The GRIN built-in monad

The GRIN built-in monad, a specialised "graph reduction monad", is defined by the following (in the style of Wadler [Wad92]):

- a monad type M,
- a monad *bind* operator, written ";",

- a monad *unit* operation, written "unit",
- a set of special "graph reduction operations" used to manipulate heap objects (fetch, store and update).

The state monad type M is abstract, which means that there is no way for a GRIN programmer to access the state, other than by using the built-in monad operations. The state is basically an abstraction of the heap memory, which together with the graph reduction operations is supposed to mimic what would happen in a real implementation of graph reduction.

It is actually very easy to implement the GRIN monad in a functional language (like Haskell), and then run GRIN programs as Haskell programs with only minor syntactic changes. Of course, this will not be very efficient, but it will work! In a real implementation, though, the idea is that the monad itself should basically disappear. Since the monad is used only to capture side effects it is not surprising that it is no longer needed at the level of "real machine code". The monad heap will map directly onto the normal heap memory and the sequencing in GRIN will become ordinary sequencing between machine instructions (the code generation from GRIN to RISC machine code is described in chapter 5).

2.3 Syntax

The GRIN syntax is given by the grammar in figure 2.1. While studying the formal rules, some readers may want to glance at some example code. For this we recommend the GRIN program in figure 2.8 on page 45 (which is the GRIN version of the small functional program in figure 2.7 on page 44).

A GRIN program is a set of function definitions (which we often call procedures) one of which must be called *main*. GRIN functions are essentially *super combinators* [Hug82], i.e., all GRIN functions are defined on the "top level" and a function body may not contain any free variables, other than the names of (global) functions or values. Function bodies are GRIN expressions (called *exp* in the syntax).

2.3.1 Sequencing operations

The basic construction for building GRIN expressions is the monad *bind* operator (";"), used to sequence GRIN operations:

$$operation\ ;\ \lambda\, pattern \rightarrow rest$$

A GRIN expression can be seen as a *sequence* of possibly side-effecting operations, going from left to right, i.e., the left hand side of a bind operator is "performed" before the right hand side. The possible GRIN operations are

prog	::=	$\{\ binding\ \}^+$	program
binding	::=	$var\ \{\ var\ \}^+\ =\ exp$	function definition
exp	::=	$sexp\ ;\ \lambda\, lpat \rightarrow exp$	sequencing
	|	case *val* of $\{\ cpat\ \rightarrow\ exp\ \}^+$	case
	|	*sexp*	operation
sexp	::=	$var\ \{\ sval\ \}^+$	application
	|	unit *val*	return value
	|	store *val*	allocate new heap node
	|	fetch *var* $\{[\,n\,]\}$	load heap node
	|	update *var val*	overwrite heap node
	|	$(\ exp\ )$	
val	::=	$(tag\ \{\ sval\ \}^*)$	complete node (constant tag)
		$(var\ \{\ sval\ \}^*)$	complete node (variable tag)
	|	*tag*	single tag
	|	()	empty
	|	*sval*	simple value
sval	::=	*literal*	constant, basic value
	|	*var*	variable
lpat	::=	*val*	lambda pattern
cpat	::=	$(tag\ \{\ var\ \}^*)$	constant node pattern
	|	*tag*	constant tag pattern
	|	*literal*	constant

$\{\ \ldots\ \}$	means 0 or 1 time
$\{\ \ldots\ \}^*$	means 0 or more times
$\{\ \ldots\ \}^+$	means 1 or more times

Figure 2.1: Syntax for the GRIN language.

called *sexp* (simple expressions) in figure 2.1. Above, "*operation*" will first be performed and the value it returns bound to "*pattern*". After that we will continue with the operations in "*rest*".

It is important to understand that this monadic structure (the bind operator followed by a lambda) is "hard coded" in the GRIN grammar (in figure 2.1), which explains the previous claim that GRIN is a first order language. It is *not* possible to use arbitrary lambda abstractions in GRIN, all lambdas must follow the hard coded structure above. The lambda in GRIN is only used to introduce bindings, not to introduce arbitrary functions.

An alternative way to represent monadic code would have been to use a traditional expression grammar (allowing arbitrary lambda abstractions), and then treat ";" as a normal *infix operator*. By putting a lambda abstraction to the right right of a bind we could represent the same kind of structures as above. However, we really want to force *all* GRIN expression to be of the simple first order form: "operation - bind - lambda", and have therefore decided to treat that as a single syntactic construct. Without that we would have been forced to specify additional rules for how valid GRIN expressions could be built. We do not want to allow arbitrary expressions, since GRIN is intended to be a "simple" (read low-level) language.

Furthermore, we emphasise the functional nature of GRIN code by naming the main non-terminal in the grammar *exp* (as in expression). Some might claim that *statement* would be a more suitable name, given the imperative nature of "performing operations in sequence". Different people tend to have very different opinions when it comes to the issue of whether (state-) monadic functional code really should be called "functional" or "imperative". We prefer to call GRIN a functional language.

When writing larger GRIN expressions we will usually write one operation per line:

$$
\begin{array}{l}
\mathit{operation}_1\ ;\ \lambda\, \mathit{pattern}_1 \rightarrow \\
\mathit{operation}_2\ ;\ \lambda\, \mathit{pattern}_2 \rightarrow \\
\mathit{operation}_3\ ;\ \lambda\, \mathit{pattern}_3 \rightarrow \\
\mathit{rest}
\end{array}
$$

The bind operator (";") is *right associative*, so the syntactic structure of the above becomes *right skewed* as shown in figure 2.2 on the next page. A different structure can of course also be useful sometimes, and can be achieved using parentheses. Note how the bind operator together with a lambda is shown as a "single node" in the syntax tree, to emphasise that they really are a single syntactic construct.

The associativity of the bind operator is further discussed in one of the program transformations, the *bind normalisation* (see section 4.4.1 on page 156).

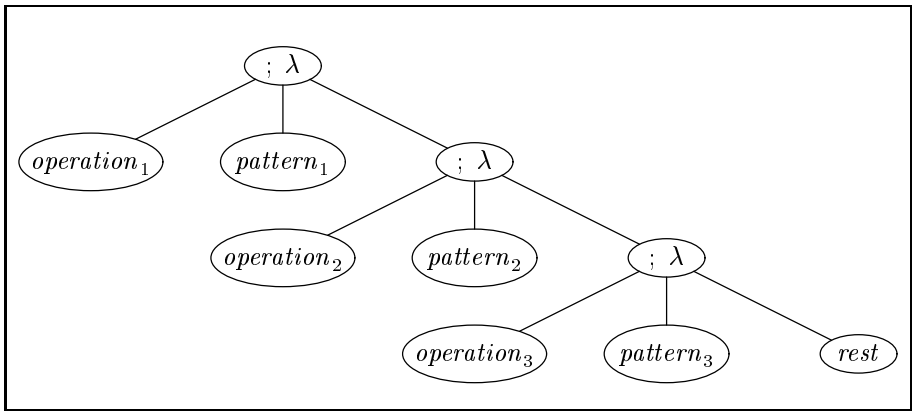

Figure 2.2: Sequencing of GRIN operations using *bind* (";").

2.3.2 Control flow

Besides the bind operator, the other structuring tool for GRIN is *case* expressions. These can be used to change the *control flow* inside a GRIN expression. A case expression can be put both to the left and to the right of a bind operator, which means that it can be used to both *split* and *join* the control flow, albeit in quite a restricted way. As a consequence, when generating real machine code from GRIN expressions (see chapter 5), we will always end up with *flow graphs* that are *directed acyclic graphs* (DAGs). When a case expression is put to the right of a bind operator, i.e., it is put "last" in a GRIN expression, it will only result in a flow graph that is a tree. But if a case expression is put to the left of a bind (we call this an "internal case") it will result in a flow graph that is a true DAG. These properties are further discussed in section 5.3.3 on page 192. The "binds" and the "cases" form what we call the *spine* of a GRIN expression. See also the discussion around *static single assignment* (SSA) in section 2.3.7 on page 34.

2.3.3 Nodes and tags

To fully understand GRIN it is essential to understand our notions of *node value* (or just *node*) and *tag*. A node is a *sequence* of one or more GRIN values, where the first is always a tag (used to determine what kind of a node it is) and the rest are either pointers (into the heap) or basic values (like unboxed integers). The tag itself is a value that can be *inspected* just like any other value (by a case expression).

In this respect our notion of "tag" is not the same thing as what is normally called a *value constructor* in the context of *algebraic datatypes*. Such a con-

structor is usually viewed as a *function* from its arguments to a value of the corresponding type. Normally, a constructor can not by itself be "inspected" within the language, it has to be *fully saturated* before you can see what the actual constructor is that builds up a value (since there is no way to inspect functions). In GRIN the concepts of tag and node are more closely modelled after a concrete implementation. We can easily inspect a tag by itself, or any partial application of a tag for that matter.

An example of a GRIN node value is:

$$(\mathsf{CCons}\ x\ xs)$$

Here, a sequence of a CCons tag and two variables denotes a list *cons* node. Note that this syntax may be slightly misleading. It may look as if the tag is being "applied" to the rest of the values of the node, but that is just a convenient syntax for writing entire node values. For simplicity we also use a similar terminology and say that the tag is "applied" to its "arguments". The tag arguments (or "node arguments") are defined to be the sequence of values that build up a node except the very first (the tag). But again, this should not be taken as a sign that the tag is a function, it is just an ordinary value.

We also define the *tag arity* to be the number of tag arguments, and the *size* of a node value to be the tag arity plus one (the tag itself).

It is important to understand that this view of tags and nodes does not really imply any "fixed" implementation, or mapping onto the hardware. GRIN nodes and tags are in principle just abstract notions used inside the GRIN language, and can be implemented in a number of ways. We do put two restrictions on an actual implementation, though. The first restriction is that given a node value there must be a simple way to "extract" the tag, the first argument, etc. The second restriction is very weak, and it is that tags must in some way be "distinguishable" from each other. In the current GRIN implementation we use small integers to represent tags (see section 4.4.3 on page 159), which in some sense is a "natural" implementation given how the language looks, but is in no sense a requirement. An implementation using pointers (e.g., to dispatch tables or to code) could also be used.

To make GRIN programs easier to read, we will differentiate between three kinds of tags:

- C-tags – representing value constructors (from ordinary datatypes),
- F-tags – representing suspended function applications (closures),
- P-tags – representing partial applications (of functions).

By convention, we write all C-tags starting with the letter 'C', all F-tags starting with 'F' and all P-tags starting with 'P'. We use a similar convention for node values when the node tag is known, saying F-node, C-node and P-node. Note that this is only a naming convention that we use when writing GRIN

programs. The GRIN language itself does not make any *a priori* interpretation of different tags. This is completely up to the GRIN code in the program.

The GRIN code generator will automatically create a number of unique tags according to:

- one C-tag for each value constructor in the program,
- one F-tag for each function in the program,
- a set of P-tags for each partially applied function in the program.

All P-tags, i.e., partial applications, have an additional "argument" encoding the number of missing arguments (relative to the original function). E.g., if the function *foo* has arity three and is partially applied somewhere in the program, we would create the tags:

$$\mathsf{Pfoo}_3,\ \mathsf{Pfoo}_2,\ \mathsf{Pfoo}_1,\ \mathsf{Pfoo}_0$$

where Pfoo_0 is really the same as Ffoo, a fully saturated function application (zero missing arguments). Higher order functions and partial applications in GRIN are described in more detail in section 2.5.6 on page 54.

Tags in GRIN can be "used" in two ways, either as stand-alone values or as *fully saturated* "applications" of a tag to its arguments. In other words, whenever a tag is applied to arguments, it must be applied to exactly the right number of arguments. For F-tags, this number is the same as the function arity, for C-tags it is the arity of the corresponding constructor, and for P-tags it is the number of "already supplied arguments" of that tag. E.g., for the function *foo* of arity three, the tag Ffoo must always be applied to exactly three arguments. If we need to apply "a *foo* tag" to fewer arguments, we must use a P-tag. Each tag Pfoo_n must be applied to exactly "$3 - n$" arguments.

2.3.4 Values

Values in GRIN (called *val* in the syntax, in figure 2.1) can be *nodes* and *tags* (as described above), *basic values* (like unboxed integers or characters) or *pointers* (to nodes stored in the heap). In fact, we will also call a tag "on its own" a basic value (a tag is not always part of a node value). Apart from these "normal" values there is also an *empty value*, written "()". The empty value is used mostly together with side-effecting GRIN operations, operations that return no real value.

Weak head normal form. It is important to understand that in GRIN, all *values* are weak head normal forms. There is no such thing as "closures" (suspended computations) since GRIN is a strict first order language. However, it is possible to *represent* closures from the "original program" using GRIN values. In fact, one of the main strengths of GRIN is that closures can be

represented as normal GRIN values. In the GRIN language this is manifested by the so-called F-nodes. An F-node, e.g., "(Ffoo a b)", is a GRIN value (i.e., is a whnf in the GRIN language) but represents a closure from the original program. From now on, whenever we use the term weak head normal form we will mean "in the original program", since it is not very meaningful to talk about whnfs in the GRIN program (everything in GRIN is in weak head normal form).

2.3.5 Variables

A GRIN variable can hold any kind of GRIN value, including entire node values. By convention, we will name variables containing basic values (including single tags) by prepending a single apostrophe ($'$).

In the GRIN language nothing is really said about the "size" of values or variables. However, the intention is that a GRIN variable will later be mapped onto one or more machine registers. Pointers and basic values will typically only need a single register whereas variables containing entire nodes will need several registers. Some basic values will probably also need more than one register, e.g., number types with higher precision.

One of the basic ideas of GRIN is that all variables should be seen as values in registers. All node variables will (at least initially) be given a number of registers by the code generator when translating GRIN to RISC machine code (see chapter 5). Some of these registers may eventually be *spilled* by the register allocator, i.e., temporarily kept in memory instead of in registers, but the main idea is nevertheless that GRIN variables should be seen as corresponding to machine registers.

In fact, the ability to keep entire node values in registers is closely related to the generalised variant of *unboxing* that is presented in section 4.3.3 on page 118.

2.3.6 Patterns

Patterns in GRIN appear in two places, in lambda bindings and in case expressions. Patterns in lambda bindings have the same syntax as values. Node patterns can have either a known tag or a tag variable in the "tag position", and either pointers or basic values in the "argument positions". In other words, only *simple patterns* [Aug85] are allowed.

Patterns in case expressions are even more restricted. Node patterns in a case expression must contain a known tag, and the arguments can only be variables. Besides node patterns, a single tag or a literal is also allowed in case patterns, but not a single variable. Additionally, patterns in case expressions must be *non-overlapping*. However, patterns do not have to be *exhaustive* (see the discussion in section 2.4.4 on page 41).

Note especially that a single variable is not a valid case pattern in GRIN.[1]

[1] This will probably change in the future, see the discussion in section 10.2.1.

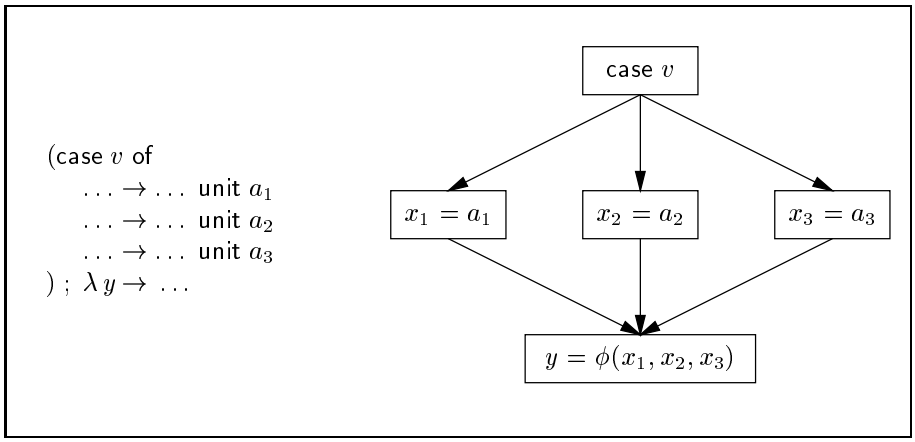

Figure 2.3: Comparison of GRIN and SSA.

2.3.7 Static single assignment

All variables in GRIN must have a *single definition*, i.e., appear in exactly one binding. This property is often called *static single assignment* (SSA, see e.g., [AWZ88, CFR+91]). The word static means that there must only be a single "occurrence" in the code of an assignment for each variable. In conventional SSA form, a variable is allowed to be dynamically assigned to more than once, e.g., if the assignment is inside a loop (in an imperative language). Of course, the same "variable name" can also be assigned to several times in different invocations of the same procedure, but then it is not really the "same" variable.

The relationship between GRIN and SSA is illustrated in figure 2.3, which compares a GRIN fragment with a similar piece of code written in a typical SSA style. Note how a *phi-node* is used in the SSA version to *merge* all the different variables. This is done automatically by the bind operator in GRIN. Note also that a case expression to the left of a bind operator is the only way to create a control flow *join* in GRIN.

Since GRIN does not have any (imperative) loop constructs, static single assignment in GRIN really means that once assigned a value for the first time, a variable can never "change" its value. This is an important part of our claim that GRIN is a functional language. The similarities between GRIN and SSA are not coincidental, since SSA is also a functional representation, although this is not often emphasised in the literature. An exception to that is [App98] which discusses the relationship between SSA and functional programming.

2.3.8 Syntactic sugar

We allow ourselves one simple form of syntactic sugar, conditional expressions. The expression:

$$\mathsf{if}\ b'\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2$$

is just a short-hand for:

$$\begin{array}{l}\mathsf{case}\ b'\ \mathsf{of}\\ \quad \mathsf{CTrue} \rightarrow k_1\\ \quad \mathsf{CFalse} \rightarrow k_2\end{array}$$

where CTrue and CFalse are the two predefined boolean tag values.

2.4 Semantics

To formally define the meaning of GRIN programs we will give a simple operational semantics, interpreting GRIN expressions. Having a formal semantics is a good thing. It can help in gaining an operational understanding of the language, and it can sometimes show inconsistencies or other semantic "peculiarities" of the language. Also, having a formal semantics is necessary to be able to prove anything about GRIN code or GRIN program transformations. However, our main reason to present a semantics will be to make the presentation of the GRIN language complete, and to convey an operational understanding of the language rather than to use the semantics to do proofs. In fact, our semantics may not be the most adequate type of semantics to use for reasoning about GRIN programs. Something in the lines of Launchbury's *natural semantics* [Lau93] would probably be more useful for that.

2.4.1 Semantic framework

The framework we use to explain the GRIN semantics can be seen in figure 2.4 on the following page. It describes the basic domains and functions used.

Since there are really two "parts" of the GRIN language, an expression language and a value language, the semantics is split correspondingly into two parts. The two basic semantic functions, $\mathcal{E}$ and $\mathcal{V}$, give the meaning of GRIN expressions and values, respectively. The GRIN expression semantics (see section 2.4.3 on page 37) is the most interesting part. The semantics of GRIN values are not much more than an identity function, mapping syntactic values to "isomorphic" semantic values, but is included here for completeness (see section 2.4.2 on page 37).

The semantic functions use an *environment* (ρ), mapping variables to semantic values, and a *store* (σ), mapping heap locations to values. The environment

Semantic functions :
$\mathcal{E}$ :: $exp \rightarrow Env \rightarrow Store \rightarrow (Value, Store)$
$\mathcal{V}$ :: $val \rightarrow Env \rightarrow Value$

Semantic domains :
$Value = i \mid t \mid (t\ v_1 \ldots v_n) \mid l \mid () \mid \bot$
$Env = Var \rightarrow Value$
$Store = Loc \rightarrow Value$

Semantic variables :
$i \in Int,\ t \in Tag,\ l \in Loc,\ v \in Value$

Syntactic variables :
$x \in var,\ e \in val,\ m \in sexp,\ k \in exp,\ c \in tag$

Utility functions :

sel_i	–	select the i:th component of a node value 0 means the tag, 1 means the first argument, etc.
$selalt_t$	–	select the case alternative matching the tag (or literal) t return the bound variables and the alternative body
newloc	–	return a new (unused) heap location
getglobal	–	find a global definition, return its parameters and body
runprim	–	execute a primitive operation

Figure 2.4: The semantic framework, domains and utilities.

can be seen as encoding storage for all the local variables (and function parameters) of the functions being "executed", mapping variable names to values. The store encodes the heap memory, together with the *newloc* utility function that is used to find a new unused heap cell. The only kind of GRIN values that can be put in the store are GRIN *node values* (see section 2.3.3 on page 30).

Semantic values

In the semantics we use a single value domain, *Value*, to represent all possible semantic values (the result of either $\mathcal{E}$ or $\mathcal{V}$). This domain is basically the same as the syntactic one (see section 2.3.4 on page 32). It consists of node values, tag values, unboxed integer values (representing all basic values) and locations (heap pointers). As in the GRIN language, the semantic value domain also includes a special *empty value* (no value at all) used for side-effecting operations that do not return a normal value. A special *undefined* value is also included (written

$$
\begin{aligned}
&\mathcal{V}\,[\,(c\ v_1\ \ldots\ v_n)\,]\ \rho = (c\quad \mathcal{V}\,[\,v_1\,]\,\rho\ \ldots\ \mathcal{V}\,[\,v_n\,]\,\rho)\\
&\mathcal{V}\,[\,(x\ v_1\ \ldots\ v_n)\,]\ \rho = (\mathcal{V}\,[\,x\,]\,\rho\quad \mathcal{V}\,[\,v_1\,]\,\rho\ \ldots\ \mathcal{V}\,[\,v_n\,]\,\rho)\\
&\mathcal{V}\,[\,x\,]\ \rho = \rho(x)\\
&\mathcal{V}\,[\,i\,]\ \rho = i\\
&\mathcal{V}\,[\,l\,]\ \rho = l\\
&\mathcal{V}\,[\,()\,]\ \rho = ()
\end{aligned}
$$

Figure 2.5: GRIN value semantics.

"$\perp$"). Note that this undefined value not always means "program failure", it can sometimes be allowed even in "working" programs (see the section about *Bindings* below).

Restrictions

All kinds of values are not meaningful in all contexts in the semantic equations, and we use different meta variables (described in figure 2.4) to place restrictions on some values. E.g., in describing the fetch operation, the environment ρ is applied to a variable. The result of this *must* be a location, otherwise a "program error" will result. In general, if any of these kinds of restrictions is not met, the semantics of the entire program is undefined.

2.4.2 GRIN value semantics

The semantic function $\mathcal{V}$ (see figure 2.5) gives the semantics of GRIN values. It is basically an identity mapping, translating between syntactic and semantic values. The only thing that really takes place in $\mathcal{V}$ is that variables will be looked up in the environment. Note that for a pointer variable, this lookup will result in a value that is a heap location, which is also what is returned from $\mathcal{V}$. There is never any implicit "reading" of the heap in $\mathcal{V}$. In fact, the heap (i.e., the store) is not even available to the $\mathcal{V}$ function. The only thing in GRIN that can read the contents of a heap node is the fetch monad operation (see section 2.4.5 on page 41).

2.4.3 GRIN expression semantics

The $\mathcal{E}$ function (see figure 2.6 on page 39) takes a syntactic expression, an environment (ρ) and a store (σ), and returns a possibly modified store together with the semantic value of the expression. The store must be returned from

$\mathcal{E}$ since some of the GRIN operations have side effects (write to the heap). Expressed in that way, the $\mathcal{E}$ function is really a small functional program. The only exception to this is the *runprim* operation which besides returning a value also can have an effect on the "outside world", e.g., doing Input/Output. Such external effects are not part of our semantics.

The semantics of a GRIN program is the value of $\mathcal{E}$ applied to the body of the function named *main*, in an initial store and environment. The initial store should contain unevaluated node values (F-nodes with no arguments) for all *Constant Applicative Forms* (CAFs). CAFs are global definitions without arguments (i.e., constants with global scope) and must be handled with some care [PJ87]. The initial environment should bind the names of all the CAFs to the corresponding locations for the CAF closures in the initial store. A CAF should not be evaluated until its value is first needed, which is why we must store F-nodes for all CAFs in the initial heap.

If any kind of "error" occurs during the execution of the program, the semantics of the entire program is undefined (a typical error would be a failed binding, see below).

2.4.4 Bindings

Patterns that bind variables occur in two situations in GRIN: in the monad *bind* operator (";") and in case expressions. To describe this in the semantics we introduce an *environment extension* operator ("$\mapsto$"). To simplify the semantics in figure 2.6 we define this operator to automatically take care of the fact that a GRIN pattern can bind several variables (for node patterns).

The bind operator

The semantics of the monad bind operator can be seen at the top of figure 2.6. As shown in the figure, if we have the GRIN expression "$m\ ;\ \lambda p \to k$" and the semantic value of m is v, we use the environment extension operator to extend the environment before applying the semantics to k:

$$\rho\,[\,p \mapsto v\,]$$

The extension operator should be interpreted in quite a general way. We define it to automatically "split" node patterns (when bound to node values), and bind all variables occurring in the pattern. E.g., if we have an environment extension like the following (an explicit node pattern):

$$\rho\,[\,(x_0\ x_1\ x_2)\ \mapsto\ (\mathsf{CCons}\ v_1\ v_2)\,]$$

it will really mean:

$$\rho\,[\,x_0 \mapsto \mathsf{CCons},\ x_1 \mapsto v_1,\ x_2 \mapsto v_2\,]$$

$$
\begin{array}{l}
\mathcal{E}\ [\![\, m\ ;\ \lambda p \to k \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ (v,\sigma') = \mathcal{E}\ [\![\, m \,]\!]\ \rho\,\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \rho' \quad\ = \rho\,[\, p \mapsto v \,] \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ \mathcal{E}\ [\![\, k \,]\!]\ \rho'\,\sigma' \\
\\
\mathcal{E}\ [\![\, \mathsf{case}\ e\ \mathsf{of}\ \mathit{alt}_1\ \ldots\ \mathit{alt}_n \,]\!]\ \rho\,\sigma\ = \\
\quad \mathit{case}\ \mathcal{V}\ [\![\, e \,]\!]\ \rho\ \mathit{of} \\
\quad\quad (t\ v_1\ \ldots\ v_n) \to \mathit{let}\ \ (x_1\ \ldots\ x_n, k) = \mathit{selalt}_t(\mathit{alt}_1\ \ldots\ \mathit{alt}_n) \\
\quad\quad\qquad\qquad\qquad\quad \rho' = \rho\,[\, x_1 \mapsto v_1\,, \ldots\,,\ x_n \mapsto v_n \,] \\
\quad\quad\qquad\qquad\quad \mathit{in}\ \mathcal{E}\ [\![\, k \,]\!]\ \rho'\,\sigma \\
\quad\quad t \qquad\qquad\quad \to \mathit{let}\ \ (\_, k) = \mathit{selalt}_t(\mathit{alt}_1\ \ldots\ \mathit{alt}_n) \\
\quad\quad\qquad\qquad\quad \mathit{in}\ \mathcal{E}\ [\![\, k \,]\!]\ \rho\,\sigma \\
\quad\quad i \qquad\qquad\quad \to \mathit{let}\ \ (\_, k) = \mathit{selalt}_i(\mathit{alt}_1\ \ldots\ \mathit{alt}_n) \\
\quad\quad\qquad\qquad\quad \mathit{in}\ \mathcal{E}\ [\![\, k \,]\!]\ \rho\,\sigma \\
\\
\mathcal{E}\ [\![\, \mathsf{unit}\ e \,]\!]\ \rho\,\sigma\ = (\mathcal{V}\ [\![\, e \,]\!]\ \rho, \sigma) \\
\\
\mathcal{E}\ [\![\, \mathsf{store}\ e \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ l\ \ = \mathit{newloc}(\sigma) \\
\qquad\qquad\qquad\qquad\qquad\quad \sigma' = \sigma\,[\, l \mapsto \mathcal{V}\ [\![\, e \,]\!]\ \rho \,] \\
\qquad\qquad\qquad\qquad\ \ \mathit{in}\ (l, \sigma') \\
\\
\mathcal{E}\ [\![\, \mathsf{update}\ x\ e \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ l\ \ = \rho(x) \\
\qquad\qquad\qquad\qquad\qquad\quad \sigma' = \sigma\,[\, l \mapsto \mathcal{V}\ [\![\, e \,]\!]\ \rho \,] \\
\qquad\qquad\qquad\qquad\ \ \mathit{in}\ ((), \sigma') \\
\\
\mathcal{E}\ [\![\, \mathsf{fetch}\ x \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ l = \rho(x) \\
\qquad\qquad\qquad\qquad\qquad\quad v = \sigma(l) \\
\qquad\qquad\qquad\qquad\ \ \mathit{in}\ (v, \sigma) \\
\\
\mathcal{E}\ [\![\, \mathsf{fetch}\ x\,[i] \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ l = \rho(x) \\
\qquad\qquad\qquad\qquad\qquad\quad v = \sigma(l) \\
\qquad\qquad\qquad\qquad\ \ \mathit{in}\ (\mathit{sel}_i(v), \sigma) \\
\\
\mathcal{E}\ [\![\, \mathit{fun}\ e_1\ \ldots\ e_n \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ (x_1\ \ldots\ x_n, k) = \mathit{getglobal}(\mathit{fun}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad v_1 = \mathcal{V}\ [\![\, e_1 \,]\!]\ \rho \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad v_n = \mathcal{V}\ [\![\, e_n \,]\!]\ \rho \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \rho' = \rho\,[\, x_1 \mapsto v_1\,, \ldots\,,\ x_n \mapsto v_n \,] \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathit{in}\ \mathcal{E}\ [\![\, k \,]\!]\ \rho'\,\sigma \\
\\
\mathcal{E}\ [\![\, \mathit{prim}\ e_1\ \ldots\ e_n \,]\!]\ \rho\,\sigma\ = \mathit{let}\ \ v_1 = \mathcal{V}\ [\![\, e_1 \,]\!]\ \rho \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ v_n = \mathcal{V}\ [\![\, e_n \,]\!]\ \rho \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ (\mathit{runprim}(\mathit{prim},\ v_1\ \ldots\ v_n), \sigma)
\end{array}
$$

Figure 2.6: GRIN expression semantics.

In general, since node values are first class citizens in GRIN we must also take into account what happens if a node value gets bound to a pattern of different *size* (the size of a GRIN node is defined as the number of "parts" in it, including the tag, see section 2.3.3 on page 30). At first, this might seem as as an error, but it is in fact very useful (and essential in GRIN) to allow a node to be bound to a pattern of greater size (but *not* to a pattern of smaller size). To motivation for this is the GRIN *vectorisation* program transformation (see section 4.2.4 on page 97). As an example, consider the function *foo* in figure 4.10 on page 99, which returns a list node value. Before the transformation the returned value is handled by a node variable ("v"), but after the vectorisation the "register usage" of the code has been made a bit more clear by introducing an explicit node instead of "v" (the node pattern "$(t'\ a_1\ a_2)$"). The size of the pattern is chosen as the maximum size of the node values that can occur at that point (a CCons tag in this example). However, when the actual value returned by *foo* is a CNil node then we get exactly the situation above, with a size "mismatch" in the semantics.

To handle this the environment extension operator must be defined also in the case of a node value that is bound to a pattern of a larger size. We solve this by binding the superfluous variables to the *undefined* value. I.e., the node binding:

$$\rho\,[\,(x_0\ x_1\ \ldots\ x_n)\ \mapsto\ (t\ v_1\ \ldots\ v_m)\,] \qquad\qquad m \leq n$$

will be translated to:

$$\rho\,[\,x_0 \mapsto t,\ x_1 \mapsto v_1, \ldots, x_m \mapsto v_m,\ x_{m+1} \mapsto \bot, \ldots, x_n \mapsto \bot\,]$$

The undefined value is written "$\bot$". Note that this is not the same as the *empty* value described earlier (written "()"). Having a variable bound to the undefined value is completely safe as long as we do not try to "access" the value of that variable. If we do that, the result of the entire program is undefined. Even in totally correct GRIN code it can sometimes be normal to have variables bound to the undefined value. This can be seen in the vectorisation example cited above (figure 4.10 on page 99). The example works as long as the code inside "$\langle m_1 \rangle$" never tries to access the variables "a_1" and "a_2" (which it will never do). The code can be further simplified as can be seen in figure 4.11 on page 100 (called *case simplification*).

Note that the other way around, binding a node value to a pattern that is smaller, is not defined. This is a situation that should never occur in a correct GRIN program. It would lead to a program error and the semantics of the program would be undefined.

Note also that the above automatic splitting in the environment extension operator does not apply unless the pattern is a node pattern. E.g., it would not happen if the value was a node but the pattern was just a (node) variable. In

that case a normal binding would be added, like in:

$$\rho\,[\,v \;\mapsto\; (\mathsf{CCons}\; v_1\; v_2)\,]$$

Since we allow constant tags and literals in patterns, bindings can also fail to match. E.g., if we try to bind a constant tag value to a constant tag pattern and the two tags are not identical. This would result in an extension of the environment like the following:

$$\rho\,[\,\mathsf{CCons} \mapsto \mathsf{CNil}\,]$$

If this happens, the binding *fails*. A failed binding anywhere in a GRIN program means that the semantics of the entire program is undefined.

Case expressions

There are two parts in giving the semantics to GRIN case expressions: selecting the right alternative and the binding of variables. To select the right alternative it is enough to check the tag (or literal) of the scrutinised expression (due to the syntactic restrictions on case patterns, see section 2.3.6 on page 33). In the semantics the utility function $selalt_t$ is used to select the alternative matching the tag (or literal) "t", or if none matches fails. Since GRIN case patterns are not required to be exhaustive, it is in theory possible that a case expression can fail. In practice however, even if a programmer writes a "non-exhaustive" case expression, the front-end is expected to fill in the unused constructors together with code to produce "no match" error messages. Hopefully, the *heap points-to analysis* (see chapter 3) can later deduce that some alternatives are indeed impossible, and can be removed by the *sparse case optimisation* (see section 4.3.6 on page 127).

Patterns in case expressions can also bind variables (for node patterns). These bindings are defined in the same way as for lambda patterns (see above). In the semantics, $selalt_t$ returns the bound variables of the pattern in addition to the selected alternative. In this case though, the issue of different size between the node value and the pattern will not appear, since GRIN tags are always *fully saturated* (see section 2.3.3 on page 30).

2.4.5 Monad operations

There are four built-in GRIN monad operations: unit, store, update and fetch. The unit operation is the basic component that together with a *bind* (our ";") defines the base of any monad [Wad92]. The other three monad operations are specialised to the GRIN monad.

The unit operation is used to *return* values, either "internally" inside a GRIN function, or "externally" (return a value as a result from a function). For an example of both kinds of unit operations, see figure 4.2 on page 85. In the

semantics we simply lookup the value of the expression being returned, using $\mathcal{V}$ (see figure 2.6).

The store operation allocates a new heap cell, using the "*newloc*" utility function, and writes its argument value at that location. It is only allowed to write entire node values to the heap, i.e., not pointers or basic values "on their own". The allocated location is returned as the result of the store operation.

The update operation is almost like a store but it overwrites an already existing heap node rather than allocating a new one. The point of the update operation is its side-effect, it returns the empty value. In the code that is generated from the current GRIN code generator the update operation will never be used in "ordinary" GRIN code, it will only appear as part of the special *eval* procedure (see section 2.5.2 on page 44). However, after some GRIN program transformations, update operations can appear anywhere in a GRIN program.

The fetch operation takes a location (heap pointer) as argument and returns the node value stored at that location. Note that a node value that is loaded from the heap using fetch does not have to represent a weak head normal form, it could very well represent a suspended computation (a closure). If a whnf is desired a call to *eval* should be used instead of a fetch operation (see section 2.5.2 on page 44).

As with update, we will initially only use the fetch operation inside the *eval* procedure, but later on it can appear anywhere in a GRIN program.

The fetch operation can also take a second argument, an *offset*, in which case a certain "component" of the node value will be returned rather than an entire node value. In the semantics, the utility function sel_i is used to select a component of a node value. Offset 0 is defined to return the node tag, offset 1 the first "argument", etc. The sel_i function is defined to return the *undefined* value if component "i" does not exist in the node value. Introducing offset fetch operations is one step in transforming the GRIN program into a very "simple" and "low level" form (see the *split fetch* transformation in section 4.2.7 on page 102). Note that for this transformation to work it is necessary that the semantics for a fetch operation with an illegal index does not immediately fail, just returns the undefined value (which gets bound to some variable and put in the environment).

Although not a direct requirement of the GRIN language, we have in our implementation an additional restriction on all GRIN *memory operations*, i.e., store, update and fetch (but not unit). The actual memory layout of a node value in the current implementation depends on the node tag (see section 5.2.3 on page 178). This means that some memory operations may need to be *annotated* with a concrete tag, but only in certain circumstances. This is described in more detail in the sections on GRIN program transformations, e.g., see the GRIN *update specialisation* (4.2.3 on page 90).

2.4.6 Function application

Function application GRIN are restricted in one important way, the function symbol (the name in the function position of an application) *must* be the name of a known global function (or a primitive operation). This is a very important property of GRIN, *all* function calls are to known, global functions (this is a first order language)!

Of course, we must also be able to handle calls that in the original program code are "unknown", e.g., a function application where the function is an incoming parameter. The methods we use to deal with this are described later (*Higher order functions*, see section 2.5.6 on page 54). Note that these methods will not break the above property that all GRIN calls are to known functions, instead unknown calls in the source code are turned into a "test" followed by a number of calls to known functions, depending on the test.

In the semantics of GRIN function calls, we first calculate the values of all arguments using $\mathcal{V}$, before doing the actual "call". This mean that function application in GRIN is strict. However, this is not really of great importance, since there is no computation going on in the $\mathcal{V}$ function. It is only a simple mapping between the syntactic and the semantic value domains. The order in which the function arguments are evaluated does not matter since $\mathcal{V}$ has no side effects (it does not involve the store).

2.5 Compiling to GRIN

To conclude the presentation of the GRIN language and to make GRIN a bit more concrete, we will now show some examples of how it can be used. We will also discuss some further issues relating to the use of GRIN in this thesis, rules and conventions that do not really fit within either the description of the GRIN syntax or semantics. As an example, we will present how graph reduction can be encoded in GRIN, and how the `update` monad operation is supposed to be used. We will also discuss GRIN code generation, calling conventions and higher order functions.

2.5.1 An example program

As explained in the introduction we translate a very simple functional language into GRIN (see the compiler overview in figure 1.3 on page 17). As a first example of GRIN code, we will show a possible translation into GRIN of the small functional program in figure 2.7 on the following page. It is written in a Haskell-like notation, but without "complicated" constructs. The list *cons* operator is written ":", and the empty list "[]". The program generates a list of numbers and computes their total sum.

A GRIN version of the program is shown in figure 2.8 on page 45. Here we have assumed a rather unsophisticated translation, to really show the core of

```
main = sum (upto 1 10)

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

sum l = case l of
          []     → 0
          n : ns → n + sum xs
```

Figure 2.7: A small functional program.

GRIN, the encoding of graph reduction.

The *main* function will build an initial piece of graph, using the store monad operation, and then call *eval* to "start" the graph reduction. Eventually, an integer value will be returned which is output using the built-in primitive operation *intPrint*.

2.5.2 Forcing suspensions – the *eval* procedure

In all implementations of graph reduction there must be a way to force (or evaluate) a suspended computation (also called *suspension*, *closure* or *thunk*), and turn it into a *weak head normal form* (whnf). There exists a number of ways to implement this "forcing", e.g., as special code in the runtime system [Joh84] or as a tagless pointer dispatch [PJ92] (see section 2.6.1 on page 59 for a discussion of more related work).

GRIN, however, is expressive enough to be able to implement an *eval* function within the language itself, or rather an *eval procedure* (as we will usually call it to emphasise its side-effects, even if it always returns a value). In the program in figure 2.8 on the next page, apart from the "normal" functions we have also included an *eval* procedure for that particular program.

The *eval* procedure is given a pointer as argument, and will start by loading the node pointed to from the heap and then examine it, using an ordinary GRIN case expression. If it finds a GRIN node value representing a weak head normal form it will simply return it, using a unit operation. In this simple program all whnfs are represented by C-nodes, but in general, they can also be P-nodes (representing partial applications). If *eval* finds a suspended function application (an F-node), it will "extract" the function arguments and call the corresponding function. Eventually, the function will return a node value representing a whnf, after which *eval* will update (overwrite) the original redex with the new value.

```
main = store (CInt 1) ; λt1 →
       store (CInt 10) ; λt2 →
       store (Fupto t1 t2) ; λt3 →
       store (Fsum t3) ; λt4 →
       eval t4 ; λ(CInt i') →
       intPrint i'

upto m n = eval m ; λ(CInt m') →
           eval n ; λ(CInt n') →
           intGT m' n' ; λb' →
           if b' then
             unit (CNil)
           else
             intAdd m' 1 ; λx' →
             store (CInt x') ; λt5 →
             store (Fupto t5 n) ; λt6 →
             unit (CCons m t6)

sum l = eval l ; λu →
        case u of
          (CNil)        → unit (CInt 0)
          (CCons t ts)  → eval t ; λ(CInt t') →
                          sum ts ; λ(CInt r') →
                          intAdd t' r' ; λs' →
                          unit (CInt s')

eval p = fetch p ; λv →
         case v of
           (CInt x')      → unit v
           (CNil)         → unit v
           (CCons x xs)   → unit v
           (Fupto a b)    → upto a b ; λw →
                            update p w ; λ() →
                            unit w
           (Fsum c)       → sum c ; λz →
                            update p z ; λ() →
                            unit z
```

Figure 2.8: A GRIN version of the small program.

Finally, *eval* returns the value to its caller.

To be safe, the case expression inside *eval* must enumerate all node values that are ever the subject of a store (or update) operation in the program. This may seem impractical, for all but the smallest toy programs such a case expression would become very large. However, as we will later show, the "general" *eval* procedure can be completely eliminated from all GRIN programs after some analysis and program transformations (see section 4.2.1 on page 84). Having a general *eval* procedure is only a temporary step during the GRIN compilation process, and in fact, in the actual implementation the "full" *eval* procedure will never be created at all. Note that a basic assumption of the GRIN back-end is that the entire program is available at compile time, otherwise we would not be able to construct an *eval* procedure in this way.

Even though the general *eval* procedure will not be directly used, the ability to "tailor" a specialised *eval* for every program is a very important part of GRIN. The key to all this is GRIN's ability to "inspect" a node representing a closure without evaluating it (the F-nodes). This is something that can never be done in ordinary "functional" code, e.g., the STG language [PJS89], the intermediate code used by the Glasgow Haskell compiler.

This basic idea of encoding closures in a first-order language using unique tags and a *dynamic dispatch* function is originally due to Reynolds [Rey72].

2.5.3 Calling conventions

As explained earlier a variable in GRIN can hold either pointers, basic values (including single tags), or complete node values. For function arguments and return values the following rules apply:

- function arguments can be any kind of value,
- functions can return basic values and node values but *not* pointers.

In addition, a returned node value must represent a weak head normal form (either a C-node or a P-node, not an F-node).

The restriction not to allow returned pointers is a deliberate design choice, and the reason is two-fold:

- one of the original motivations for GRIN was to improve the register usage of lazy functional code, by exposing more "registers" in the code. One way to do that is to return entire nodes from functions, using several registers, rather than to return a pointer to a node in the heap using a single register,
- the GRIN handling of calls to *eval*, and in particular update operations make it natural to return entire nodes (see below).

2.5.4 Updates

In order to guarantee a *call by need* evaluation order in an implementation of a lazy functional language, some kind of *update operation* is required. The point of the update is that after a closure (suspended computation) has been forced into a whnf for the first time, the original closure (*redex*) is *overwritten* with the whnf value, so that subsequent references to the same computation immediately can return the value.

In the G-machine [Joh84], a function is itself responsible for updating the "current apply node" with a whnf just before it returns (there are some variations to this, in different variants of the G-machine). In the STG machine [PJS89, PJ92], an *update frame* is pushed onto a special stack before calling (called *entering*) a closure, and then popped off (performing the update) just before a value is returned. These two methods have in common that the "original closure" (i.e., a pointer to the cell in the heap to be updated) is automatically "known" by the code performing the update. In the former case the pointer is always at the top of the stack, and in the latter the pointer is part of the stored frame.

In GRIN, we want to do updates using an explicit operation inside the language. In contrast to the above methods, a pointer to the original closure must be passed as an explicit argument to the GRIN `update` operation. This immediately disqualifies the G-machine method, to put the update operation last in the function being called, because the redex pointer is simply not in scope at that point. Additionally, we want as much of "what happens" during runtime to be visible in the GRIN code, and consequently a built-in update in the runtime system is not a very tempting idea. The update should be an explicit GRIN operation.

This basically leaves us with two choices: to always put an `update` operation after each call to *eval*, *or* to make the *eval* procedure responsible for performing the update. The former choice has the disadvantage that we will always update, even if the value was already a whnf. The latter choice, which is what we have adopted in GRIN, is more natural in that sense. As can be seen in the *eval* procedure in figure 2.8 on page 45 updates need only be performed when the original node represents a closure (an F-node), not for weak head normal forms (C-nodes and P-nodes). Using a *sharing analysis* we can do even better, and only include updates for *shared* closures (see section 3.2.2 on page 69). These optimisations would have been more difficult if the `update` had been separate, and not a part of the *eval* procedure.

Updates and return conventions

The way updates are done is also closely related to how values are returned from functions, the "return part" of the calling convention (see above). When *eval* is responsible for updates, the choice to return a node value rather than

a pointer to a node in the heap, is very natural. A called function (the callee) does not really know anything about how the caller intends to use the returned value. By returning a node value (using several registers in the real code) we leave it to the caller to decide if the value needs to be written to the heap at all. In many cases that will not be necessary!

Consider for a moment the other choice, to return a pointer instead. Often, a function "creates" the return value in registers, so if it had to return a pointer, the node value would first have to be written to the heap by the callee just before it returns. Then, if the caller needs to do an update, it must again write the same value to the heap, now to overwrite the original closure. In this case we would write the same value twice to memory immediately after each other, which seems like a clear waste. Another situation is when a caller immediately needs to "use" the returned value, e.g., in a case expression (this is probably a very common situation). In this case, the caller would have to load the value from the heap again, the same value that the callee just wrote! By returning complete node values in several registers, and by leaving the responsibility to do the update with the caller, we will hopefully avoid situations like the above. Of course, it is possible to construct a situation where returning a pointer would instead be better, but we believe this to be less common.

The handling of updates is further discussed in section 4.2.3 on page 90, in the description of the *update specialisation* transformation.

2.5.5 GRIN code generation

The GRIN code generation is done from a simple functional language (called λ) that is input to the GRIN back-end. The organisation of the GRIN part of the back-end is shown in figure 2.9 on the facing page. See also figure 1.3 on page 17 for an overview of the complete compiler.

The GRIN code generation is quite straightforward. In this section we will give a few examples, to show the general strategy, and to highlight some important details.

The functional language λ is very much like the *Core* language [PJ96] used by the Glasgow Haskell Compiler. The syntax of λ is shown in figure 2.10 on the facing page. Unlike standard Core, the code must be lambda lifted [Joh85], i.e., all functions must be super combinators [Hug82]. All "high level" Haskell constructions, like complicated patterns in bindings, list comprehensions and type classes, have been transformed away by the front-end. In the GRIN back-end, we also assume that the entire program is available.

The front-end includes a transformation where all expressions are given a name, i.e., the transformation introduces a let binding for each sub-expression of a complicated expression. As an example, this will result in all function application arguments being variables (or literals). The "functions" will also be variables (or globals, of course). Evaluation is done using either case expressions or strict let expressions.

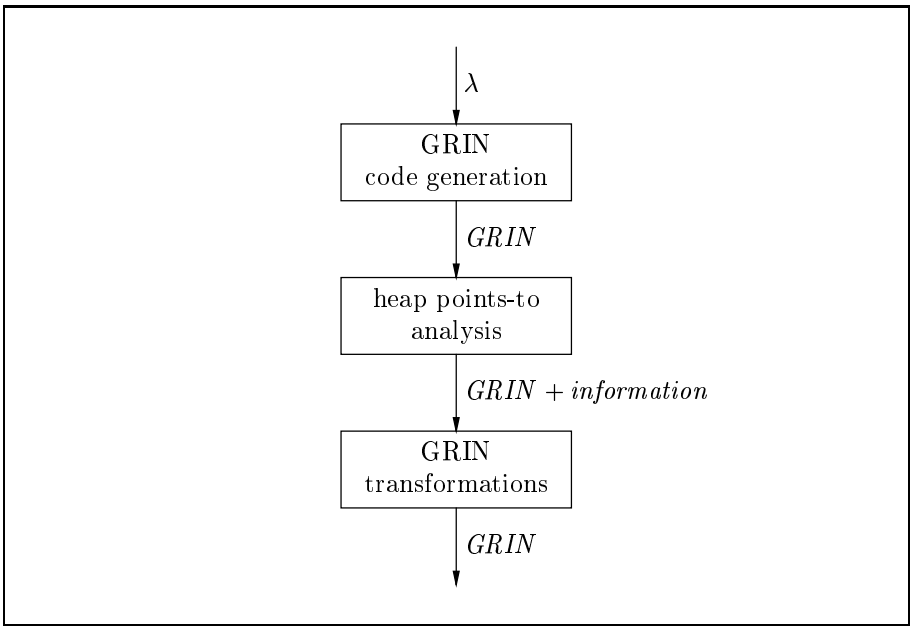

Figure 2.9: Overview of the GRIN part of the back-end.

binding	::=	*var* { *var* }∗ = *exp*	top-level bindings
exp	::=	*var* { *atom* }∗	application
	|	case *exp* of { *pat* → *exp* }+	case expression
	|	let { *var* = *exp* }+ in *exp*	let bindings
	|	letS *var* = *exp* in *exp*	strict let binding
	|	*con* { *atom* }∗	constructor
	|	*atom*	
atom	::=	*var* | *literal*	variable or constant
pat	::=	*con* { *var* }∗ | *literal*	

Figure 2.10: Abstract syntax of λ, a simple combinator language.

All these things taken together make the simple functional language very easy to translate into GRIN code.

Code generation schemes

The GRIN code generator uses a number of *compilation schemes* in a style very similar to the original code generator for the G-machine [Joh84, PJ87]. The $\mathcal{E}$ scheme translates an expression in a *strict context*, i.e., it will generate code to evaluate the expression to a value (a weak head normal form). The $\mathcal{C}$ scheme on the other hand translates an expression in a non-strict context, i.e., it will build graph (a representation of the suspended computation). The $\mathcal{R}$ scheme is a special version of the $\mathcal{E}$ scheme that is used to translate a right hand side of a function definition (or really, the entire "returning spine" of a function definition). In addition to the evaluation $\mathcal{R}$ will also emit a "function return".

We will now show some schematic examples of how the compilation schemes can be used to translate different constructs of the simple functional language (λ) into GRIN. For simplicity, we will leave out all "book keeping", like environments and other tables that a real code generator would need. We will also not bother with translating "variable names" etc, but none of this should affect the general ideas. In the examples of strict contexts below, we will alternate between showing either $\mathcal{R}$ or $\mathcal{E}$ (they result in exactly the same GRIN code in most cases).

Let expressions. The first example: an ordinary non-recursive let expression, appearing in a strict context (the $\mathcal{R}$ scheme). We use the $\mathcal{C}$ scheme to build graph for the binding, and then $\mathcal{R}$ again to evaluate the result:

$$\mathcal{R}\ [\mathsf{let}\ v = e_1\ \mathsf{in}\ e_2\,] \quad = \quad \mathcal{C}\ [\,e_1\,]\ ;\ \lambda p \rightarrow \mathcal{R}\ [\,e_2\,]$$

The GRIN expression resulting from "$\mathcal{C}\ [\,e_1\,]$" will return a pointer to a node value in the heap. This pointer is bound to the variable "p" in the rest of the code (which is still in a "returning context", thereby the use of the $\mathcal{R}$ scheme for "e_2").

For a strict let expression (written "letS"), we will instead immediately evaluate "e_1" using the $\mathcal{E}$ scheme:

$$\mathcal{R}\ [\mathsf{letS}\ v = e_1\ \mathsf{in}\ e_2\,] \quad = \quad \mathcal{E}\ [\,e_1\,]\ ;\ \lambda v \rightarrow \mathcal{R}\ [\,e_2\,]$$

Here, the result of "$\mathcal{E}\ [\,e_1\,]$" will be a node value rather than a pointer. Both the $\mathcal{E}$ and the $\mathcal{R}$ schemes return either basic values or entire node values, depending on the context, but never pointers.

For methods to handle recursive let bindings, i.e., truly circular data structures (not functions, they are all lambda lifted), see the discussion on page 52.

Function calls. If we find a function application in a non-strict context we create a closure in the heap, using the store monad operation:

$$\mathcal{C}\,[\,foo\;x\;y\,] \;=\; \mathsf{store}\;(\mathsf{Ffoo}\;x\;y)$$

whereas in a strict context we would just do the call:

$$\mathcal{E}\,[\,foo\;x\;y\,] \;=\; foo\;x\;y$$

Note that the above only applies when *foo* is a known function and the number of actual arguments matches the arity of the function. If not, the techniques in section 2.5.6 on page 54 (higher order functions) are used instead.

Return a value. Returning a node value is very simple, we just use the GRIN unit operation:

$$\mathcal{R}\,[\,Cons\;x\;xs\,] \;=\; \mathsf{unit}\;(\mathsf{CCons}\;x\;xs)$$

Note that the unit operation creates a node value "in registers", it does not write anything to the heap. A return in GRIN does not necessarily have to mean "function return", it can also mean returning a value anywhere "inside" a GRIN expression. E.g., if we have an embedded case expression (to the left of a bind), all the branches of the case expression will "return" a value into the top level binding. For an example of this, see figure 4.2 on page 85 where the result of an internal unit is returned into the variable "*w*". Both function return and this kind of internal return are done using the unit operation.

Evaluation. When we find a (pointer) variable in a strict context we can not just load the value from the heap using the fetch operation. We must also make sure that the value it points to represents a whnf. But this is exactly what the *eval* procedure is for:

$$\mathcal{E}\,[\,x\,] \;=\; eval\;x$$

As described in section 2.5.2 on page 44 the *eval* procedure is constructed automatically for each program. Normally, the "need" to evaluate a variable arises because its value is needed. As an example, to add two (boxed) integer variables, we would generate code like the following:

$$\begin{aligned}\mathcal{E}\,[\,x+y\,] \;=\;& eval\;x\;;\;\lambda\,(\mathsf{CInt}\;x')\rightarrow\\ & eval\;y\;;\;\lambda\,(\mathsf{CInt}\;y')\rightarrow\\ & intAdd\;x'\;y'\;;\;\lambda\,z'\rightarrow\\ & \mathsf{unit}\;(\mathsf{CInt}\;z')\end{aligned}$$

The primitive operation *intAdd* is used to add the two integer basic values, and then an integer node is returned using a unit operation. Note that if the variables

"x" and "y" in the original program had been *unboxed* variables [PJL91], then there would be no need to evaluate them at all, and no need to "reconstruct" an integer node after the addition.

In the original G-machine, the $\mathcal{E}$ scheme could be defined as first building a graph, using the $\mathcal{C}$ scheme, and then calling the G-machine EVAL instruction. Exactly the same is true in GRIN, illustrated by the following:

$$\mathcal{E}\ [e] \quad \equiv \quad \mathcal{C}\ [e]\ ;\ \lambda p \rightarrow \mathit{eval}\ p$$

Here, the "$\equiv$" should be read as "will generate code that gives the same result" rather than "will generate the same code". Hopefully the $\mathcal{E}$ scheme will generate more efficient code.

If a variable that needs to be evaluated is not a local variable but instead the name of a global symbol, then it must be a Constant Applicative From (CAF), i.e., a global without arguments. In GRIN, a CAF is evaluated in exactly the same way as "ordinary" pointers, the only difference is that the CAF name really refers to the "global" heap location associated with that particular CAF. If the variable is instead the name of a "normal" global function, we have really found a partial application (with no arguments) and need to use the methods for higher order functions described in the next section.

A simple, and standard, optimisation to the insertion of calls to *eval* is to locally track what variables have already been evaluated. If a variable is found for the second time, we can just reuse the previous result of the *eval* call.

Case expressions. A case expression in the functional language will translate into a GRIN case expression:

$$\mathcal{R}\left[\begin{array}{l}\mathsf{case}\ e\ \mathsf{of}\\ \quad p_1 \rightarrow e_1\\ \quad\quad\vdots\\ \quad p_n \rightarrow e_n\end{array}\right] \quad = \quad \begin{array}{l}\mathcal{E}\ [e]\ ;\ \lambda v \rightarrow\\ \mathsf{case}\ v\ \mathsf{of}\\ \quad \mathcal{P}\ [p_1] \rightarrow \mathcal{R}\ [e_1]\\ \quad\quad\vdots\\ \quad \mathcal{P}\ [p_n] \rightarrow \mathcal{R}\ [e_n]\end{array}$$

where $\mathcal{P}$ is the scheme used to translate patterns. Note how the $\mathcal{R}$ scheme is propagated to the right hand sides of the case alternatives. In the current code generator, the $\mathcal{E}$ scheme will never be used for case expressions (a property of the code generated by the front-end).

Circular data structures. Using the GRIN code as presented earlier it is possible to capture all properties needed when implementing a lazy functional language, except for one thing: circularly defined data structures [Bir84, Joh87a]. We have not shown how to do this simply because it has not yet been implemented in the (experimental) GRIN back-end. The reason is not a technical difficulty, it would be easy to add the necessary parts to the language, as we will show in this section, we have just not had the time to implement it yet.

First, we assume that the compiler front-end has done the standard *dependency analysis* on let bindings, to separate the bindings as much as possible. After this, most of the lets will (probably) be non-recursive bindings of single variables, for which the standard code generator techniques apply. Furthermore, all functions (recursive or not) defined via let bindings have been lifted to the top level by the lambda lifting transformation. This leaves us with a few truly recursive let bindings, that may bind one or more variables at the same time, creating circular data structures. An example would be the functional code (using letrec rather than let to emphasise the recursive binding):

```
letrec
  ones = 1 : ones
in
  e1
```

Here, *ones* will become a circular data structure. To build such a structure using GRIN, there are at least two possible alternatives.

The first possibility is to use the primitives already available in GRIN, but with a slight abuse of the store operation. E.g., we could create the *ones* structure using:

```
store (Hole () () ) ; λp →
store (CInt 1) ; λq →
update p (CCons q p) ; λ() → ...
```

Here, we expect the first store operation to be interpreted in a special way by the code generator, simply allocating an "empty" node value of size three. The update operation will then fill the node with a "cyclic" CCons node. Of course, we could also add a new GRIN operation "alloc" to allocate heap memory.

This method is in fact exactly what is used by most implementations to create circularly defined values. First, a "hole" is allocated which later is overwritten (updated) with the real value.

An alternative method, is to add a new primitive to the language, a *fix-point operator*. The above example could then be written:

```
store (CInt 1) ; λp →
fix (λq → store (CCons p q))
```

Here, the point is that the binding of the variable "q" should be available already before the actual store operation have been performed. An implementation of this fix operator could be done exactly as the method above, first allocating a "hole" and then updating. However, when using the fix method such details will not be visible in the GRIN code, something which can be seen as an advantage. Even though we often claim that GRIN is nice to use because it offers "low level control", we do not want the language to become too low level. E.g., from the

point of view of the control flow analysis (see chapter 3) it may be easier to handle a fix-point operator than to handle an allocation of an empty node that later is updated.

2.5.6 Higher order functions

So far we have not yet shown any special constructs to handle *higher order functions* (HOFs). And in fact, it turns out that we do not need any special constructs to do that. The standard GRIN language is itself fully capable of expressing the machinery needed.

To handle HOFs, there are basically two different things to take care of, *partial applications* and applications of *unknown functions* (i.e., an application where the function symbol is a local variable, a function parameter or a CAF rather than a global function). The lambda lifting transformation [Joh85] have made sure that all applications of ordinary datatype constructors are fully saturated, so we do not need to consider them here.

Partial applications

When the GRIN code generator finds a partial application of a known function, i.e., with fewer actual arguments than the arity of the function, it will use a P-tag (see section 2.3.3 on page 30) to denote a node that represents a partial application. Assume that the function *foo* has arity three and that we are in a non-strict context, i.e., the $\mathcal{C}$ code generation scheme. If we find an application of *foo* to only two arguments, we would generate the code:

$$\mathcal{C}\ [\![foo\ x\ y]\!] \quad = \quad \mathsf{store}\ (\mathsf{Pfoo}_1\ x\ y)$$

The tag Pfoo_1 encodes that there is still one argument missing before the application becomes fully saturated. Note how the arguments already supplied are built into the P-node.

If we instead were in a strict context, the P-node itself is already a whnf, so we would just return it:

$$\mathcal{E}\ [\![foo\ x\ y]\!] \quad = \quad \mathsf{unit}\ (\mathsf{Pfoo}_1\ x\ y)$$

Applications of unknown functions

When we find an application with a variable in the function position, and are in a strict context, we must somehow be able to call an unknown function. Directly calling unknown functions is not allowed in GRIN (see section 2.4.6 on page 43), so we must work-around that in some way.

Fortunately, it is not very hard, and we will solve the problem in a way similar to the *eval* procedure used to force "unknown" closures (see section 2.5.2

$$
\begin{aligned}
&main = quad\ quad\ inc\ 0\\
&quad\ f = twice\ twice\ f\\
&twice\ f\ x = f\ (f\ x)\\
&inc\ n = n + 1
\end{aligned}
$$

Figure 2.11: A small functional program using HOFs.

$$
\begin{aligned}
apply\ f\ x = \mathsf{case}\ f\ \mathsf{of}\\
&(\mathsf{Pquad}_1) \rightarrow quad\ x\\
&(\mathsf{Ptwice}_1\ a) \rightarrow twice\ a\ x\\
&(\mathsf{Ptwice}_2) \rightarrow \mathsf{unit}\ (\mathsf{Ptwice}_1\ x)\\
&(\mathsf{Pinc}_1) \rightarrow inc\ x
\end{aligned}
$$

Figure 2.12: An example *apply* procedure.

on page 44). For unknown applications we will instead introduce an *apply* procedure. Like *eval* it is automatically constructed and tailored for each program. As an example, let us look at the small functional program in figure 2.11 on page 55. This program makes extensive use of higher order functions.

Assume for the moment that to write the GRIN code for the program it is enough if we can apply an unknown function to exactly one additional argument. This means that the *apply* procedure itself will take two arguments, a function object (always a P-node) and the additional argument. We will return to the more general case below. Given this assumption, an *apply* procedure for the example program could look like in figure 2.12.

The *apply* procedure. The *twice* function normally takes two arguments, so if it were partially applied it must be with one or two missing arguments. These two cases are represented by the Ptwice_1 and Ptwice_2 tags. The *apply* procedure in figure 2.12 always take exactly two arguments, a node value representing the partially applied function and one additional argument.

In the case when the function value is a Ptwice_1 node (one missing argument), applying it to an extra argument means that the application becomes fully saturated. When this happens, the actual function call can be performed, which eventually will return a value representing a whnf. Note that in this case, the first argument to *twice* ("a" in the figure) is already available in the original

$$
\begin{array}{l}
quad\ f = \mathsf{store}\ (\mathsf{Ptwice}_2)\ ;\ \lambda\, t_1 \rightarrow \\
\qquad twice\ t_1\ f \\
\\
twice\ f\ x = \mathsf{store}\ (\mathsf{Fap}\ f\ x)\ ;\ \lambda\, t_2 \rightarrow \\
\qquad\qquad eval\ f\ ;\ \lambda\, u \rightarrow \\
\qquad\qquad apply\ u\ t_2 \\
\\
ap\ f\ x = eval\ f\ ;\ \lambda\, v \rightarrow \\
\qquad\qquad apply\ v\ x
\end{array}
$$

Figure 2.13: GRIN code for part of the HOF program.

P-node. Also the Pquad_1 and Pinc_1 tags will result in fully saturated applications when an extra argument is supplied, and the corresponding functions will be called.

In the Ptwice_2 case, we will still have a partial application after the addition of one extra argument, so we return a node representing the resulting partial application (now with a Ptwice_1 tag).

Similarly to the *eval* procedure, we must for *apply* enumerate all partial applications that can ever occur in the program (i.e., appear in a store or an update operation). Fortunately, as with *eval*, the general *apply* procedure can be completely eliminated using analysis and program transformation techniques (see section 4.2.2 on page 89).

Using *apply*. Once we have created an *apply* procedure, generating code for applications of unknown functions is simple. Returning to our code generation schemes, and still assuming that all unknown applications have exactly one additional argument, we would get:

$$
\begin{array}{lcl}
\mathcal{E}\ [\![x\ y]\!] & = & eval\ x\ ;\ \lambda\, v \rightarrow \\
 & & \qquad apply\ v\ y
\end{array}
$$

Note that we must make sure that "x" is a whnf by calling *eval* before we can apply it. The result of the call to *eval* will be a node value, which is passed as first argument to *apply*. Because of that, this value must always be a P-node.

GRIN code for part of the example HOF program is shown in figure 2.13. Note that we have been forced to create an additional function, "*ap*", since there is no way in GRIN to create a closure of an unknown function (the application "$f\ x$", in the body of the *twice* function, which is in a non-strict context in the original program). This is in fact a common problem, shared with many other implementations of lazy functional languages. It is normally "fixed" already by

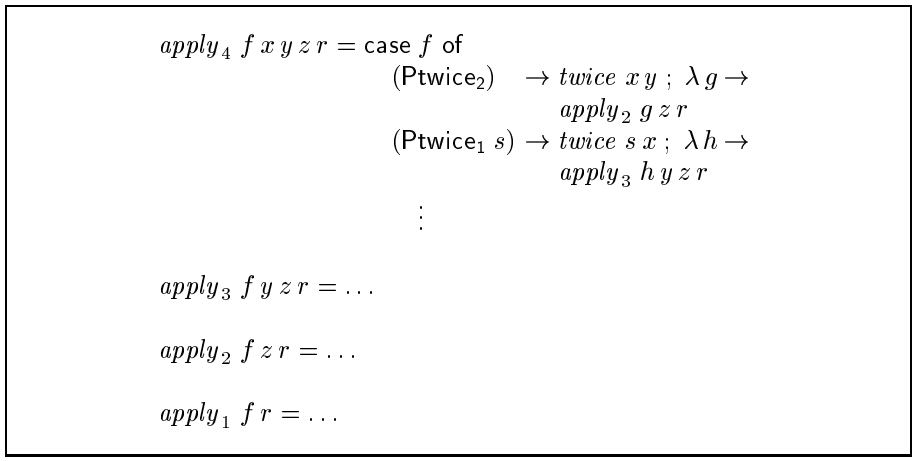

Figure 2.14: An alternative *apply* implementation.

the compiler front-end, during a *lambda lifting* pass (the application "$f\ x$" will be lifted and thus an *ap* function created).

Note also that in the *quad* function we store a Ptwice$_2$ node, representing a function object of *twice* with all arguments missing.

Applications to several arguments. Above we assumed that applications of unknown functions were always done to exactly one additional argument. But of course, the normal case is to have more than one extra argument. There are basically two ways to handle this in GRIN, either to use a series of $apply_N$ procedures (each taking N extra arguments), or to use a single *apply* procedure (as in figure 2.12) and create "chains" of calls to *apply*. The main idea of the $apply_N$ method is outlined in figure 2.14.

One of the "difficult" things about higher order functions is that we must be able to handle the possibility that the "first" unknown function does not "consume" all the supplied argument, but instead returns a new function that in turn will consume some more arguments, and so on. This is illustrated in figure 2.14 in the $apply_4$ procedure. Depending on the tag found $apply_4$ will consume a number of arguments, perform a function call, and finally call an *apply* procedure of lower arity.

This capability is basically the same thing as the UNWIND operation in the original G-machine [Joh84]. In GRIN however, all the work can be done explicitly within the language rather than by a built-in operation.

In $apply_4$ we also see the fact that the *twice* function actually returns functions (or really node values representing functions, i.e., P-nodes).

Although the above method looks attractive, it has one serious drawback, it

does not directly work with the program transformation technique that we will later use to eliminate calls to *apply*.

We will instead use the alternative method, i.e., to use a single *apply* procedure and create chains of calls to it. This is conceptually an even simpler method than the above, but it has one potential inefficiency. Creating chains of calls to *apply* implies the unnecessary construction of intermediate function values (P-nodes). For long *apply* "chains" we might have to scrutinise the same value (almost) over and over again until the right number of arguments have been provided to do an actual call. Fortunately, this overhead can be largely avoided using GRIN program analysis and a simple program transformation, called *case hoisting* (see section 4.3.11 on page 137).

The single *apply* method is what we will use in the rest of this thesis.

Generating *apply* chains. The code generation schemes are extended to generate chains of calls to *apply* when an unknown function is applied to more than one argument. For the two argument case we get:

$$\begin{array}{lcl} \mathcal{E}\ [x\ y\ z] & = & \mathit{eval}\ x\ ;\ \lambda v \rightarrow \\ & & \mathit{apply}\ v\ y\ ;\ \lambda u \rightarrow \\ & & \mathit{apply}\ u\ z \end{array}$$

Note that the result of the first *apply*, the node value in the variable "u", must already represent a whnf, since functions in GRIN always return values representing whnfs, so there is no need to call *eval* again between the two calls to *apply*. This is also the reason that the call to *eval* was not put first inside the body of the *apply* procedure itself, the *eval* is not always necessary.

The general case, with many arguments is handled analogously, creating one extra call to *apply* for each additional argument.

Too many arguments. A slight variant of the above is when a *known* function is applied to too many arguments (more than its arity). In this case we first do an ordinary call, using as many arguments as the function arity prescribes. From this call a value is returned which we know must represent a partial application (a P-node), otherwise the original functional program would not have been type correct. So, we can *apply* the result of the first call to the rest of the arguments. Imagine a known function *foo*, of arity two, applied to three arguments. This would result in the following GRIN code:

$$\begin{array}{lcl} \mathcal{E}\ [\mathit{foo}\ x\ y\ z] & = & \mathit{foo}\ x\ y\ ;\ \lambda v \rightarrow \\ & & \mathit{apply}\ v\ z \end{array}$$

2.6 Related work

This section will discuss some related work in addition to what has already been mentioned. Here, we will mostly discuss work that is directly related to

GRIN, i.e., intermediate languages and abstract machines used to implement lazy functional languages via *graph reduction*.

2.6.1 Intermediate languages for graph reduction

The earliest implementations of functional languages with non-strict evaluation were in the form of *interpreters*, performing *normal order reduction* of λ-calculus terms, often using variations of Landin's SECD machine [Lan64]. The term *graph reduction* was introduced by Wadsworth [Wad71], and *lazy evaluation* by Henderson and Morris [HM76]. Other early attempts were Burge [Bur75], and Turner [Tur75] with his SASL implementation.

In [Tur79] Turner took a different approach and first translated into a small set of *combinators* (from combinatory logic), which then were interpreted.

Hughes attacked some of the efficiency problems of Turner's combinator method and showed that any program could be transformed into a set of specialised combinators, called *super combinators* [Hug82]. One of Hughes main issues were the notion of *full laziness*. Johnsson instead concentrated on the efficient implementation of combinator code and devised an algorithm, called *lambda lifting* [Joh85], for transforming a program into super combinators.

Johnsson also created an abstract machine called the *G-machine* [Joh84, Joh87b], that performed efficient graph reduction. Together with a translation method for combinator code into G-machine code this was the start of a new revolutionary implementation technique, called *compiled graph reduction*. The G-machine was refined by Augustsson [Aug86], and implemented by Augustsson and Johnsson in the well-known Chalmers Lazy-ML compiler [Aug84, Aug87, AJ89c].

With compiled graph reduction it became possible to apply many optimisations (from conventional compiler technology) that had previously (in implementations based on interpreters) not been possible [Joh81, Joh86].

A variant of the G-machine, called the *Spineless G-machine*, was created by Burn, Robson and Peyton Jones [BRJ88]. The idea was to avoid creating long chains of so-called *apply nodes* that the original G-machine created when building closures. Also, a method was proposed to avoid *updates* for non-shared redexes. Similar ideas were also part of the $\langle \nu, G \rangle$-machine by Augustsson and Johnsson [AJ89b], which in addition was a parallel implementation of graph reduction on a shared memory multiprocessor system.

Independently of the G-machine, Fairbairn and Wray developed the *Three Instruction Machine* (TIM) [FW87]. In TIM, closures with references to free variables could be represented, thereby eliminating the need to do lambda lifting. Another abstract machine, called the *ABC machine*, was developed by Smetsers *et al.*, and successfully used in the *Clean* compiler [SNvGP91].

The next step in the G-machine tradition was the *Spineless Tagless G-machine* (STG machine), by Peyton Jones and Salkild [PJS89, PJ92]. Here, the concept of *tag*, used to identify heap objects in the original G-machine,

was eliminated and a *code pointer* used instead. This meant that to evaluate a closure (called a *thunk*), a simple (indirect) call through the code pointer was needed (called to *enter* a thunk). Or more accurately, *all* kinds of objects were represented with code pointers, and thus could be entered in a uniform way, resulting in a value on weak head normal form. At first, the STG machine was thought to be a totally different approach compared to the original G-machine, but it was later realised that the two methods were actually not that different in practice. The actual machine code resulting from the two was quite similar, with both approaches resulting in an indirect call through a code pointer to implement evaluation. For fairness sake it should be said that the STG machine were a lot better than the original G-machine on some things, e.g., efficient handling of tailcalls to (compile time) unknown functions. But common to the two approaches were still the indirect call for evaluation, which is one of the main things that we try to attack with our GRIN approach.

Many of the abstract machines presented sofar worked in a similar way, translating to a sequence of abstract machine instructions (a simple imperative code). This code could then be optimised using conventional compiler technology. The STG machine changed this in that its intermediate code were a completely functional language, i.e., on a much higher level compared to the earlier abstract machine approaches. Details such as how evaluation is done, and how updates are performed are not directly visible in the STG language. The details are hidden in the implementation of the STG machine.

In the Glasgow Haskell Compiler, the STG language is used together with a *Core* language. Both these are essentially 2:nd order λ-calculus. The STG language is on a slightly lower level. Compared to Core and STG our GRIN language is a much more low level code. A typical example of the difference is that we, in GRIN, can "inspect" a node value representing a closure without evaluating it. This is not possible in the STG language. On the other hand, a weak point of GRIN may be its lack of a typing. STG and Core are explicitly typed languages, something which is valuable when building a compiler, e.g., to help in verifying that program transformations are correct.

The FAST compiler [HGW91b, HGW91a], by Hartel, Glaser and Wild translates into an intermediate language called "Functional C", which can be compiled with a normal C compiler (see also the discussion in section 6.4.4 on page 244).

By using explicit language constructs to build and evaluate suspended computations (sometimes called *force* and *delay*) it is possible to build an implementation of a non-strict functional language on top of a strict functional language. An example of this is the Yale Haskell compiler [PH93], which is built on top of *Common Lisp* [Ste84]. An interesting problem with such an approach is how to handle side-effecting updates, i.e., how to get call-by-need rather than call-by-name. This is discussed in [OLT94] which also shows a direct translation of a lazy language into CPS (see below).

Another example of explicit building of closures is used by Faxén [Fax95a, Fax97], with his intermediate language FLEET (Functional Language With Ex-

plicit Evals and Thunks).

2.6.2 Other intermediate languages

Although our primary concern is ways to implement lazy functional languages, it is also interesting to examine intermediate languages used to implement other kinds of languages. In particular, implementations of strict functional languages are interesting because many similar problems arise in that context, e.g., the need to build closures to represent higher order functions. Strict functional languages is traditionally implemented using a transformation of the program into *Continuation Passing Style* (CPS) [Ste78, AJ89a]. The relationship between GRIN and CPS is not trivial to characterise. Both approaches are intermediate languages that are relatively high level (compared to machine code), but at the same time provides enough "low level" control. CPS achieves many benefits due to its use of continuations to represent all control flow. In GRIN, on the other hand, we want to avoid creating many small functions (continuations) and use a more conventional "call-return" model. In a sense the difference can be compared to the two corresponding programming styles: to program with monads or with continuations, which is probably mostly a matter of taste.

An influential abstract machine used to implement strict evaluation is the *Categorical Abstract Machine* [CCM85].

Recently, so-called *Typed Intermediate Languages* (TILs) have received much interest. These languages are often based on some version of higher order λ-calculus (compare with the STG language above). Some examples are [SA95, TMC+96, PJLST98].

Although not closely related as an intermediate code, it is still interesting to relate GRIN to *Static Single Assignment* (SSA) code [AWZ88, CFR+91], which has become very popular among implementors of imperative languages. Both GRIN and SSA are functional representations and handle the single assignment in similar ways (see also the discussion in section 2.3.7 on page 34).

The technique to use unique tags combined with *dynamic dispatch* in a first order language to implement closures, both higher order functions and the laziness (our *apply* and *eval* functions), is originally due to Reynolds [Rey72] (recently also in [Rey98b, Rey98a]). The idea has later been used and refined by many others [CD96, Tol97, BBH97, TO99].

Chapter 3

Control flow analysis

In this chapter we will describe a program analysis for GRIN programs, developed by Thomas Johnsson [Joh91, BJ96]. This analysis, called *heap points-to analysis*, forms a large part of the foundations for all the work on using the GRIN language as an intermediate compiler language. Especially the *interprocedural* optimisations done by the GRIN back-end rely heavily on the results of this analysis. Without the use of this analysis, the GRIN language and our optimisations would need to be changed, but most of the work would probably still be possible in some variant.

The analysis is called a "heap points-to" analysis, because that is exactly what it does, it analyses what pointers into the heap points to. However, as a "side effect" of analysing pointers (and approximating heap contents) in a graph reduction language also we also get what is normally called *control flow information*. Recall the special *eval* procedure (see figure 2.8 on page 45). Apart from all values representing weak head normal forms, *eval* also enumerates all possible closures that can appear in the program (or really, it enumerates values that represent closures, there are no closures in the GRIN program itself, see section 2.3.4 on page 32). When a value that represents a closure is found, the corresponding function is called. However, for each particular use of *eval* in a program, the actual closures that can arise at that particular point are in most cases just a small subset of all the closures in the program. Finding this information, i.e., a safe subset of closures for each *eval* point, is exactly what the heap points-to analysis does. Since the set of closures found for a certain *eval* points approximates what actual function calls that can arise from that call to *eval*, the analysis is really also a *control flow analysis*.

Before we show the analysis, we will discuss control flow in the context of graph reduction. We do this to show how important, and difficult to find, control flow information is when compiling a lazy functional language. In particular, we will discuss *function calls* and try to answer questions like: "what function calls take place?" and "who is calling who?". We will call such information *evaluation*

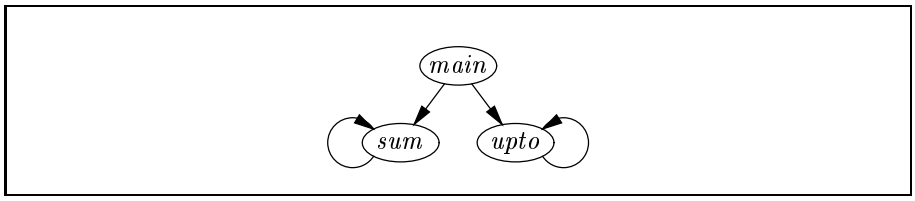

Figure 3.1: A "strict" call graph.

order information.

Having a good approximation of the evaluation order is very important for some of the optimisations that are done by the GRIN back-end, in particular the *interprocedural register allocation* (see chapter 6), so we will go into some depth discussing this issue.

3.1 Graph reduction and evaluation order

Evaluation order in a non-strict language can be very hard to predict. In many cases the actual evaluation order that arise when a program is executed is completely different from what one might expect by just looking at the (functional) source code. The reason is, of course, that in a non-strict evaluation (or a *normal order reduction*), a function application in the "source code" does not necessarily imply an immediate function call. When in a non-strict context, an application will be suspended and the actual call "happens" later when the value of the suspended application is needed (if it is needed at all, otherwise the call will never take place).

3.1.1 Strict evaluation order

To be a bit more concrete, let us go back to the example program in figure 2.7 on page 44. If we do not worry about evaluation order and simply look at all function applications appearing in the program, we can draw the *call graph* in figure 3.1.

We define the call graph to be the union of all possible executions of a program. Each node in the call graph represents a procedure, and an arrow between two nodes, say from A to B, means that A may (in some possible execution) make a function call to B.

The effect of considering all function applications in the original, functional source code (and nothing else), is that the call graph in figure 3.1 really is a "strict" call graph, i.e., the call graph we would get from an execution of the program in a strict functional language (like Standard ML). In a strict execution, the *main* function will start by calling *upto*, to produce a list of numbers. The

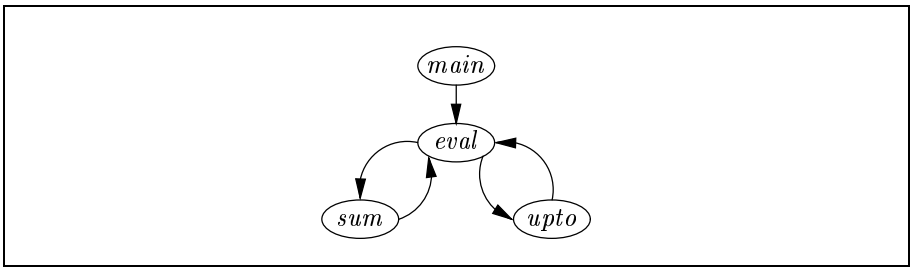

Figure 3.2: A "lazy" call graph.

upto function will generate the complete list, using recursion (calling itself), and then return to *main* (returns are not visible in the call graphs we use). After that, *main* will pass the list of numbers to *sum*, which will compute the total sum, again using recursion.

The correspondence between what happens during the execution and the call graph is actually quite close. Although the "order" of function calls or the "number" of calls is not visible in the call graph, quite a bit of useful information (for a compiler) can be deduced from this call graph. E.g., the fact that both the *upto* and the *sum* function are self-recursive, but call no other functions.

3.1.2 Non-strict evaluation order

However, if we instead look at a non-strict (or lazy) evaluation of the program, the actual function calls will be completely different from the above! A non-strict evaluation is encoded by the GRIN version of the same program, shown in figure 2.8 on page 45. If we use the same technique, to take all function applications in the GRIN program (including *eval* of course, it is a normal function), we would get the call graph in figure 3.2.

Here, we can see that *eval* plays a central role in the execution. All functions (except *main*) have arrows both to and from *eval*. An arrow *to eval* means that there was a need to evaluate something (possibly a closure). An arrow *from eval*, on the other hand, means that when *eval* examines its argument and finds a closure (an F-node) representing some suspended function application, it will make a function call to the corresponding function.

Another thing to note about the "lazy" call graph is that the direct recursion that was visible in the "strict" version, in both *upto* and *sum*, now has disappeared.

In our example there are only two functions, besides *main* and *eval*, but for larger programs the call graph would be even more dominated by the *eval* procedure, sitting like a spider in the middle of a web. Most procedures would have arrows to and from *eval*, and only a few would make direct calls to other

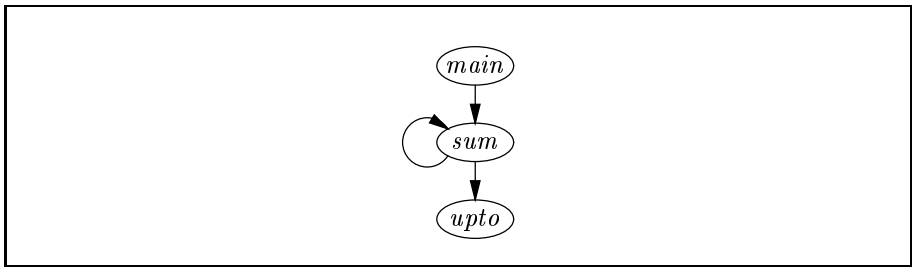

Figure 3.3: An improved "lazy" call graph.

procedures, calls that were found in a strict context.

As said earlier, some optimisations really benefit from a good approximation of the "real" function call behaviour of the program. However, having a call graph like the above, where most function calls are "indirect" through *eval* is virtually useless. The only thing it tells us about the evaluation order is that all functions call *eval* and that all functions can be called by *eval*. This information is in most cases a very crude approximation of the "real" evaluation order, that will actually happen in an execution of the program. Remember that all call graphs are approximations, but the "lazy" one is really far too inexact to be useful for our purposes, to do interprocedural optimisation.

For this simple program, a much more restricted (but still correct) call graph can be drawn. Figure 3.3 describes exactly the set of calls that will happen (in the only possible execution that exists for this simple program). In this version of the call graph, all the calls to *eval* have been "short-circuited", i.e., have been replaced by direct calls to only those procedures that were "possible" for each call to *eval* in the program.

Of course, this improved call graph does *not* reflect the original GRIN code for our example program. The improved call graph reflects a program where all calls to the *eval* procedure have been eliminated and replaced with direct calls. However, using the information derived by the heap points-to analysis (see below) we will be able to automatically transform the original GRIN program into an equivalent program, without *eval*, that have exactly the call graph in figure 3.3. The way this is done is described in detail in section 4.2.1 on page 84, *inlining calls to eval*.

In this "improved" lazy call graph, the graph reduction nature of the code has been made more explicit. The real "loop" in the program is now visible, in the *sum* function. For each iteration, *sum* will call *upto* once, to deliver the next number (although this is not exactly visible in the call graph). The *upto* function itself does not do any function calls at all, which is clearly visible in the call graph.

The above example also makes clear the *producer/consumer* relationship

(between *upto* and *sum*) that is so typical of lazy evaluation. In this example, the "loop" of the program actually consists of the *sum* and the *upto* functions together, and since *upto* now is non-recursive we could actually benefit a lot by *inlining* it into *sum* (see the inlining transformation in section 4.3.10 on page 135, and also the GRIN transformation example in appendix A on page 301).

However, even without inlining *upto*, when used to guide interprocedural optimisations the call graph in figure 3.3 will always be an improvement over the one in figure 3.2. In general, the better the call graph approximates the "real" behaviour of the program the better result we can expect from interprocedural optimisations.

A larger example of this is shown in section 3.3 on page 71.

3.2 The heap points-to analysis

The analysis that we will use to determine control flow information, the *heap points-to analysis*, was developed by Thomas Johnsson. The analysis was first described in [Joh91] and more recently in [BJ96] (which also describes a much more efficient version than the original).

In this thesis we will only give an overview of the analysis.[1] We will concentrate on the usage of the analysis and on the analysis results. For a more elaborate description of the analysis we refer to [BJ96].

The way the analysis fits into the GRIN part of the back-end is shown in figure 2.9 on page 49. See also figure 1.3 on page 17 for an overview of the complete compiler.

3.2.1 The analysis result

The main aim of the analysis is to:

> *determine, for each call to eval, a safe approximation to what different node values (or rather tags) that eval might find when it fetches a node from the heap via its argument pointer.*

Note that the GRIN value domain can represent both "ordinary values" and closures (called F-nodes). This means that the result of the analysis predicts what closures can appear at each *eval* point, i.e., it predicts the actual function calls that can result from each call to *eval*. In other words, the heap points-to analysis is also a *control flow analysis*.

The analysis is implemented as an *abstract interpretation* [CC77], i.e., it computes an interpretation of the program using an *abstract semantics* of GRIN code. The analysis currently requires the entire program at once.

[1] It should be noted that the author of this thesis has not, in any significant way, taken part of the development of the analysis. However, since the use of the analysis is such an important part of the GRIN framework we believe that it must still be described here.

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

Figure 3.4: An abstract store.

The abstract store

The main difference between the concrete semantics and the abstract semantics is that in the abstract semantics all heap locations are abstracted to a bounded set of abstract locations $\{\,1,2,\ldots,\mathit{maxloc}\,\}$, where maxloc is the total number of store operations appearing in the program (recall that a store operation allocates a new heap location, see section 2.4.5 on page 41). In the abstract semantics, each store operation in the program will use the same abstract heap location each time it is executed, and hence, each abstract location will contain a *set* of abstract node values (all nodes ever written to that location, either by a store or an update). Similarly to in the concrete semantics, the analysis uses an abstract value domain consisting of basic values (all denoted by the single value BAS), abstract locations (pointers) and abstract node values.

In figure 3.4 we show an example of an abstract heap (called *store* in the analysis), that the analysis might derive for our running example, the GRIN program in figure 2.8 on page 45 (the actual end result will have better precision than this, using sharing analysis; see section 3.2.2 on the facing page).

The six store operations in the program have been numbered with the abstract locations 1 to 6, from top to bottom in the program. For each abstract location we can see that it always contains the tag that appeared in the actual store operation (remember that tags in GRIN store operations are always known tags, see section 2.4.5 on page 41). However, the abstract location can also contain tags that was written by an update operation, if the location initially contained a closure and was later the subject of a call to *eval*. In the abstract store we can also see that the "arguments" of abstract node values are themselves *sets* of abstract values, either abstract locations or basic values.

The abstract environment

In addition to the abstract store, the analysis will determine an *abstract environment*, mapping all variables to a set of abstract values. For our example program the analysis will produce the environment in figure 3.5 on the facing page.

$t_1 \mapsto \{1\}$	$m' \mapsto \{BAS\}$	$u \mapsto \{\mathsf{CNil}\,[], \mathsf{CCons}\,[\{1,5\}, \{6\}]\}$
$t_2 \mapsto \{2\}$	$u' \mapsto \{BAS\}$	$t \mapsto \{1,5\}$
$t_3 \mapsto \{3\}$	$b' \mapsto \{BAS\}$	$ts \mapsto \{6\}$
$t_4 \mapsto \{4\}$	$x' \mapsto \{BAS\}$	$t' \mapsto \{BAS\}$
$i' \mapsto \{BAS\}$	$t_5 \mapsto \{5\}$	$r' \mapsto \{BAS\}$
$m \mapsto \{1,5\}$	$t_6 \mapsto \{6\}$	$s' \mapsto \{BAS\}$
$n \mapsto \{2\}$	$l \mapsto \{3,6\}$	

Figure 3.5: An abstract environment.

In the abstract environment we can see how the variables that bind the result of the store operations $(t_1, t_2, \ldots, t_6)$ are bound to the corresponding abstract locations $(\{1\}, \{2\}, \ldots, \{6\})$. The node variable "$u$" is bound to an abstract node value, which is either a CNil or a CCons node, and all basic value variables are bound to *BAS*.

Let us for a moment consider the abstract value of the first argument of the *upto* function, the variable "m" (see figure 2.8 on page 45). Its value in the abstract environment is the set $\{1,5\}$, i.e., the value comes from either the store numbered 1 (the first in *main*) or the store of a CInt node later in *upto* itself (part of building the next *upto* closure). Furthermore, the second argument to *upto*, the variable "n", has the abstract location 2 as only possible value (corresponding to the second store in *main*). This is the "upper limit" of *upto* which is stored once and for all in *main* and then examined repeatedly by the *upto* function.

In the actual implementation of the analysis, the special *eval* procedure does not take part of the analysis. This explains why the variables used inside the body of *eval* do not appear in the abstract environment.

3.2.2 Sharing analysis

It has turned out to be quite easy to incorporate a *sharing analysis* into the heap points-to analysis. This is good for several reasons, it can be used for *update avoidance* (see section 4.2.1 on page 84) but can also be used to improve the precision of the heap points-to analysis itself. The sharing information will determine, for each abstract location, whether any concrete instance of the abstract location might be *shared* or if it is guaranteed to be *unique*.

When the combined analysis is run on our example program it will result in the abstract store shown in figure 3.6 on the next page. Compare this to the previous version, done without sharing analysis (figure 3.4 on the facing page).

In addition to giving sharing information (the shared and unique attributes), we can see how the use of sharing information internally in the analysis have

$$
\begin{array}{ll}
1 \mapsto \{\mathsf{CInt}\,[\,\{\,BAS\,\}\,]\,\} & \textit{shared} \\
2 \mapsto \{\mathsf{CInt}\,[\,\{\,BAS\,\}\,]\,\} & \textit{shared} \\
3 \mapsto \{\mathsf{Fupto}\,[\,\{\,1\,\},\{\,2\,\}\,] & \textit{unique} \\
4 \mapsto \{\mathsf{Fsum}\,[\,\{\,3\,\}\,] & \textit{unique} \\
5 \mapsto \{\mathsf{CInt}\,[\,\{\,BAS\,\}\,]\,\} & \textit{shared} \\
6 \mapsto \{\mathsf{Fupto}\,[\,\{\,5\,\},\{\,2\,\}\,] & \textit{unique}
\end{array}
$$

Figure 3.6: Improved abstract store (with sharing analysis).

actually improved the final result. E.g., the abstract locations 3 and 6 are now known to be Fupto nodes rather than being a set of three possible nodes. Although *eval* might update the concrete locations with either a CNil or a CCons node this will never be "visible" to the "readers" of that location. Such information will be used later to optimise the GRIN program (see section 4.3.9 on page 133).

3.2.3 The analysis machinery

As mentioned earlier, we refer the reader to [BJ96] for a more in-depth description of the heap points-to analysis itself. We will here only note a few things.

An overriding design goal for the analysis was that it had to be very fast in order to be able to analyse large entire programs at once. To accomplish this, some precision of the analysis had to be sacrificed. To summarise:

- the analysis is flow insensitive: a single abstract heap approximates the concrete heap, at all times and at all program points,
- the analysis is insensitive to calling context: a procedure parameter gets its abstract value by taking the union of the actual parameters at all call sites, and the same abstract return value is returned as a result of all calls to a procedure.

3.2.4 Using the analysis result

The way we use the result of the heap points-to analysis is described in the chapter on GRIN program transformations (chapter 4) and in particular in section 4.2.1 on page 84, *inlining calls to eval*.

Returning to the call graphs discussed earlier, by running the analysis on our example program, and then inline all calls to *eval* using the results of the analysis, we will get a program with a call graph exactly as the improved "lazy" call graph in figure 3.3 on page 66.

3.3 A larger example

To really show the capabilities of the heap points-to analysis as a control flow analysis, we will show a slightly larger example. The program in figure 3.7 on the following page solves the well known *eight queens* problem (the result of the program is the number of solutions found). The program was originally written to use higher order functions, but have been rewritten and *specialised* by hand to include only first order functions. As a result, the program contains calls to *eval* but no calls to *apply*, which will make this example simpler.

A translation of the program to GRIN would be very straightforward, and we refrain from giving it here. We will however show two call graphs, in the same style as for the previous example (see figures 3.2 and 3.3). We do this to show how the result of the heap points-to analysis can be used to predict the "real" control flow of a GRIN program.

Figure 3.8 on page 73 shows the call graph for a straightforward translation of the queens program into GRIN. The *eval* procedure is sitting as a "spider" in the middle of its web, like in figure 3.2, but here it is even more apparent. Most procedures have arrows both to and from *eval*. Remember that an arrow *to eval* is what we have termed an "unknown" call. E.g., when the *append* function calls *eval* it is not possible to say what other function calls this will result in (what arrows *from eval* that can be taken in the next step).

If we were to remove all arrows to and from *eval* from the call graph, we would be left with only the direct function calls (some *tailcalls* and some ordinary returning calls), but they would not be very many in this example. Most of the calls in the program are what we call "unknown".

If we run the heap points-to analysis on the queens program, and then use the techniques described in section 4.2.1 on page 84 to eliminate the *eval* procedure, we will get a program with the call graph in figure 3.9 on page 74.

In this improved call graph, all "unknown" calls have been replaced by direct calls. In the general case the analysis will replace each unknown call with a "set" of direct calls. However, by looking at the improved call graph we can see that this "set" in most cases is very small. In fact, the analysis has exactly determined the actual control flow of the program, i.e., there are no function calls visible in the improved call graph that will not take place during a real execution of the program.

The queens example is still a very small program, but we hope that this method will scale to larger programs, although we have no proofs of that yet. The fact that we analysed a first order version of the queens program in this example is not very important, the result would have been similar for a higher order program, i.e., the improved call graph would have been much "better" than the original call graph.

As we will later see, the information about function calls contained in the improved call graph will be of great help during some of the interprocedural optimisations, and most notably the interprocedural register allocation (see chapter 6).

```
main = solve 8

solve nq = length (generate nq nq)

generate nq 0 = [[]]
generate nq n = concatMapAddOne nq (generate nq (n − 1))

concatMapAddOne nq [] = []
concatMapAddOne nq (x : xs) = append (addOne nq x)
                                     (concatMapAddOne nq xs)

addOne nq xs = filterOk (mapCons xs (upto 1 nq))

filterOk [] = []
filterOk (x : xs) = if ok x then
                      x : filterOk xs
                    else
                      filterOk xs

mapCons xs [] = []
mapCons xs (y : ys) = (y : xs) : mapCons xs ys

ok [] = True
ok (x : xs) = safe x 1 xs

safe x d [] = True
safe x d (y : ys) = (x ≠ y) ∧ (x ≠ y + d) ∧ (x ≠ y − d) ∧ safe x (d + 1) ys

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

append [] ys = ys
append (x : xs) ys = x : append xs ys

length [] = 0
length (x : xs) = 1 + length xs
```

Figure 3.7: The eight queens program.

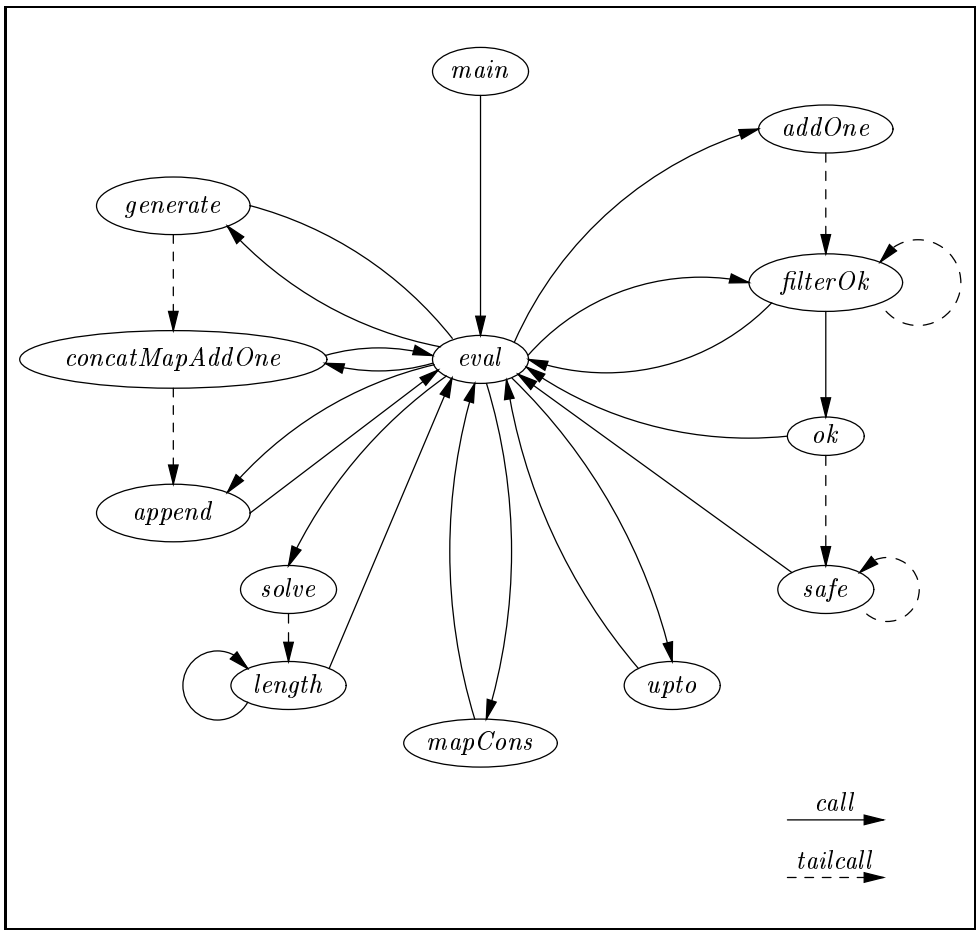

Figure 3.8: The original queens call graph.

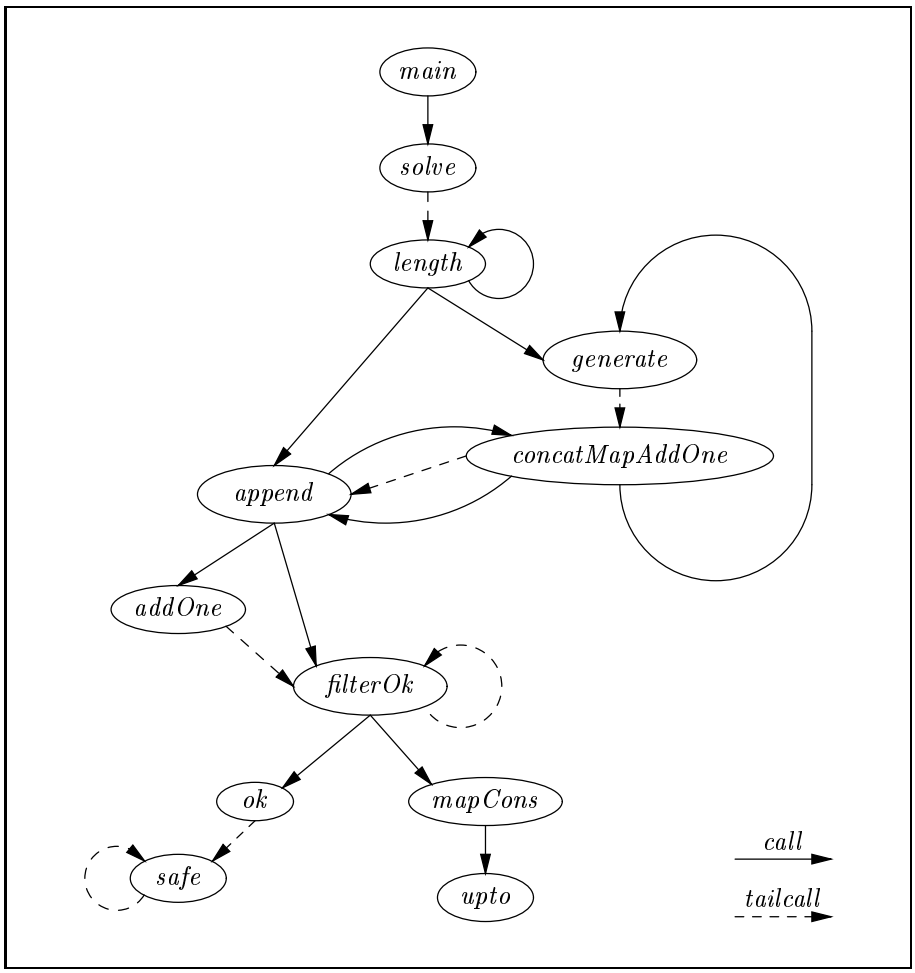

Figure 3.9: The improved queens call graph.

3.4 Related work

A great deal of work has been done on points-to analysis of more conventional languages, e.g., see the overview in [SH97]. For a lazy language, a *heap* points-to analysis will also be a *flow* analysis, or rather, we will get control flow information as a side effect of a heap points-to analysis (due to closures being written to and read from the heap).

The work by Karl-Filip Faxén [Fax95a, Fax95b] is quite similar in scope to ours, and he addresses many of the same problems as we do. Somewhat related is also Bloss' *path analysis* [Blo94].

For *Scheme*, the various control flow analyses by Shivers [Shi88] provide similar information as the analysis we use.

Chapter 4

Program transformations

In this chapter we will show how GRIN code can be optimised using program transformation techniques. The main idea is to use a large number of GRIN source-to-source transformations, each very simple, but which taken together produce highly optimised code. The transformations are repeatedly applied by a *transformation engine*. Some of the transformation are no more than "GRIN versions" of well known conventional optimisations, whereas others are more novel and special to the GRIN framework.

4.1 Overview

The GRIN program transformations can be divided in two main categories:

- *simplifying* transformations,
- *optimising* transformations.

The set of *simplifying* transformations should be seen as a variant of *compilation by transformation* [KH89]. The simplifying transformations will turn "high level" GRIN code into "low level" GRIN code. As explained in chapter 2, some constructs in the GRIN language exists in two variants, a "high level" and a "low level" variant. A typical example is the fetch operation, it can either load an entire node value from the heap in one step (a high level operation), or it can load only a single component of a node value (a more low level operation). In the GRIN back-end, the GRIN code generator will emit "high level" GRIN code which is then transformed by the simplifying transformations until a very simple form of GRIN is reached. The final GRIN code is then fed into the RISC code generator (see chapter 5). This has two consequences: first, the RISC code generator becomes simpler since is does not have to handle all possible GRIN code. Second, it means that the simplifying transformations become "necessary", the GRIN back-end will not work without them.

The *optimising* transformations, on the other hand, are *not* necessary in that sense, they are just "normal" optimisations that can be turned on or off. We can choose to either include or not include each optimising transformation, we can run some of them several times, etc. The way this is done will be described in more detail below. The GRIN back-end will always be able to compile a program, independently of what optimising transformations are turned on. Hopefully, though, the quality of the generated code will be better when the optimisations are turned on!

A complete transformation example illustrating both simplifying and optimising transformations can be seen in appendix A on page 301.

4.1.1 Transformation strategy

The general strategy behind the GRIN transformation machinery (and the optimising transformations in particular) is to use a large number of, each very simple, GRIN source-to-source program transformations. Every transformation is of course correctness-preserving and hopefully performance-improving. Even though each single transformation in most cases will make only very small changes to the program, the transformations taken together will produce greatly simplified and optimised GRIN code.

The idea to repeatedly apply a large number of transformations has been used in many compilers before, e.g., in the *simplifier* in the Glasgow Haskell Compiler [PJ96, PJHH+93].

The various GRIN transformations that we will describe below should not be seen as a complete set of "GRIN versions" of all standard program transformations (for functional and other languages). Instead, the transformations have arisen on a "demand basis" during the work on the GRIN back-end. By inspecting a lot of GRIN code we have found places where improvements have been possible, and tried to adapt some conventional transformation to fix it. If that has not been possible we have invented a new transformation, specific to the needs of the GRIN back-end.

Locality

Many of the transformations are very "local" in the sense that they will try to find a small sub-expression of a GRIN expression, matching a certain pattern, and if found exchange it for a slightly different sub-expression. Such transformations can easily be implemented as a single "walk" over the expression tree.

Other transformations are a bit more involved, and we will try to describe for each transformation below how "hard" it is to implement. Remember also that we assume that the front-end has already transformed the program as much as possible on the "functional" level. The current front-end, *hbcc*, does indeed implement many standard transformations (see also the compiler overview in section 1.4.1 on page 16).

Using traditional compiler terminology, most of the transformations we will present are either *local* or *global* [ASU86, section 10.2] (these terms are also defined in section 1.3 on page 10). The above "pattern matching" transformations would traditionally be called local, or possibly global, depending on if the patterns used involve GRIN constructs that can change the *control flow* (like case expressions). If they do, it could be termed a global optimisation, otherwise it is local. In some cases a transformation may even be *interprocedural* (program-wide). However, in most such cases the "interprocedural part" consists only of collecting information (in linear time) from the entire program. This information is then used by a global transformation (done on one procedure at a time). Note that this is not problematic to implement in the GRIN framework, since the GRIN back-end relies on having the entire program available. However, in the current situation we do not make very much use of these opportunities for program-wide optimisation on the GRIN level, it would be possible to do much more.

Visualisation

Most of the transformations are visualised using GRIN "code skeletons", showing some GRIN code before and after a transformation is applied. Each transformation tries to specify the GRIN code only as much as is necessary for the transformation to "match", and uses *meta expressions* to depict arbitrary GRIN expressions. Meta expressions are written as "$\langle m_0 \rangle$", "$\langle m_1 \rangle$", etc. For an example, see figure 4.6 on page 93. For simplicity we will sometimes also use the same notation for things that are not really syntactically correct GRIN expressions, but instead are expressions with "holes" in them (sometimes called *contexts*). In figure 4.6 only "$\langle m_2 \rangle$" and "$\langle m_3 \rangle$" are syntactically correct GRIN expressions, both "$\langle m_0 \rangle$" and "$\langle m_1 \rangle$" are expressions with holes. The reason is that in the GRIN syntax an expression can not "end" with a lambda binding, something which both "$\langle m_0 \rangle$" and "$\langle m_1 \rangle$" must do (see the syntax in figure 2.1 on page 28).

Confluence

As soon as transformations are mixed, the issue of *confluence* becomes important. In the GRIN transformation system the optimising transformations are the most interesting from the confluence perspective, because only the optimisations are repeated (the simplifying transformations are run only once, see below). Although we have not proved it, we believe the GRIN transformation system to be confluent. In fact, the nature of the GRIN optimising transformation seems to prevent transformations from "undoing" the effect of each other, something which would break confluence. The only potential problem that we are aware of is a slight interaction between *register introduction*, *copy propagation*, *common sub-expression elimination* and *constant propagation*. A problem which is easily solved by a slight restriction on the copy propagation. This is

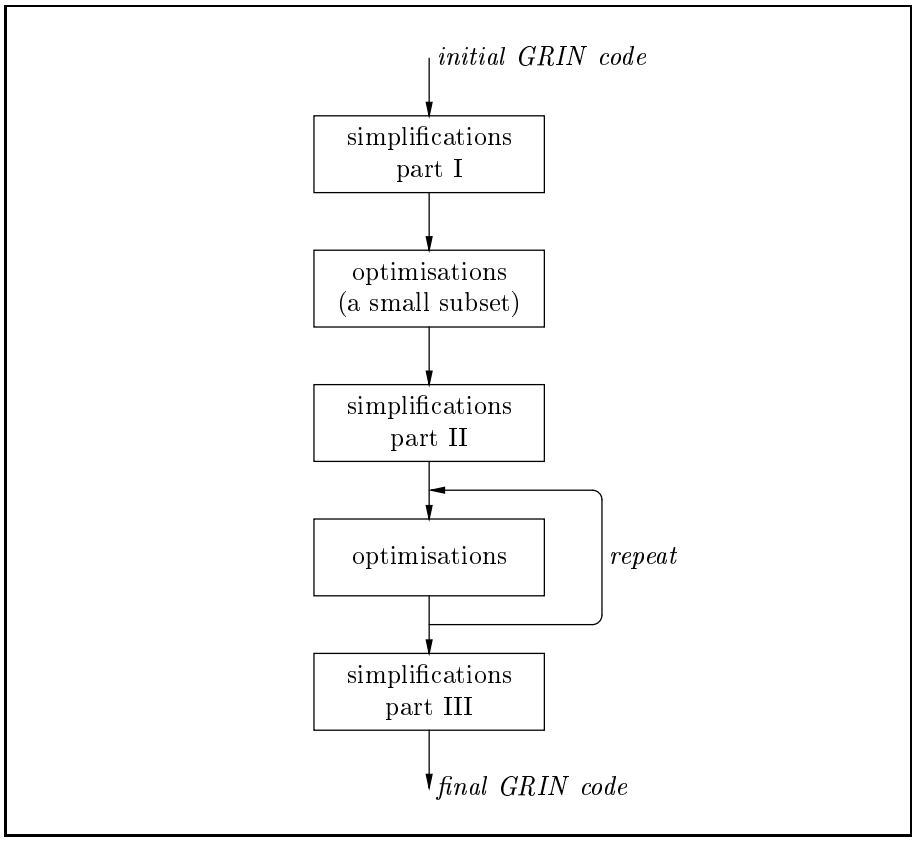

Figure 4.1: Overview of the GRIN program transformation process.

discussed more in section 4.3.2 on page 117.

4.1.2 Summary of transformations

The way in which transformations are applied differ for the two transformation categories, the simplifying and the optimising transformations. An overview of the main ideas of the transformation process is shown in figure 4.1. Compare this with figure 2.9 on page 49 that shows the organisation of the GRIN part of the back-end, and with figure 1.3 on page 17 that shows the complete compiler.

A complete transformation example, illustrating many (but not all) of the transformations is shown in appendix A on page 301. This example also shows one particularly interesting aspect of the GRIN transformation system, namely how some of the GRIN transformations, when working together, can achieve ef-

fects similar to *deforestation* [Wad88], leading to greatly enhanced performance.

Simplifying transformations

The simplifying transformations form the core of the "compilation by transformation" process. Each simplification is applied *only once*, and we apply them in a certain predetermined order. As can be seen in figure 4.1 the simplifications is what "guides" the whole transformation process, with optimisations run "in between". Each simplifying transformation belongs to exactly one simplification *phase* (the three boxes labelled "simplifications" in figure 4.1).

A summary of the simplifications in each phase is given here. Each separate transformation is then described in detail in section 4.2 starting on page 84. The phase labelled "simplifications part I" consists of the following:

- *inlining calls to eval*: unfold all calls to the *eval* procedure (see section 4.2.1 on page 84),
- *inlining calls to apply*: unfold all calls to the *apply* procedure (see section 4.2.2 on page 89).

The phase labelled "simplifications part II" consists of the following:

- *update specialisation*: specialise all uses of the GRIN `update` operation (see section 4.2.3 on page 90),
- *vectorisation*: replace all node variables with explicit nodes with a known maximum size (see section 4.2.4 on page 97),
- *case simplification*: replace case tests on entire node values with tests on tags only, and eliminate all binding done by case expressions (see section 4.2.5 on page 100).

The phase labelled "simplifications part III" consists of the following:

- *compilation of conditionals*: replace some conditionals (primitive operations returning a boolean value) with if expressions (see section 4.2.6 on page 101),
- *split fetch operations*: replace node valued GRIN `fetch` operations with sequences of simpler `fetch` operations (see section 4.2.7 on page 102),
- *right hoist fetch operations*: move some `fetch` operations inside case expression alternatives (see section 4.2.8 on page 107),
- *register introduction*: give all temporary values a GRIN variable name (see section 4.2.9 on page 110).

Optimising transformations

The optimising transformations are, with one exception, all run together in a single phase (the box labelled "optimisations" in figure 4.1). The optimisations are repeated until the program does not change anymore (see also above, the discussion about *confluence*).

As for the simplifications, we give only a short summary of the optimisations here, and describe each optimisation in detail later (section 4.3 starting on page 112). The optimisations are:

- *copy propagation*: eliminate unnecessary "copies" (see section 4.3.2 on page 113),
- *generalised unboxing*: a slightly more general form of traditional *unboxing* (see section 4.3.3 on page 118),
- *evaluated case elimination*: eliminate certain unnecessary case expressions that were introduced by the *eval* inlining (see section 4.3.4 on page 125),
- *trivial case elimination*: eliminate case expressions with only one alternative (see section 4.3.5 on page 126),
- *sparse case optimisation*: delete case alternatives that can never be reached (see section 4.3.6 on page 127),
- *case copy propagation*: a more general version of copy propagation for case expressions (see section 4.3.7 on page 128),
- *update elimination*: remove unnecessary GRIN `update` operations (see section 4.3.8 on page 132),
- *whnf update elimination*: optimise the placement of `update` operations to avoid updating weak head normal forms (see section 4.3.9 on page 133),
- *late inlining*: ordinary function inlining, but applied "late", after the elimination of unknown control flow (see section 4.3.10 on page 135),
- *case hoisting*: merge sequences of case expressions, to avoid some case tests and open up for constant propagation (see section 4.3.11 on page 137),
- *constant propagation*: eliminate known constant computations (including tests) (see section 4.3.12 on page 143),
- *arity raising*: a generalised form of traditional *arity raising* (that can also "raise" non-strict arguments) (see section 4.3.13 on page 144),
- *common sub-expression elimination*: avoid doing the same computation more than once (see section 4.3.14 on page 148),

- *dead code elimination*: remove unused functions and computations whose result is never used (see section 4.3.15 on page 153),

- *dead parameter elimination*: remove unused parameters from functions (see section 4.3.16 on page 155).

As we said earlier, all these transformations are run repeatedly with one exception: a subset of the optimisations are also applied directly after the *eval* and *apply* inlining (see figure 4.1). The reason for running some optimisations at this point is to immediately remove some of the "unnecessary" code that was introduced by the *eval* and *apply* inlining. This subset of optimisations does not include the most aggressive ones: late inlining, case hoisting and arity raising.

The choice to run a subset of the optimisations a bit earlier is not really a critical requirement of the GRIN transformation process, it is done only as a convenience step. In particular, some of the optimisations in the "simplifications part II' phase become a bit easier to implement if the code has been cleaned up after the *eval* and *apply* inlining.

Miscellaneous

Apart from the simplifying and optimising transformations there are a number of other "things" going on as part of the GRIN transformation process, things that are not very advanced, but that we still feel need to be described here to complete the picture. It can be either transformations that do not fit into either of the simplifying or optimising categories as defined above, or it can be passes that just "gather information", without changing any code (very simple kind of program analysis). We give a short summary here and explain more details in their respective descriptions (note that these are not included in the overview in figure 4.1):

- *bind normalisation*: use the monadic *associativity law* to change the structure of GRIN expressions (see section 4.4.1 on page 156),

- *argument reordering*: change the order of tag arguments to better suit the underlying implementation (see section 4.4.2 on page 157),

- *tag number assignment*: assign unique tag numbers to all tags in the program (see section 4.4.3 on page 159),

- *tag information*: collect information about the use of tags in the program, this is used by many transformations (see section 4.4.4 on page 160),

- *final GRIN*: explain the structure of the GRIN code after all transformations, i.e., the code that is input to the RISC code generator (see section 4.4.5 on page 161).

Of these, there are two in particular that are used so extensively during many of the simplifying and optimising program transformations, that a reader might want to take a quick look at them before starting to read about the transformations. The two are the *bind normalisation* transformation (see section 4.4.1 on page 156 and the collection of *tag information* (see section 4.4.4 on page 160).

4.2 Simplifying transformations

As mentioned above, the point of the simplifying transformations is to translate a GRIN program from "high level" GRIN to "low level" GRIN. We will not formally define what we mean by "high level" and "low level" GRIN, but instead give some examples. Perhaps the most apparent example is the GRIN fetch operation. Initially, the GRIN code generator will use "ordinary" fetch operations, that load complete node values from the heap. However, after the *split fetch* program transformation all the "different parts" of a node value (the tag, the first argument, etc.) will be loaded separately, using fetch operations with offsets (see section 4.2.7 on page 102).

Another example is case expressions. Initially they will "match" complete node values, and case patterns may contain variables (which become bound inside the corresponding alternatives). After the *case simplification* transformation, all case expressions will examine a single tag variable, and all case patterns will be single known tags (or basic values), not node patterns, and the patterns will not contain any variables.

Some more properties of the final low level GRIN code, after all transformations, are discussed in section 4.4.5 on page 161.

4.2.1 Inlining calls to *eval*

By far the single most important of all GRIN program transformations is the elimination of the special *eval* procedure from the GRIN program. Recall how an *eval* procedure is automatically constructed for each program (see section 2.5.2 on page 44). An example is shown in figure 2.8 on page 45. The point of the *eval* procedure is to force the evaluation of a suspended function application and eventually return a value representing a weak head normal form. If the *eval* argument is already a whnf then *eval* will return it immediately. To enforce the lazy evaluation, *eval* will also do an update operation before it returns, to overwrite the original redex with the whnf value.

When doing the *eval inlining* transformation we will use a slightly different *eval* procedure than the one in figure 2.8. This alternate version (shown in figure 4.2 on the next page) is semantically equivalent, but might seem a bit more inefficient since it will always perform the update operation, even if the value was already on weak head normal form. Despite this, the new version is

```
eval p = fetch p ; λ v →
         (case v of
            (CInt x')      → unit v
            (CNil)         → unit v
            (CCons x xs)   → unit v
            (Fupto a b)    → upto a b
            (Fsum c)       → sum c
         ) ; λ w →
         update p w ; λ () →
         unit w
```

Figure 4.2: An example *eval* procedure.

more suitable for use in the inlining transformation, mainly due to its simplicity. The potential inefficiencies can be removed later by other transformations.

Since *eval* is a normal GRIN function, calls to *eval* can be *inlined* (*unfolded*) just like calls to any other function, i.e., we can replace applications of *eval* with a copy of the *eval* body substituting actual for formal parameters. The *eval* procedure can never be recursive, which means that if we inline all calls to *eval*, the *eval* procedure itself is no longer used and can be removed.

Of course, for any real sized program the GRIN *eval* procedure, and especially the case expression inside it, will be very big. This case expression must enumerate all node values (closures and whnfs) that are ever written to the heap in the program. Or really, it has to enumerate node patterns with all possible tags that can ever occur in the program. The tag "arguments" does not have to be determined though, they are always variables (indeed, this is even enforced by the GRIN syntax, see figure 2.1 on page 28). Given such a big case expression, and the fact that there are normally very many calls to *eval*, it seems really wasteful to inline all those calls. The only thing we would achieve by that is probably code explosion. Moreover, for each particular *eval* point, only a small subset of all possible closures are "possible" values. For most programs, "all values ever written to the heap" is a huge overestimate to the actual values that can arise at each *eval* point. This means that if we do a straightforward inlining of *eval* we would get huge case expressions in which we know that most of the alternatives probably are "impossible" and can never be reached. Consequently, traditional inlining of *eval* does not seem like a very good idea.

Using the heap points-to analysis

Fortunately, there is a solution to this problem, a solution which we have already discussed at length in the previous chapter, the *heap points-to analysis* (see

chapter 3).

The analysis result is exactly what we need to perform the *eval* inlining in a more clever way. The analysis gives a safe approximation to what tags (representing possible closures and weak head normal forms) that can arise for each *eval* point. Using this when inlining *eval*, we can at the same time delete all the impossible alternatives from the potentially huge *eval* case expressions.

Clone + specialise + inline

The transformation might be easier to grasp by visualising it in several steps (compare with the example *eval* procedure in figure 4.2 on the page before):

Step 1: For each call to *eval*, create a *new instance* (an identical *clone*, but with a new name) of the *eval* procedure and replace the call to the general *eval* with a call to this new procedure.

Step 2: For each newly created version of the *eval* procedure, use the result of the heap points-to analysis to *specialise* it, i.e., delete all "impossible" alternatives from its big case expression.

Step 3: *Inline* all calls to the specialised *eval* procedures.

One might argue that the last step above could be optional, but the inlining is actually an important part of the transformation. By doing so we in some sense "expose" the evaluation code to the context in which the original call to *eval* occurred, something which will enable many other optimisations.

For efficiency reasons, all the steps above are currently implemented as a single transformation. In fact, the general (and huge) *eval* procedure will never be constructed at all. Instead, the implementation will create specialised instances of it on the fly using the result of the analysis and then immediately inline the specialised procedures. Since *eval* is not recursive, the general *eval* procedure will no longer be used in the program after the *eval* inlining, and can be removed.

Note also that each time we create a new *eval* instance we must give new names to all local variable in the copy of the *eval* body, to avoid *name capture* problems.

An example

To show what happens in practice we will return to the example program in figure 2.8 on page 45. The *sum* procedure, also shown here in figure 4.3 on the next page, contains two calls to *eval*. First, the argument list is evaluated to see if it is non-empty. If it is, the first element of the list will be forced to get an integer.

Let us initially concentrate on the first call, "*eval l*". To see what the heap points-to analysis says about "*l*" we need to return to figure 3.5 on page 69

```
sum l = eval l ; λu →
        case u of
          (CNil)        → unit (CInt 0)
          (CCons t ts)  → eval t ; λ(CInt t') →
                          sum ts ; λ(CInt r') →
                          intAdd t' r' ; λs' →
                          unit (CInt s')
```

Figure 4.3: The *sum* procedure from the example GRIN program.

where the analysis *abstract environment* is shown for the same example program. According to that "l" can be bound to the abstract locations 3 and 6. To see what possible node values those two locations can contain we need to look into the analysis *abstract store* (abstract heap). Assume for a moment that we are using the variant of the heap points-to analysis that do not include sharing information, i.e., we have the abstract store in figure 3.4 on page 68. In this case, locations 3 and 6 can both contain either Fupto, CNil or CCons nodes. In other words, the "*eval l*" call can find either an Fupto, a CNil or a CCons node. By performing the same exercise for the second call to *eval* in the *sum* procedure, "*eval x*", we can see that there is only one possible tag for "x", it will always be a CInt node. I.e., we are guaranteed that the node already represents a whnf, so the call to *eval* was really "unnecessary" and can be deleted. We do not make such considerations at this point, though. Later transformations will take care of any "unnecessary" *eval* overhead that may have been introduced (and completely remove the overhead in this case).

Using the above information, we can specialise and inline the two calls to *eval*, and get a new version of the *sum* function (shown in figure 4.4 on the following page). Note that this version of *sum* is only equivalent to the previous for the particular program that the analysis was run on! If the *sum* function was part of a different program, the analysis result (and consequently the resulting code) could be completely different!

If we instead had done the *eval* inlining using the more detailed version of the heap points-to analysis, with sharing information enabled (which is the default), the result would have been even simpler. The result of the improved analysis can be seen from the resulting abstract store in figure 3.6 on page 70. For the call "*eval l*" there is now only one possible tag, Fupto. If this information is instead used when inlining *eval* in the *sum* function, the result would be almost the same as in figure 4.4 except that in the first case expression only the Fupto alternative would be present. We would also have the information that the Fupto closure is guaranteed to be *unique* (non-shared) which will turn out useful later, when optimising update operations (see section 4.3.9 on page 133).

```
sum l = (fetch l ; λ v →
          (case v of
              (CNil)          → unit v
              (CCons x xs)    → unit v
              (Fupto a b)     → upto a b
          ) ; λ w →
          update l w ; λ () →
          unit w
         ) ; λ u →
         case u of
           (CNil)          → unit (CInt 0)
           (CCons t ts)    → (fetch t ; λ z →
                               (case z of
                                   (CInt x') → unit z
                               ) ; λ k →
                               update t k ; λ () →
                               unit k
                              ) ; λ (CInt t') →
                              sum ts ; λ (CInt r') →
                              intAdd t' r' ; λ s' →
                              unit (CInt s')
```

Figure 4.4: The *sum* procedure after *eval* inlining.

Further simplification

The resulting *sum* procedure may seem unnecessarily inefficient in several respects. E.g., it contains a case expression with only one possible alternative, a CInt tag. In the case of the sharing analysis the program also contains a case expression with only an Fupto tag. We know that these must match, so the case tests are clearly unnecessary. Another example of an inefficiency is the update operation from the first inlined call to *eval*. From the analysis abstract heap (figure 3.6 on page 70) we can see that the "l" reference to the Fupto closure is guaranteed to be unique, so updating the closure is unnecessary. No one will ever read that heap node again. The update of the variable "t" is also unnecessary, not because it is unique but because it is already a weak head normal form.

Also, if we imagine that the analysis without sharing information had really been used, i.e., if the first case expression in figure 4.4 really included three alternatives, then we would sometimes "examine" the same value twice. If the variable "v" was already representing a whnf then the same value would be

examined again by the second case expression (the variable "*u*").

However, rather than doing many *ad-hoc* simplifications at this point, we will leave the program as is and let other program transformations "clean up" the code. Many transformations will make a small simplification or optimisation to the code, and taken together the result will hopefully be very efficient code.

An experiment examining the average *number of tags* in unfolded calls to *eval* can be seen in figure 9.6 on page 285.

4.2.2 Inlining calls to *apply*

The *apply* procedure, like the *eval* procedure, is automatically constructed for each program (see section 2.5.6 on page 54) and all calls to *apply* will be eliminated in the same way as calls to *eval* was. I.e., we will use the result of the heap points-to analysis to create specialised versions of the general *apply* procedure, for each call to *apply*, and then inline the specialised versions. Since the *apply* procedure we create is never recursive, inlining all calls to *apply* will result in a program where the *apply* procedure itself is no longer used and can be removed.

The heap points-to analysis returns information for each call to *apply*, giving a set of possible node values at that particular point, in exactly the same way as for *eval*. The difference from *eval* is that these node values represent partial applications (function values) rather than fully saturated, but suspended, applications. The sharing information that we get for calls to *eval* is not applicable to *apply* since the first argument of *apply* is a node value and not a pointer into the heap.

To show a concrete example of *apply* inlining we will return to the example higher order functions program discussed earlier (see figure 2.11 on page 55). An *apply* procedure for this program is shown in figure 2.12 on page 55 and part of the GRIN code is shown in figure 2.13 on page 56.

Here, in figure 4.5 on the next page, we show the *twice* function before and after *apply* inlining. In the same transformation step we have also included *eval* inlining, to emphasise the close relationship between *eval* and *apply* in this case.

From the inlined calls to *eval* and *apply* we can see what result the heap points-to analysis gave for this particular program. The *eval* point includes all possible P-tags in the program. All P-tags represent whnfs, so they are just returned. The *eval* also includes a "normal" closure, an Fap node. Like the *eval*, the *apply* case expression also includes all possible P-tags in the program, which might seem as a "weak result" from the analysis, normally we would expect only a subset of all possible tags. However, at the point of the call to *apply all* those tags can actually appear when the program is executed, so the analysis is not to blame. This situation is not very common, though, the twice program is quite extreme in the way it uses higher order functions. For most programs, a call to *apply* will only have a small subset of all P-nodes as possible values, in the same way as for calls to *eval*.

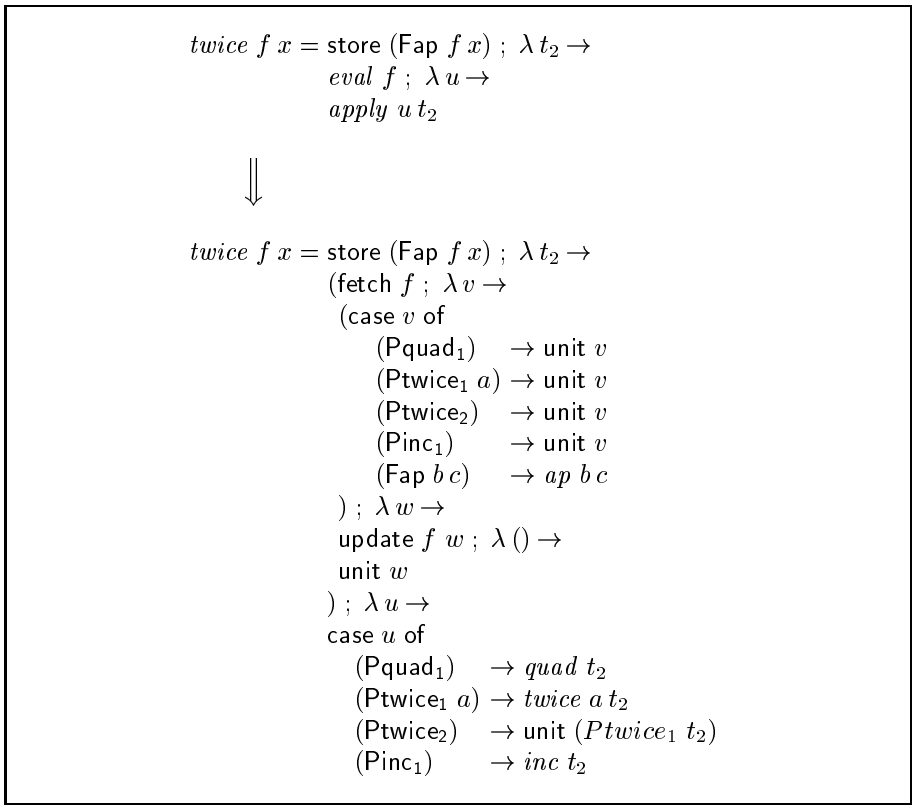

Figure 4.5: The result of *apply* (and *eval*) inlining.

However, even in "normal" programs we might run into cases where the possible set of node values for an *apply* (or *eval*) point is very large. E.g., in "library functions", with very many callers. In such situations, a more general approach to *cloning* may be very favourable.

4.2.3 Update specialisation

In the sequence of GRIN *simplifying* transformations, after the *eval* and *apply* inlining, the next transformation is the *update specialisation*. The GRIN update operation:

update $p\ v\ ;\ \lambda\,() \rightarrow \ldots$

overwrites the heap node pointed to by its first argument with the *node value* in its second argument. Unfortunately, to be able to later implement the GRIN

update operation efficiently, i.e., when translating to real machine code (RISC), we need to discuss some details of its behaviour already at this point. To summarise, the main restrictions that we have decided to obey are:

- we are not willing to accept any *unknown calls* (i.e., calls, or jumps, where the destination is unknown at compile time),
- we want "all code" to be explicit in the GRIN program, we do not want some parts of it as hidden "magic" in the runtime system,
- the current GRIN implementation, i.e., the mapping of GRIN to real machine code, uses a *node layout* where the memory layout of a node value depends on the node tag (see section 5.2.3 on page 178).

Many implementations of graph reduction use some kind of *dynamic dispatch* technique, built-in to the runtime system, to implement updates. Unfortunately, that is not really an option in GRIN , because such a dispatch normally imposes an unknown jump. This immediately disqualifies the dispatch technique according to the first item on the list above. Also, contrast this to the *eval* and *apply* inlining transformations, where we go through quite a lot of trouble just to eliminate unknown calls! It would not be a tempting idea to introduce them again just to handle updates.

The STG machine uses a technique based on an *update stack*, but that is not really a viable alternative either, because it "hides" code from the GRIN program. It is hard to see how an explicit update stack could be added to GRIN without interfering too much with the rest of the language. It would not really make sense to add such a thing to GRIN.

Yet another alternative could be to create a "magic" update procedure that is built-in to the runtime system. This procedure could inspect the node at runtime and do the right thing depending on the tag. However, this would also mean that we "hide" the problem inside the runtime system, and would go against the overall GRIN philosophy of making as much as possible visible in the GRIN code itself.

A better alternative would then be to use a technique similar to what we used for *eval* and *apply*, and make the tag inspection explicit in GRIN. We would then insert calls to an *update* procedure, rather than to use the GRIN update operation directly. The general *update* procedure would inspect its second argument using a GRIN case expression containing alternatives for all whnfs in the program. Using the heap points-to analysis we could then specialise, and inline all calls to the general *update* procedure. However, doing this would result in an "extra" case expression whenever we need to do an update, an extra cost which does not seem attractive at all. And as we will soon see, we can use a different technique that will avoid such an extra cost in almost all situations.

Updates need a known tag

When taking all the above into account, the problem basically boils down to the following observation: the GRIN builtin update operation can not be made to implement a "general" update, it can only implement an update for a specific tag.

In some sense, part of the cause of these problems is the GRIN design decision to put the update operation inside the *eval* procedure rather than at the end of each function (at return points). Together with the *eval* inlining, this will have the effect of putting the updates in the *caller* rather than in the *callee* (see also section 2.5.4 on page 47). If we instead had put the updates in the callee, the tag of the returned value would often be "known", so implementing the update would be less of a problem. Compare this to our method which results in updates in the caller. In the caller it is not obvious what the returned value is (i.e., the value *after* the *eval*). For an example look at the update in the inlined *eval* in figure 4.5 on page 90.

However, the GRIN update placement is not the only reason for our problems, even with updates in the callee we would get difficulties when the returned value was not known for some reason. And the choice to put updates in *eval* can not really be changed for other reasons (see section 2.5.4 on page 47).

The solution

Fortunately, there is an easy solution to the above problem, a simple GRIN program transformation. Since all update operations in GRIN stem from inlined calls to *eval*, an update can only occur in a few specific situations, when something needs to be evaluated. Normally the GRIN code generator will not insert a call to *eval* unless a whnf is needed. This means that all update operations (with one exception, see "Returning calls to *eval*" below) will occur in such a way that it is always immediately followed by either a scrutinising case expression or a pattern matching lambda binding (with a known tag in the pattern). An example of both these situations can be seen in figure 4.4 on page 88 (the first update is followed by a case expression and the second by a CInt pattern).

Scrutinising case expressions. When an update is followed by a scrutinising case expression we will perform the transformation shown in figure 4.6 on the next page. The update is moved into each branch of the case expression. Each of the moved updates is also *specialised*, i.e., it is turned into an update that "knows" the actual tag of the value it is writing to the heap. A specialised update is shown with the tag as an annotation.

For the transformation to be valid there is one side condition, the variable "p" must not occur inside "$\langle m_1 \rangle$". The reason for this is that in the transformed code, "p" still points to a node that has not yet been updated (when "$\langle m_1 \rangle$" is executed). If the node were read at that point we would get a different value

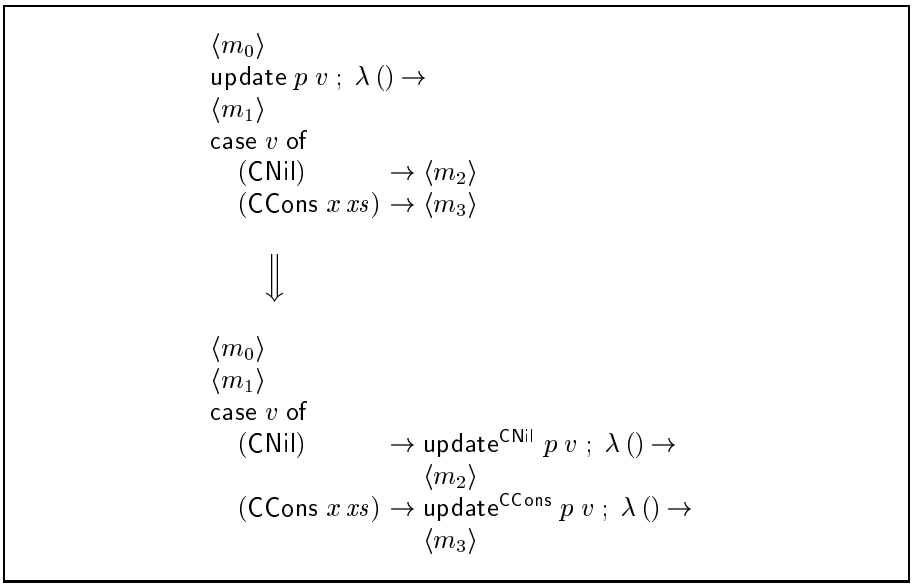

Figure 4.6: Update specialisation (case expression).

compared to in the original GRIN code. This is not a problem in practice though. In code generated by the current GRIN code generator, "$\langle m_1 \rangle$" will in fact always be empty by the time that the update specialisation is applied. The reason is that other GRIN transformations will always eliminate code that initially may appear between the update and the scrutinising case expression. As an example, to make the update in figure 4.5 on page 90 appear immediately before the case expression we need to use two transformations: *bind normalisation* (section 4.4.1 on page 156) and *copy propagation* (section 4.3.2 on page 113).

Another condition is of course that the variable "v" must not be redefined inside "$\langle m_1 \rangle$", but this is enforced by the GRIN static single assignment property (see section 2.3.7 on page 34) without any special treatment.

Replicating the update operation can increase the total code size, slightly, but that is probably not a problem in practice.

GRIN pattern matching lambda bindings. Sometimes the value that is returned by a call to *eval* is completely known (i.e., the tag is known). In this case there is no need to use a case expression to examine it. A typical example is when the value of an integer is needed, e.g., to add it to another integer, so a call to *eval* is inserted. In this case there will be only one possible tag (after the *eval*), the CInt tag. An example of this is the second call to *eval* in the *sum* function, originally shown in figure 4.3 on page 87 and after the *eval* inlining in

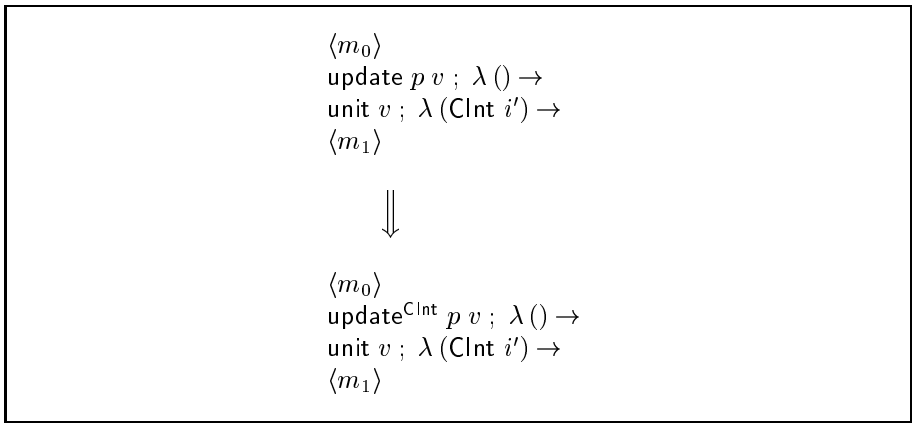

Figure 4.7: Update specialisation (pattern binding).

figure 4.4 on page 88.

In the integer example the type of the value is a *product datatype*, so there can never be more than one possible tag. However, in GRIN we can also have situations where a value of a *sum datatype* is completely determined because the heap points-to analysis has predicted that only one tag can ever appear at that point. Whenever a single known tag is detected it will result in GRIN code with a pattern matching lambda binding instead of a scrutinising case expression. The known tag will be put in the node pattern of the binding, to show that there is only one possible tag at that point.

Returning to the update specialisation, if an update followed by a pattern matching lambda binding is found all the transformation has to do is annotate the update with the known tag, no code needs to be moved or changed. An example of this is shown in figure 4.7.

The current GRIN code will match exactly the code in figure 4.7, due to the way we inline *eval*. However, the transformation is in fact done in a more general way. We use GRIN *tag information* (see section 4.4.4 on page 160). The point of tag information is to say, for all node and tag variables in the program, what actual tags they might hold during an execution. The tag information is really a merge of the heap points-to analysis result and other information collected from the program at various stages during the compilation. If an update is found by the transformation, and the tag information gives a singleton tag for the node variable, the update will immediately be annotated with that tag. If the tag information gives several possible tags, then there *must* exist a scrutinising case expression in the following code, and the above method will be used instead.

Returning calls to *eval*

There is a small complication with the above methods, they will not work in one special situation: when there is no code "after" the update that we can deduce a tag from. This happens when the original source code consists of a function that returns a previously unevaluated variable without the value being "inspected" or used in some way. Since a function must always return a value representing a whnf, the GRIN code generator will make sure that the value is indeed a whnf by inserting a call to *eval*. We call this situation a "returning *eval*" (a tailcall to *eval*).

The simplest possible example of a returning *eval* is the identity function, defined by: "$id\ x = x$". The GRIN code for this function consists solely of a call to *eval*: "$id\ x = eval\ x$". Once the call to *eval* is inlined, the code for the function will end with an update followed by a unit operation that returns the value. There will be no way in the code to "see" what actual tag is returned. To handle this given the design decisions regarding update operations discussed above there is really no other way than to insert a scrutinising case expression after the update, containing all tag values that are possible at that point. The set of possible tags, *after* the *eval*, is fortunately part of the result of the heap points-to analysis. After we have inserted a case expression, we will get an update followed by a scrutinising case expression, and we can apply the normal update specialisation described above.

An example of all this is shown in figure 4.8 on the following page where we have assumed that the possible tag values for the variable "u" are CNil and CCons. In the figure we include the *eval* inlining to show the origin of the update operation.

Having to insert an extra "unnecessary" case expression occasionally must be considered a weakness of the current GRIN implementation. Fortunately, in many situations the effect of the extra case expression can be eliminated using a general *inlining* transformation (see section 4.3.10 on page 135) followed by the *case hoisting* transformation (see section 4.3.11 on page 137).

As an example, assume that a call to the function in figure 4.8 was followed by a case expression examining the result of the call (which is the normal situation). In this case we could *inline* the function in the figure, and as a result get two case expressions scrutinising the same value immediately after each other (the update case expression and the one originally in the caller). We can then "combine" the two case expressions using *case hoisting*, and avoid the extra overhead.

A more advanced transformation would be to try to move only the update operation to the caller, without inlining the complete function. We would then have to make sure that the pointer to the original closure (the first argument to update) is available in the caller, something which is not always the case. Such a transformation have not yet been explored.

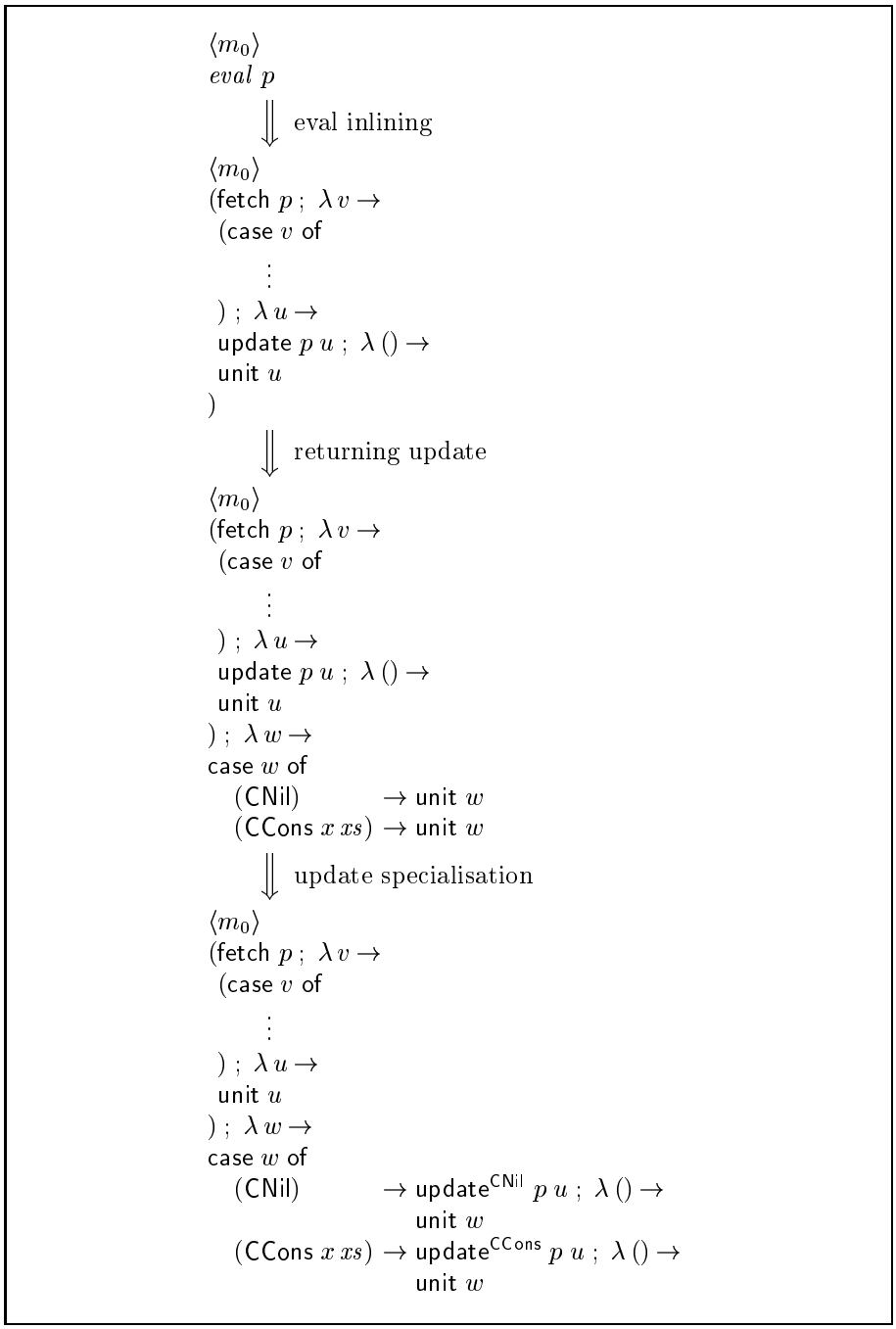

Figure 4.8: Update specialisation for "returning *eval*".

Other update transformations

Apart from specialising `update` operations, which we call a *simplifying* transformation (i.e., it is "necessary"), there are also two *optimising* (and hence optional) transformations involving updates. A transformation to remove unnecessary updates is described in section 4.3.8 on page 132 and a transformation that tries to avoid updates on whnfs (only closures need to be updated) is described in section 4.3.9 on page 133.

4.2.4 Vectorisation

The GRIN *vectorisation* transformation is an important step in the simplification from "high level" to "low level" GRIN. The main idea of vectorisation is to make "register usage" visible in GRIN. The term "register usage" in this context should not be interpreted too literally. It is not meant to exactly describe the register usage of the real machine code, it should be seen more as an approximation of the register needs of the code.

Consider the different kinds of GRIN values and variables (see section 2.3.5 on page 33). For variables holding pointers or basic values (like integers) we know that such a value will always fit in a single machine register (or maybe a few registers depending on things like number precision, etc.). However, the same can not be said about variables containing entire node values, in principle they can occupy an arbitrary number of registers, depending on the sizes of the nodes they can hold. Recall that the *size* of a node value is defined as the arity of the tag plus one (see section 2.3.3 on page 30).

The idea of vectorisation is to eliminate all node variables, replacing them with *explicit nodes*, i.e., node values or node patterns containing a tag variable (or a known tag) followed by a number of argument variables. When we do this we make the register usage of the code, if not exact, at least more concrete.

Using node sizes

Using *tag information*, explained in section 4.4.4 on page 160, we can say for each node variable in the program exactly what tags that node variable might hold. We know that the *maximum size* of a node variable is the maximum size of the node values it can hold. Since the node size is just the *tag arity* plus one (the tag itself) and the tag arity is always known (see section 2.3.3 on page 30), we can calculate the maximum variable size as the maximum tag arity plus one. Once we know the maximum size of a node variable the actual vectorisation transformation is quite simple, we can replace all occurrences of the variable in the program with an explicit node value.

An example of vectorisation is shown in figure 4.9 on the next page. Here we have assumed that the maximum possible size of the node variable "v" is three, i.e., a tag followed by at most two arguments. During the transformation,

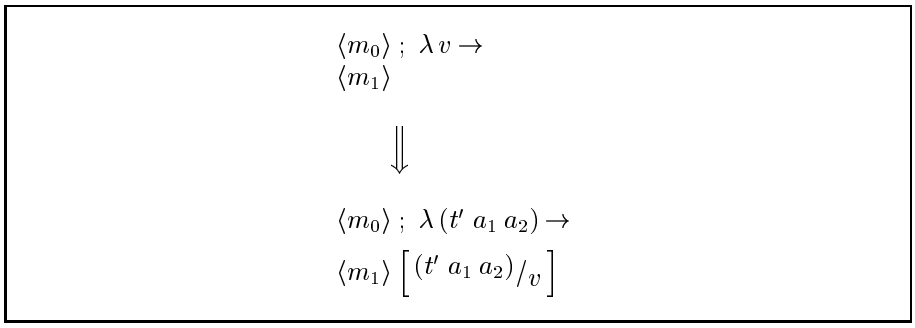

Figure 4.9: Vectorisation.

when the variable "v" is first found, in the lambda binding, it is replaced by an explicit node. The variable names "t'", "a_1" and "a_2" are all fresh names. A substitution is applied to the rest of the code as shown in the figure, replacing all other occurrences of the node variable. Note that these "occurrences" must all be *uses*, a GRIN variable is never *defined* in more than one place (the static single assignment property, see section 2.3.7 on page 34). The substitution will actually need a small modification to work correctly in all cases (see below).

Note that we consider a tag value itself to be a basic value. This is not very strange, e.g., a single machine register will always be enough to hold a tag value. The fact that tags are considered basic values can be seen in the figure where we have used the basic value convention for the tag variable name (see section 2.3.5 on page 33).

Note also that making the node explicit does not necessarily say anything about the way a node is represented in registers or stored in memory. In the current implementation there will indeed be a close correspondence between the node size and the number of registers used to hold it (or the number of memory words used to store it), but it is easy to imagine a different implementation, maybe concentrating more on "packing" the different parts of a node. The only thing regarding the representation that is required from an implementation is that there should be some way to "extract" the tag, the first argument, etc.

The tag information that is used to determine the maximum size of each node variable is of course very important for the vectorisation to work. This information is determined from the result of the heap points-to analysis combined with other "information" found in the code. This is basically done by initialising the tag information using the analysis result, and then refining the information during the GRIN transformation process. In many cases the heap point-to analysis will give exactly the same result as a simple inspection of the code would reveal. An example of this is given in figure 4.10 on the facing page. Here, a function *foo* returns a node value that is then immediately examined

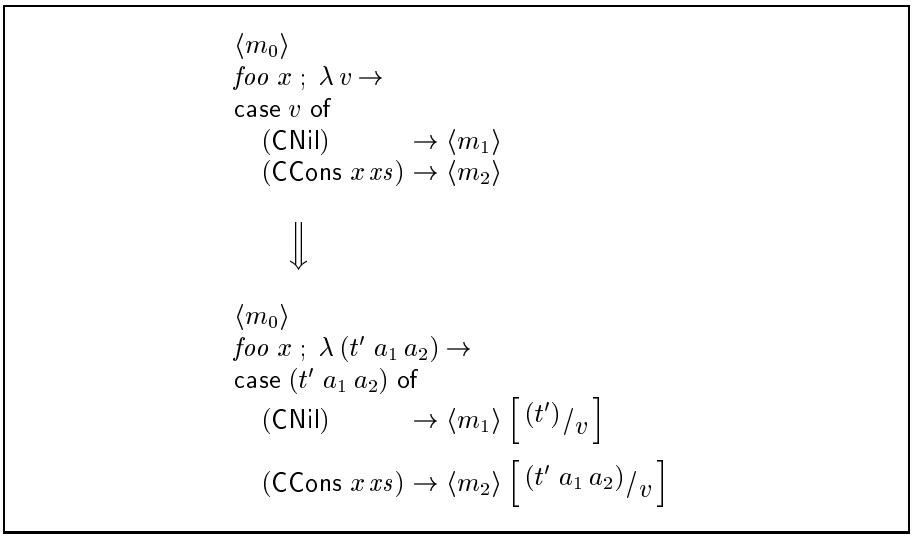

Figure 4.10: Another example of vectorisation.

by a case expression. By looking at the possible tags in the case expression the maximum node size for the variable "v" is trivially determined to be three.

Binding a value of an unknown size (the result of the call to *foo*) to a node with an explicit size might seem a bit strange. One might wonder what happens with the two node arguments if the tag is a CNil? The result is actually not very strange at all, and is all explained in the GRIN semantics (see section 2.4.4 on page 38). In the semantics, in the case of a CNil tag the value of the variables "a_1" and "a_2" will simply be bound to a special *undefined* value. We guarantee that an undefined variable can never be used in the respective alternatives of the case expression by modifying the substitutions as indicated in figure 4.10. The substitution for a particular case branch will never include more argument variables than the arity of the corresponding tag, i.e., the *size* of a case pattern must be the same as the size of the node substituted. E.g., in the CNil alternative an occurrence of "v" will be replaced by the explicit node "(t')", containing only the tag. This guarantees that the argument variables never occur in that branch, so an execution can never "run into" an undefined value (at least not because of the vectorisation transformation).

Of course, in the actual implementation of GRIN, the "undefined value" does not appear at all. For the CNil case in the figure, the variables "a_1" and "a_2" will simply be unbound.

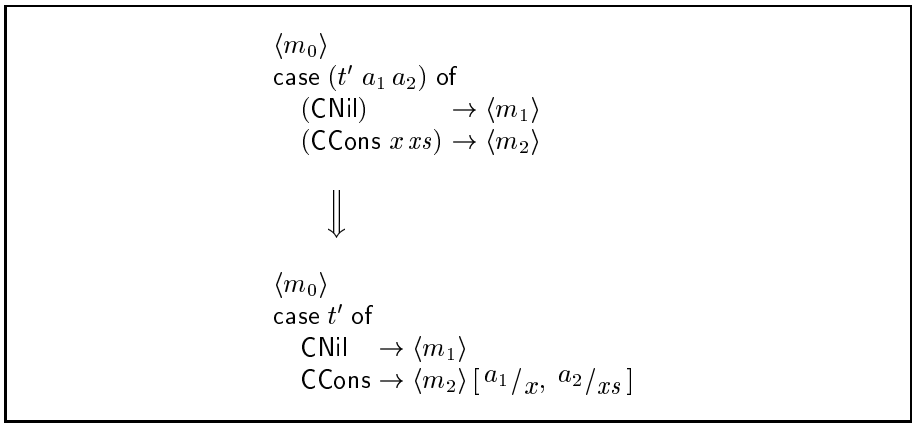

Figure 4.11: Case simplification.

4.2.5 Case simplification

The previous vectorisation transformation makes yet another simplifying transformation possible, to make case expressions a bit simpler. Originally, the "scrutinised value" in case expressions are normally node variables (which the vectorisation changes to explicit node values). Consequently, most case patterns are *node patterns*, patterns that match entire node values. These patterns can also bind variables, if the tag has arguments (the GRIN syntax enforces that tag arguments in patterns are always variable, see figure 2.1 on page 28).

However, to "execute" a GRIN case expression we only need to look at the tag value, not the entire node. Remember that case patterns in GRIN are always simple and non-overlapping, each pattern with a known tag (see section 2.3.6 on page 33). This means that it is always enough to find the alternative with the matching tag, we can never run into a match that later "fails" as in some implementations of non-simple patterns [Aug85].

Since all node variables have been turned into explicit nodes by the vectorisation, it is very easy to find case expression where the scrutinised value is an entire node. The *case simplification* transformation will change such a case expression into an equivalent that only inspects the tag rather than the entire node. Since node patterns can bind variables we must also apply a substitution to the right hand side of each case alternative, changing the bound variables to the corresponding variables taken from the scrutinised node. The transformation is very simple and is illustrated by an example in figure 4.11. Similarly to in the vectorisation transformation the substitution is adjusted so that only "existing" arguments are substituted for.

Note that the body of the CNil alternative, "$\langle m_1 \rangle$", is guaranteed to not contain the variables "a_1" and "a_2" due to the way the vectorisation transformation

is done (see section 4.2.4 on page 97).

After the case simplification, all case expressions will be scrutinising only basic values (including tag values), and all case patterns will correspondingly be just basic values. The patterns will not contain (and bind) any variables.

4.2.6 Compilation of conditionals

The following simplifying transformation is a kind of "preparation" for the RISC code generation. It is only applicable (or meaningful) to some computer architectures, that lack instructions to turn *condition codes* into boolean values in general purpose registers. This is the case with the SPARC architecture [SPA92] which so-far has been the primary target for the GRIN back-end. The problem can be visualised by the following simple GRIN code example:

$$intEQ\ x'\ y'\ ;\ \lambda b' \rightarrow \ldots$$

Here, *intEQ* is the GRIN primitive operation used to compare two basic values for equality. It will produce a boolean value as result, in the variable "b'". A basic value variable in GRIN should really be seen as the same thing as a "general purpose register", but unfortunately there is no easy way on the SPARC processor to transfer the result of a conditional instruction into a general purpose register. Instead, all "boolean results" are put in a special *condition code register*. To really transform a condition code into a boolean value in a general purpose register we would have to insert a conditional branch instruction and code to explicitly set the register to either *true* or *false*, depending on if the branch were taken or not. The above work-around results in very inefficient code and we should try to avoid it as much as possible, i.e., try to avoid "explicitly creating" boolean values. Instead, the RISC code generator must try to match "compound" GRIN expressions, that create and then immediately use the boolean value:

$intEQ\ x'\ y'\ ;\ \lambda b' \rightarrow$
if b' then
  $\langle m_1 \rangle$
else
  $\langle m_2 \rangle$

When the RISC code generator finds the above expression it can generate efficient code, which for the SPARC basically means a comparison instruction followed by a conditional branch. The "boolean value" (in "b'") is never really constructed, it will only exist for a short while in the condition code register. Of course, for this to work "b'" must not be used in either "$\langle m_1 \rangle$" or "$\langle m_2 \rangle$".

This is all fine. The above is a well-known code generation technique, and can be seen as a very simple form of *boolean short-circuiting* [ASU86, section 8.4]. Unfortunately, this short-circuiting is not always possible. In some situations

we have to create the boolean value. E.g., if we have GRIN code with a boolean primitive but no if expression:

$$intEQ\ x'\ y'\ ;\ \lambda b' \rightarrow$$
$$\langle m_1 \rangle$$

If "$\langle m_1 \rangle$" does not contain an if expression examining the variable "b'", we must really create a boolean value, which in GRIN means either of the two boolean tags, CFalse or CTrue. For the SPARC this must be done using the technique mentioned above. Besides being inefficient, this has two disadvantages. First, the code generator will have to include two special cases for boolean primitive operations, differing only in if the primitive is followed by an if expression or not. Both these cases will result in basically the same generated code. Second, it might be seen as breaking the "informal invariant" that the GRIN boolean primitives should be very "simple" to implement, typically only one or two machine instructions.

Fortunately, even if we can not avoid the inefficient code, we can avoid the latter two problems with a really simple GRIN program transformation. The transformation makes the "extra conditional branch" explicit in the GRIN code for situations where it is needed, by inserting an extra if expression. This fixes both the extra special case in the code generator and the broken invariant. As we have said earlier, one of the intentions with the GRIN simplifying transformations is exactly to make it simpler for the code generator. An example of the transformation is shown in figure 4.12 on the facing page.

Note that for other architectures, like the MIPS [HP90], that have sufficient instructions for creating boolean values, this transformation should not be used. In such cases the transformation will only introduce unnecessary overhead.

4.2.7 Split fetch operations

The next step in simplifying GRIN is to deal with fetch operations. So far, all the fetch operations we have seen have been used to load entire node values from the heap. However, to load an entire node is a somewhat "complex" operation that we would like to simplify, as part of the translation process from "high level" to "low level" GRIN.

What the *split fetch* transformation will do is to change each fetch operation into a sequence of simpler fetch operations using *offsets* (see section 2.4.5 on page 41). A fetch operation with offset i will load only the i:th component of a node value, where the tag is defined to be component zero, the first argument component one, etc. Each offset fetch operation will bind its result to a single variable rather than an entire node. Note that the vectorisation transformation (see section 4.2.4 on page 97) guarantees that all node variables have been replaced with explicit nodes, so we will always be able to "split" a lambda binding that follows a fetch. The transformation is illustrated in figure 4.13 on the next page.

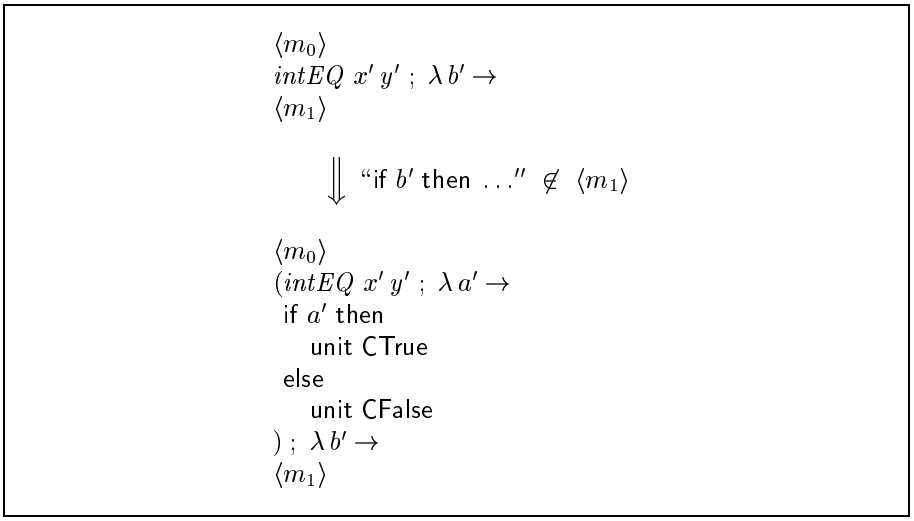

Figure 4.12: A boolean primitive without an if expression.

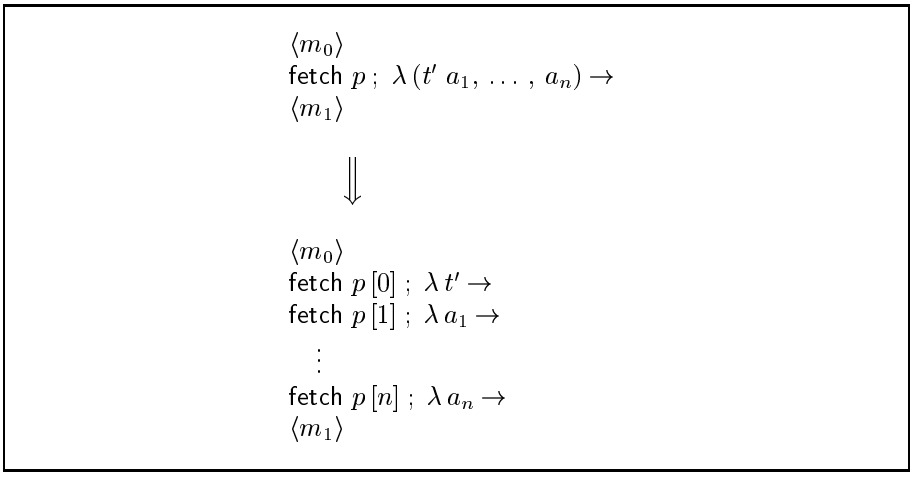

Figure 4.13: The split fetch transformation.

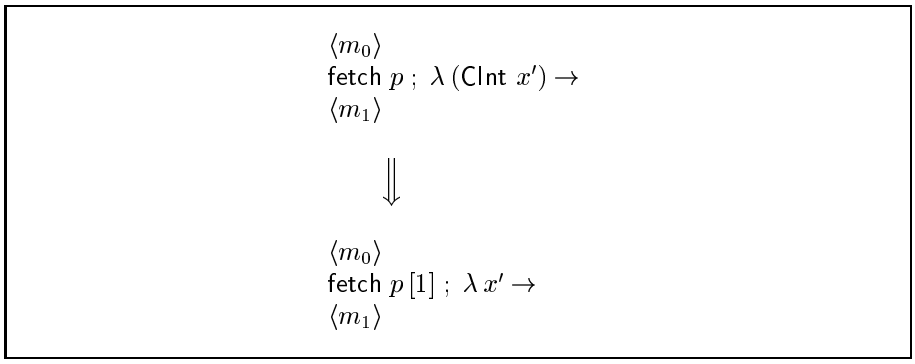

Figure 4.14: Another example of the split fetch transformation.

Note again that this implies very little about the way node values are actually stored in memory. Even though the fetch operations are split, an actual GRIN implementation could, at least in principle, treat them as dependent of each other in some way. E.g., the load of the tag and the first argument might be combined into a single load machine instruction. This all depends on the actual memory layout decisions made by an implementation. In the current GRIN implementation, though, each offset fetch will in fact be turned into a single load instruction (see section 5.2.3 on page 178).

Note that the split fetch transformation does not change the semantics of the program. The interesting question is what happens if the tag variable ("t'" in figure 4.13) holds a tag with less than "n" arguments? But this is really no different than the situation before the transformation, when the result is bound to an explicit node. Some of the argument variables ("a_1" to "a_n") can sometimes be bound to undefined values, as discussed in the vectorisation transformation (see section 4.2.4 on page 99). This does not change at all when we "split" fetch" operations. Although an "undefined fetch" is ok semantically, the intention is of course that an implementation should never actually load a value that will not be used. This will be taken care of by the next transformation (*right hoist fetch operations*, see section 4.2.8 on page 107).

Sometimes the "tag position" (offset zero) of a node pattern that binds the result of a fetch operation is a known tag, rather than a tag variable. In this case we will skip the fetch of the known tag and just load the arguments, as shown in figure 4.14.

In fact, we will be a bit more clever also in the case where the bound node contains a tag-variable (like "t'" in figure 4.13), and use *tag information* (see section 4.4.4 on page 160) to check if there is only one possible tag value for that particular tag variable. If there is, we will avoid doing a fetch of the tag and just insert a unit operation instead. An example of this is shown in figure 4.15,

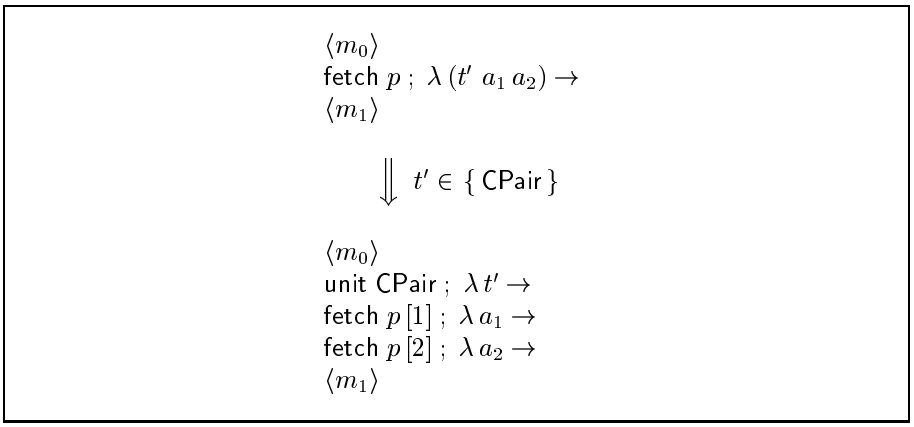

Figure 4.15: The split fetch transformation using tag information.

where we assume that the only possible tag value for the variable "t'" is known to be the tag CPair.

Note that we can *not* simply delete the binding of "t'" like we did in figure 4.14, since the tag might be used somewhere inside "$\langle m_1 \rangle$". If it is not used, though, the "unit CPair" operation and the binding will be removed later by the *dead code elimination* transformation (see section 4.3.15 on page 153), so it is not a problem to err on the safe side here.

Returning fetch operations

There is one situation where the above transformation is not directly applicable, and that is when we find a "returning fetch", i.e., a fetch operation with no code "after". This problem is essentially the same as what we ran into earlier with a "returning *eval*" that made updates a bit problematic (see section 4.2.3 on page 95). Recall that all fetch operations in our GRIN code originate from the inlining of the *eval* procedure (see for example figure 4.2 on page 85). Sometimes the heap points-to analysis will give a result saying that for one particular call to *eval* the argument to *eval* already points to a node representing a whnf. In this case, some of the optimising transformations described later will remove both the case expression and the update operation from the original *eval* body (compare with the *eval* procedure in figure 4.2 on page 85). The only thing that will be left of the *eval* is a fetch operation. In this situation, if we started with a "returning *eval*", we will after some transformations be left with a "returning fetch".

The way we handle this is to do the same thing as for the "returning *eval*", i.e., we insert a scrutinising case expression. In the same way as for updates, we can use the results of the heap points-to analysis to create this case expression (to

find what tags to include). Fortunately, as in the update case, the overhead of an extra case expression can in many cases be eliminated by other transformations (see the discussion in section 4.2.3 on page 95). A relevant issue is also the next transformation, the *right hoist* fetch transformation (see section 4.2.8 on the facing page), which is closely related to the *split* fetch transformation.

Architectural considerations

We said above that we will always avoid loading an already known tag. This is probably a good idea in general, but it is not completely obvious that it always is. Consider again the code in figure 4.15, and assume that the variable "t'" is really used in "$\langle m_1 \rangle$", and that the tag is not otherwise "available" in the code. With the above method, the operation "unit CPair" is used to put the tag value in a "register". If we assume that tag values are integers, as in the current GRIN implementation, the unit operation might need up to three machine instructions when translated into real machine code. E.g., this could be the case on the Alpha architecture, if the integer was sufficiently large. On most RISC architectures the unit would probably translate into two instructions (except for very small integers).

Contrast the above with if we instead had loaded the (already known) tag from memory using "fetch p [1]", an operation which would normally translate into a single load instruction. Assuming that the node is already in the first level *cache memory*, such a load may very well take only a single machine cycle to execute, i.e., it could actually be faster than doing the two or three (dependent) instructions needed to fill a register with a constant. Even if the node is not in the cache, the cost of the tag load may be very small since we are just about to load the node arguments anyway (the next few words in memory), so we will just get the *cache miss* a little earlier. In fact, some RISC processor conventions include a dedicated *global pointer register* that is used to hold the memory address of an area with certain commonly used global variables, exactly because loading a global address (which is just a large constant) has a relatively high cost.

In most cases however, modern computer architectures are not really bound by the number of instructions they execute. Other things like *fine grain parallelism* and *memory usage patterns* tend to be much more important. In the above case, we could reduce the total *memory bandwidth* (of data, not instructions) by using a unit rather than a fetch operation. Our basic assumption will be that it is favourable to avoid using memory operations whenever possible, even at the cost of a few extra ALU instructions (which are comparably "cheap"). This is particularly important in programs written in lazy functional languages, which normally require much more memory to execute than imperative programs. So, all methods to reduce the memory needs for the GRIN back-end will hopefully be an overall win.

4.2.8 Right hoist fetch operations

Part of the motivation, besides simplifying the code, for the *split* fetch transformation is to prepare for the following, to *right hoist* (i.e., move to the right) fetch operations. This is done primarily to avoid doing unnecessary memory loads, but also to adhere to some requirements of the current GRIN implementation (see below).

Recall the example in figure 4.13 on page 103. When discussing the meaning of the bindings to the variables "a_1" and "a_2" in the figure we said that semantically the variables can be bound to undefined values in some situations. E.g., this could happen if the tag variable turned out to be a CNil tag, which has no arguments. This is all fine semantically, but in an actual implementation we should of course try to avoid unnecessary load operations, for efficiency reasons. This means that the argument fetch operations should be *postponed* until we know that their value will be used.

In GRIN, the above observation really means that we should move some fetch operations into the alternatives of the associated case expression (i.e., the case expression that examines the tag associated with the fetch operation). Additionally, in each case alternative we should only load the components that are actually used in the body of that alternative. A safe approximation to this is to load exactly as many components as the arity of the corresponding tag (we can actually do a little better than this, see below). As described earlier, the vectorisation transformation (see section 4.2.4 on page 97) guarantees that the rest of the argument variables are not used inside the alternative. An example illustrating the transformation is shown in figure 4.16 on the next page.

For the transformation to be valid, the variables "a_1" and "a_2" in the figure must not be used inside "$\langle m_1 \rangle$" (this is enforced by earlier phases of the back-end). Note also that when doing the transformation we introduce new variable names in each case alternative, and apply a corresponding substitution to the rest of the code. This is not strictly necessary to preserve the semantics, but we do it to keep the GRIN *static single assignment* property (see section 2.3.7 on page 34).

When doing the fetch hoisting transformation, we will for each "fetch sequence" examine the associated case expression (it it exists) to decide which of the fetch operations hoist. We do this by checking each case alternative to see which of the loaded variables that appear in the code of the case alternatives (i.e., they *may* be used). If a variable can be used in *all* alternatives, we will *not* move that particular fetch. If a certain variable is not used in all alternatives, the associated fetch will be moved, but only into the alternatives where the variable actually appears.

One might argue that we should not "stop" the hoisting once a fetch is moved into a case alternative body, but instead continue to move it until we definitely know that the variable will be used. It is possible that this could be a win in some cases, but this has not yet been investigated. It is not certain that doing

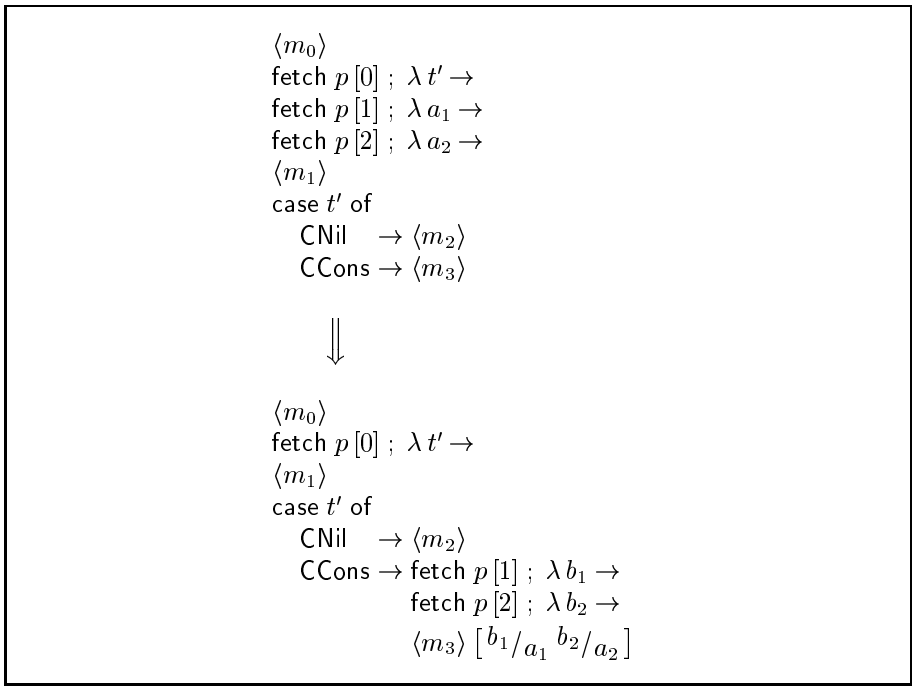

Figure 4.16: The right hoist fetch transformation.

so would be a win though (see the discussion about *pre-fetching* below).

If we can not find an "associated case expression" we must have a situation where the tag is already completely determined (at least in the current GRIN back-end), and the right hoisting transformation will not be applied.

Hoisting vs. pre-fetching

Although it may sound reasonable (or even obvious) to avoid loading values that will never be used, it is in fact not clear that the fetch hoisting transformation on the whole is a beneficial transformation.

In most modern computer architectures (using *pipelining*) the issue of *memory latency* has become quite important. Because of speed differences between the CPU and the memory subsystem, it will normally take "a while" between when a memory load instruction have been issued and when the result of the load is available to the CPU. The standard technique to *hide* this latency is to try to execute some other (independent) instructions inbetween the issue of the load instruction and the use of its result. A compiler can help doing this by emitting load instructions a bit before the point in the code where the actual

value is needed. This is called (software) *pre-fetching*. In some cases a compiler can emit a load instruction even if it is not absolutely certain whether the result will be needed or not (for example just before a case expression, for a value that is only used in some of the alternatives). This technique is called *speculative* pre-fetching. Even if we may load some unnecessary values, speculative pre-fetching can still be an overall win because the CPU will spend less time "waiting" for the result of memory loads to appear. Of course, if we load unnecessary values too often, that cost will outweigh any potential benefits.

Now, consider again the example in figure 4.16 on the preceding page and assume that the CCons alternative is much more common than the CNil one. In this case it may in fact be best *not* to move the fetch operations, and instead speculatively load the two argument values before the case expression. We would sometimes load the two arguments unnecessarily, when the tag was a CNil, but the overall effect could be a win because of the reduced time waiting for the result of the loads (of course, this also depends on how "early" the two arguments are used in "$\langle m_3 \rangle$").

On the other hand, looking at *cache memory behaviour*, if the tag has just been loaded, the rest of the node value is probably already in the first level cache, so subsequent loads will be cheap. A *cache miss*, if any, occurs probably already for the tag load. This indicates that the right hoist fetch transformation might very well be beneficial after all.

The best thing we can do to decide whether the transformation is profitable or not on the whole is probably to run benchmarks, on a large number of "real" programs. Unfortunately, the result of such an experiment will also depend on the particular architecture used and its memory characteristics. If we look at a single case expression, the result will depend on the relative frequencies of the different alternatives. One would expect that *profiling information* could be of good use in such a situation, to guide the transformation to select the most profitable alternative.

In addition to "hoisting all" vs. "hoisting none" one could also imagine "hoisting some", i.e., leaving some fetch operations outside the case expression and moving the others into the case alternatives. Another question is whether the "order" between the fetch operations should be preserved. Currently we always preserve the order, in the sense that if a fetch of a certain offset is moved, then so are also all with higher offsets.

The fetch hoisting transformations is somewhat related to the *let floating* described in [San95], where a let binding can be moved inside the alternatives of a case expression. However, there is no way to "split" a let binding, so that transformation is "all or nothing". This is a situation where we benefit from the extra low level control that GRIN provides compared to a more traditional functional intermediate code, like the STG code.

Specialised fetch operations

A fact that we have so-far ignored is that some (but not all) fetch operations need to be *annotated* with a concrete tag (to be used by the RISC code generator). This is exactly the same problem that appears for update operations during the update specialisation transformation (see section 4.2.3 on page 90). As with all GRIN operations which reference node values in memory, the implementation of the fetch operation depends on the actual node layout in memory, which in turn varies with different tags. Hopefully though, a large part of the fetch operations will just use the "default layout" and need not be annotated. Exactly which fetch operations that need to be annotated depends on the *node size limit*, which is what decides between the different node layout schemes. All this is explained in detail in section 5.2.3 on page 178. An important property of the method used is that the tag fetch (offset zero) never needs to be annotated, the "layout" of the tag is always independent from the layout of the rest of the node.

The fetch hoisting together with the split fetch transformation (see section 4.2.7 on page 102) are responsible for the annotation of fetch operations. We will however not always write out these annotations since they are not really necessary from a GRIN semantics point of view, they are only necessary for the RISC code generator. Also, it is normally "obvious" from the context what the tag is. E.g., in figure 4.16 on page 108 the two argument fetch operations (after the transformation) could really have been annotated with a CCons tag.

The need to annotate certain fetch operations with a concrete tag does in fact restrict the "freedom" for the right hoisting transformation. The fetch operations that need to be annotated *must* always be hoisted into the case alternatives, otherwise we do not know the tag to annotate with. This restriction can be weakened when we know that all of the tags appearing in the case expression follows the same node layout scheme (again, see section 5.2.3 on page 178).

4.2.9 Register introduction – naming all values

The *register introduction* transformation has two purposes: to simplify the GRIN code, and to give a *name* to all values. It is in part a simplification intended to make it easier for the RISC code generator to handle GRIN code, and in part a transformation that is a precondition for many of the *optimising* GRIN transformations (see section 4.3 on page 112). A typical example of a transformation that benefits from there being a name for all values is the *common sub-expression elimination* (see section 4.3.14 on page 148). We use the term "register introduction" since the intention is that GRIN variables and machine registers are approximately the same thing at "low level" GRIN, where this transformation is applied.

The transformation to name all values is done via the introduction of new lambda bindings and unit operations. After this transformation all explicit node values in the code will only contain variables, and arguments to function appli-

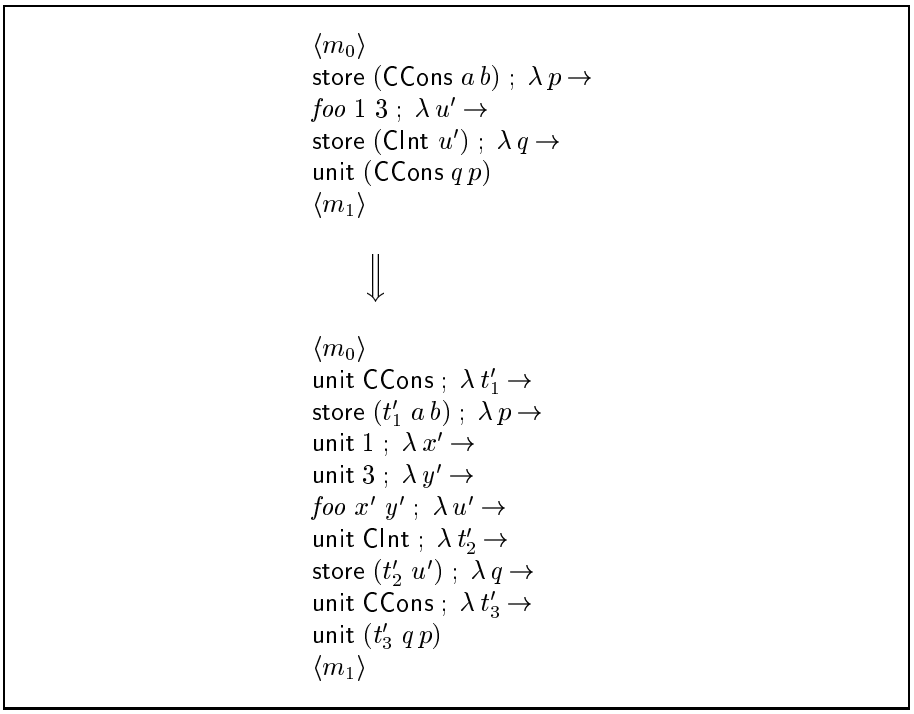

Figure 4.17: Register introduction.

cations (or primitives) will only be variables. The only place where constants (tags and other basic values) are allowed is as argument to the unit operation (and of course, still in patterns in case expressions). Some examples of this are shown in figure 4.17.

Note that this transformation does not attempt to "reuse" tags or basic values that have already been given a name. This will be done later by the normal common sub-expression elimination.

Judging from the transformation overview in figure 4.1 on page 80, the register introduction (in the box labelled "simplifications part III') would be run only once, and after the majority of the optimisations. However, as we said above, the register introduction is in fact an important precondition to many optimisations, and therefore it is also run together with all the optimisations (part of the box labelled "optimisations" in figure 4.1).

4.3 Optimising transformations

As described earlier (see section 4.1 on page 77) the aim of the *optimising* transformations is a bit different than the aim of the *simplifying* transformations. Whereas the simplifying transformations are used to translate the program from "high level" to "low level" GRIN, the optimising transformations are really just what the name implies, optimisations. This also means that in our setting the simplifying transformations are a necessary step of the compilation process, whereas the optimising transformations may or may not be applied, and can also be repeated.

4.3.1 Overview

Among the optimising transformations there are several different "kinds" of transformations. Some of them are pretty "standard", either from conventional compiler technology, or more specialised "functional" optimisations. Even if a transformations in principle is a well known optimisation, it might look rather different when expressed in GRIN terms. We will try to explain when this is the case.

Other transformations are more tailored to the needs of the GRIN back-end, and often motivated by previous GRIN transformations which when applied leave unnecessary code behind. An example of this is the GRIN code in figure 4.4 on page 88 where, after inlining *eval*, the procedure contains a lot of unnecessary code. This will be "cleaned up" by a series of mostly non-standard optimising transformations.

Yet another kind of transformations is mostly like well known optimisations, but achieve some "additional power" because of special properties of the GRIN language. Often this power is due to the extra low level control that GRIN offers, at least when compared to a normal functional intermediate language (which is often what is used as a basis for program transformations). A typical example of this is the GRIN separation of *node values* into an explicit *tag* and a sequence of *arguments*, which is used to represent both closures and weak head normal forms. E.g., this can be used to introduce *tagless return values* (see section 4.3.3 on page 118). A transformation like that is not possible in for example the STG language [PJS89]. Instead a corresponding optimisation is used at a lower level (or "built in" to the runtime system). It is also possible to inspect a GRIN value (that can represent a closure) without "evaluating" it, something which is not possible in a functional language.

The optimising transformations are run repeatedly and "in-between" the sequence of simplifying transformations (see the overview in figure 4.1 on page 80). Most optimisations are applied to relatively "low level" GRIN code (mainly, after the *vectorisation* transformation, described in section 4.2.4 on page 97).

4.3.2 Copy propagation

Copy propagation is a well known conventional optimisation [ASU86, section 10.2]. It is sometimes also called *copy elimination*, *subsumption* or *coalescing*. Interestingly, it turns out that copy propagation in GRIN is exactly the same as applying the *monad unit laws* [Wad90, Wad92].

In theory

The purpose of copy propagation is to remove unnecessary assignments (copies) between variables by eliminating one of the variables. Note that in general this can only be done if the copy really is unnecessary, i.e., it is possible to eliminate one of the variables by replacing all uses of it with the other variable, without changing the meaning of the code.

In GRIN there are no direct assignments, instead we have monad operations and lambda bindings, but we can still formulate a copy propagation transformation. The closest we can come to a copy in GRIN is the unit operation together with a lambda binding. However, it turns out that we get two, rather different copy propagation transformations, depending on if the unit operation is on the *left* or on the *right* side of the lambda binding. This is not a coincidence, the two variants of copy propagation laws in GRIN correspond exactly to the left and right unit laws for monads. All monads must satisfy these laws. Wadler [Wad92] have formulated the two laws as:

- left unit law: $(unit\ a)\ `bind`\ k = k\ a$,
- right unit law: $m\ `bind`\ unit = m$.

Since all GRIN expressions are built using the builtin GRIN monad (see section 2.2.1 on page 26), it is not surprising that the monad laws can be applied to GRIN. The two monad unit laws, formulated as GRIN program transformations, are shown in figure 4.18 and figure 4.19 on the next page.

The GRIN transformations are not identical to Wadler's formulation of the laws because the GRIN syntax requires a lambda binding after each bind operator (see the syntax in figure 2.1 on page 28). It is however easy to verify the GRIN transformations using the Wadler monad laws. E.g., the GRIN left unit transformation can be derived as:

$$\begin{array}{l} \mathsf{unit}\ v\ ;\ \lambda u \to \langle m \rangle \\ \qquad \Downarrow \text{ Wadler's left unit law} \\ (\lambda u \to \langle m \rangle)\ v \\ \qquad \Downarrow\ \beta \text{ reduction} \\ \langle m \rangle\ [{}^{v}/_{u}] \end{array}$$

The right unit transformation can be verified in a similar way.

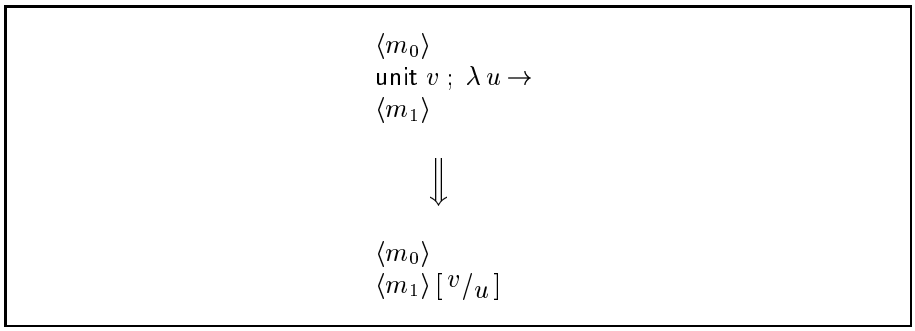

Figure 4.18: Copy propagation (left unit law).

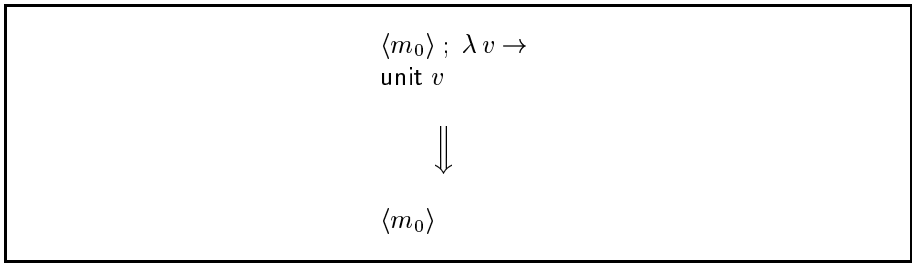

Figure 4.19: Copy propagation (right unit law).

Section 4.4.1 on page 156 describes another monad law that can be used to formulate a useful GRIN program transformation, the *monad associativity law*.

In practice

Note that the transformations are very simple when the "copy" is between two variables, as in the figures. However, since GRIN also allows explicit nodes both as arguments to unit operations and in lambda patterns, things get a bit more complicated in practice. Although the transformations themselves are unchanged, there are two other things that must be taken care of. First, the substitution (in the left unit law) may need to be extended to handle several variables (entire node values). Second, we must be careful not to lose *tag information* (see section 4.4.4 on page 160) that can not be recovered.

Exactly what happens when an explicit node is involved in a copy depends on the following:

- if the copy is a *left* or a *right* unit (figure 4.18 or 4.19),
- if the left (or right) hand side of the copy is a node variable or an explicit node,
- if the lhs (or the rhs) is an explicit node, it also depends on if the node tag is a known tag or a tag variable.

Unfortunately, this will result in 18 different cases, all with slightly different rules for what happens (what substitution to create and what tag information to preserve). We will not give all cases here, instead we will go through some of the left unit rules, to show the general ideas. The right units are similar. For the examples below, compare with the general left unit transformation in figure 4.18.

First, let us look at the case where we have two explicit nodes, the left with a known tag and the right with a tag variable:

$$\mathsf{unit}\ (\mathsf{CInt}\ x')\ ;\ \lambda\,(t'\ y') \rightarrow \langle m_1 \rangle$$

Here, we must substitute both the variables "t'" and "y'". It would be wrong to just substitute for the complete node "$(t'\ y')$" following the general rule of figure 4.18, because the variables can appear on their own inside "$\langle m_1 \rangle$". If we only substituted the complete node and deleted the binding (the copy), we could be left with unbound variables inside "$\langle m_1 \rangle$". The correctly transformed code is instead:

$$\langle m_1 \rangle \left[\mathsf{CInt}/_{t'},\ x'/_{y'} \right]$$

Since the explicit tag CInt replaces all occurrences of the variable "t'" we do not risk losing any tag information in this case. Note also that if the code above

had used an explicit tag instead of the variable "t'", then it must of course have been a CInt tag, otherwise the code would always fail.

Another example is when we have an explicit node to the left and a node variable to the right:

$$\mathsf{unit}\ (t'\ a\ b)\ ;\ \lambda v \rightarrow \\ \langle m_1 \rangle$$

This one is easy, we can just substitute for "v". The resulting code will be:

$$\langle m_1 \rangle \left[(t'\ a\ b) /_v \right]$$

However, we might also have to add information to the tag information table (see section 4.4.4 on page 160). If the variable "v" had known tag information (e.g., because there is a case expression scrutinising "v" inside "$\langle m_1 \rangle$"), then that information must now be transferred to the variable "t'" instead.

Yet another copy situation can appear if we have a variable on the left and a node pattern on the right (i.e., the opposite of the above):

$$\langle m_0 \rangle \\ \mathsf{unit}\ v\ ;\ \lambda\,(\mathsf{CFoo}\ x') \rightarrow \\ \langle m_1 \rangle$$

Unfortunately, a copy like this one can in general *not* be eliminated using the left unit law, and there are two reasons for this. First, there is nothing we can substitute for the variable "x'" if it appears on its own, inside "$\langle m_1 \rangle$". Second, even if we could guarantee that we would always find the entire node "(CFoo x')" inside "m_1", so that we could replace the node by "v", doing so could lead us to a situation where we lose tag information that we can never recover. In the general case it is *not* safe to add tag information (here CFoo) to "v", because that might not be true for all points in the program. E.g., the copy can be "inside" a case expression, scrutinising "v" itself, which means that inside the other alternatives "v" will have different tag values! The tag information is defined in such a way that, if present, it must always include all possible tags. Unfortunately, this means that we can not add tag information to "v" in the above case, and must refrain from doing the transformation. Of course, this restriction does not apply if CFoo is the only constructor of a product datatype, in that case it would be safe to add tag information.

As a side note, if the tag information had not been defined in that way it would be virtually useless for our transformation purposes, because we could never "trust" it. If the tag information gives a single possible tag for a certain variable, we must be able to trust that this information is true everywhere in the program (where the variable is in scope, of course).

Returning to the example above, even if we do not eliminate the copy at this stage, we can probably do it after the *vectorisation* transformation (see

section 4.2.4 on page 97). After vectorisation "v" will be replaced by an explicit node, and we will certainly be able to eliminate the copy to "x'". The tag copy may still need to be kept to avoid losing tag information, but even that can be eliminated by running an extra pass of the copy propagation at the end of the GRIN transformation process, where tag information can safely be ignored. In addition, when a left copy fails there is a good chance that a right copy propagation can be made instead. Above, this would mean that the last binding in "$\langle m_0 \rangle$" together with the unit operation could form a right unit and be removed.

As the example above demonstrated, the GRIN copy propagation may not always be applicable to a "copy" in the code, because of the risk of losing tag information. However, this is the only reason for the transformation not to apply. We can never get a situation (like in imperative code) where a copy can not be deleted because the two variables are in "conflict" with each other. The reason for this is the GRIN *static single assignment* property (see section 2.3.7 on page 34).

Transformation strategy

The copy propagation is important to run often, and repeatedly, during the GRIN transformation process, for two reasons:

- many other transformations insert "copies" that can easily be eliminated by the copy propagation,
- the copy propagation itself can also give rise to more opportunities for copy propagation. Sometimes, long sequences of "copies" appear, and it is not always the case that a single run of the copy propagation pass can remove all of them.

Currently, the copy propagation is implemented as a single top-down "walk" of the GRIN expression tree. This is the reason that it may not eliminate all (possible) copies during a single run.

Confluence and phase ordering

The following is a kind of "copy", but we will not treat it as a valid copy in our copy propagation sense:

$$\mathsf{unit}\ 1\ ;\ \lambda x' \rightarrow \langle m_1 \rangle$$

It would be possible to treat this as a copy, and as a result of copy propagation substitute the basic value "1" for the variable "t'" inside "m_1", but that is currently not done by the GRIN back-end. Doing that would basically be what is normally known as *constant folding*, i.e., to substitute all known constants

into the code instead of variable names, a transformation that is often a part of *constant propagation*. Unfortunately, this is exactly the reverse transformation of our *register introduction* (see section 4.2.9 on page 110). If we had included both we would get problems with the *confluence* of the transformation system. Since the register introduction is very important for many other transformations (introducing names for all values), we have chosen to not include constant folding. And it does not really matter if constant folding is done separately, as long as we do *constant propagation* (see section 4.3.12 on page 143). The confluence problem is also related to the *common sub-expression elimination* (see section 4.3.14 on page 148).

4.3.3 Generalised unboxing

In this section we will describe a transformation that in some sense is a generalisation of the well known *unboxing* transformation[PJL91, Ler92a]. In particular, we will show how unboxing in GRIN can be "split" into two separate transformations, which we will call *loading* and *untagging*. As a result of this we will be able to partly apply unboxing in many more situations than with previous methods. We will even be able to partly "unbox" non-strict function arguments (we can always *load* an argument, but not always *untag* it).

Boxes and tags

The term unboxing is normally used to describe a change in data representation, from a *boxed* to an *unboxed* representation. The unboxing transformation is mainly concerned with function arguments and return values, and tries to change them from a boxed to an unboxed representation. E.g., a boxed integer is represented as a pointer to an object in memory (the box). This object contains "more information" than just the integer itself, normally some way to tell that the object is actually an integer. The GRIN back-end uses *tags* that are small integers to identify objects in the heap, but the unboxing method we are about to describe does not really depend on that particular implementation choice.

An *unboxed* representation of an integer is normally just the integer value itself kept in a machine register, although some use the term even for integers that are *tagged* (with a special tag bit).

If we look at the "representation change", from boxed to unboxed, it is useful to view it as a two-step process. In the first step, the entire heap object is loaded into registers, i.e., the pointer is eliminated (one level of *indirection*). We call this step *loading*.[1] In the second step, the tag is eliminated, leaving a single integer in a register as the final unboxed representation. We call this second step *untagging*.

[1] Actually, we would like to use the term *lifting* for this, since we imagine a node value being *lifted* from the heap and into registers. However, that term is often used for an operation that is much the inverse of what we mean, so we use *loading* instead.

Shortcomings of traditional unboxing

In most implementations of functional languages, the unboxing is run at a rather "high level" intermediate code, often some kind of functional language. Unfortunately, this means that the unboxing is not applicable in all situations. First of all, only strict function arguments can be unboxed since the unboxing of a value enforces evaluation. Another problem is that the set of datatypes to which unboxing can be applied is quite limited (in many implementations it is only applied to "numbers"). Formally, the restrictions on the datatype are that the type must first of all be a *product type*, i.e., it must only have one possible constructor. Moreover, the type must contain only one "sub-component". E.g., an integer is ok, but a pair is not, since the two sub-components of the pair does not fit in a single register. The latter restriction can often be worked-around using a technique called *arity raising*. E.g., a strict function argument that is a pair can be transformed into two arguments, one for each component of the pair. However, arity raising will not work for function results. A function that returns a pair can not be "unboxed", to return the two components independently. There simply is no way to express in a "functional" code that a function returns "two values" without using a datatype to pack the components. Instead, such an optimisation have to be done on a lower level, where enough "control" is available. It can also be achieved as a result of implementing clever calling conventions. Even if it "looks like" the intermediate code is returning a pair, the actual machine code will not do that, it will just return the two components of the pair using two registers.

Unboxing in GRIN

It is our opinion that GRIN offers just enough "low level control" so as to avoid all of the above shortcomings. In particular, the control of data representation (nodes and tags) in GRIN is exactly what is needed to get rid of some of the ad-hoc restrictions of ordinary unboxing (and arity raising).

In GRIN, we explicitly distinguish between pointers, node values and basic values. These three kinds of values correspond exactly to the "two-step process" of unboxing, as we described above. If we start with a boxed value (a pointer), we can use the GRIN fetch operation to load the entire node value into "registers" (a node variable). This is the *loading* step (elimination of the pointer). After that we can, if possible, also eliminate the tag (*untagging*). If an integer is untagged, the result is a basic value. If a pair is untagged the result is the two components of the pair. These representation change steps are visualised in figure 4.20 on the next page.

Strict product datatypes. For product datatypes, the tag is always unnecessary, in some sense. If we apply loading and untagging to a value of a product datatype, we can get three different results, depending on the number and type

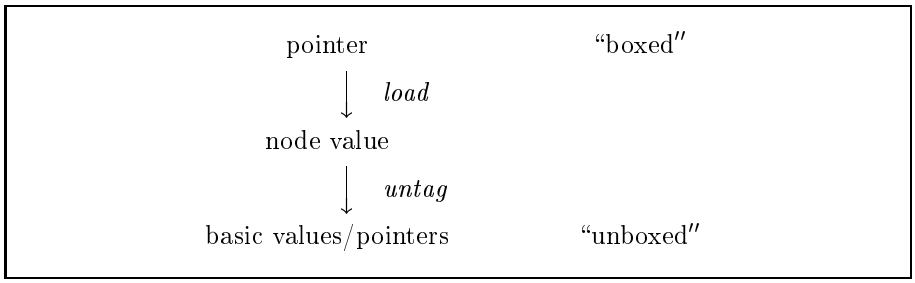

Figure 4.20: Generalised unboxing – representation changes.

of its sub-components. If the type has one unboxed sub-component, the result after untagging is an unboxed value. If the type has one boxed sub-component, the result is a pointer. And finally, if the type has several sub-components, the result will be a "tagless node value". We have not yet described what a tagless node value is, but it would be very easy to add such a notion to the GRIN language. We could define a tagless node to be nearly the same thing as an ordinary node value, a sequence of values (see section 2.3.3 on page 30), except that the first value in the sequence (the tag) is the *empty* value ("()"). This is completely non-problematic to allow in GRIN, as long as such a value is not written to the heap. A tagless node can be kept in registers without problems. The GRIN syntax in figure 2.1 on page 28 does not allow an empty value in the "tag position", but it would be a trivial change to add that.

This solution solves one of the problems mentioned above, that "multi-component" values can not be returned as function results inside the language, and instead have to be optimised later in the compilation.

Non-strict arguments and sum datatypes. The really nice thing about unboxing in GRIN is that we can achieve at least some of the effect of unboxing for *all* types, we are not restricted to strict arguments of product types (with a single sub-component). First note that in GRIN a pointer argument can *always* be *loaded*, i.e., turned into a node variable (even non-strict arguments). This is the first half of unboxing.

In the same way as for product datatypes above, we can sometimes also *untag* sum datatypes (or non-strict values), if the heap points-to analysis has been able to deduce that only one tag is ever possible. For non-strict values, this tag will then be an F-tag, representing a closure. Untagging such a value means that since we "know" what the function to call is, there is no need to keep the tag. We need only pass the function arguments, and then perform the call. This is similar to what can happen in the *split fetch* transformation, e.g., in figure 4.15 on page 105, a tag that is already known will not be loaded at all. If the analysis is unable to give a single known tag, we must keep the tag, but

have still achieved "part" of the unboxing due to the loading.

In all the above cases we can also choose to apply *arity raising*, instead of keeping a node argument. Note that this decision is independent of if untagging have been applied or not! If untagging have not been done, the tag will simply become a new function argument just as any other value.

Unboxing decisions. Above, we have shown how the GRIN low level control over data representation offers increased possibilities for various forms of unboxing. However, we have not really said when to apply the loading and the untagging. It is probably not a good idea to apply the methods "as much as possible". As an example, consider a function argument that is a pointer. Since we have the complete program available we could in principle load the argument, and turn it into a node variable. After doing that we must of course also change all calls (and building of closures of that function), to use a node value rather than a pointer to a node. If this would force us to insert fetch operations in many places to load the value from the heap, it is not clear that the transformation would be beneficial. On the other hand, if the node value is already available at all places where it is needed the transformations might be more profitable.

We have not yet found a single good algorithm for all these kinds of "more advanced" decisions. Below we show a transformation that applies to one special case: function return values that are of a product datatype. Later, in section 4.3.13 on page 144, we show a GRIN variant of *arity raising* that can achieve much of the above, but only for function arguments. However, it seems like it should be possible to combine all this into a single transformation, doing both *loading* and *untagging*, for both function arguments and results, and for all kinds of data (for both product and sum types, and for closures). But we have not yet found a nice way to do that.

An algorithm – untagging return values

The compiler front-end currently used (see figure 1.3 on page 17 for an overview of the compiler), implements a method similar to the *worker/wrapper* unboxing method [PJL91]. This means that there is no pressing need for this "normal" kind of unboxing in the GRIN back-end. However, to show the feasibility of unboxing in GRIN we have implemented a specific case that can not be handled by the front-end unboxing. What we do is to *unbox return values* for all product datatypes. Or really, since function return values are by default already *loaded* in the GRIN calling convention (see section 2.5.3 on page 46), what the transformation will do is really to *untag return values*.

To explain the transformation, we will show a simple example of how a function that originally returns a boxed integer can be changed to return an unboxed integer instead. Datatypes with "more arguments" follow analogously,

by introducing "tagless nodes" as described above. Note that unboxing a returned integer can also be done by the front-end unboxing, but not datatypes with more arguments (we show the integer case because it is simpler).

The transformation is done in three steps.

Step 1: find functions to unbox. When finding what functions to apply the return unboxing to we have to be careful with tailcalls. Tailcalls can be a bit problematic because the unboxing transformation may risk transforming them into ordinary returning calls. This should be avoided in some situations, but can sometimes be allowed (see below).

The algorithm is based on a "candidate set" (of functions to unbox). The set is initially large and then shrinks as some candidates are disqualified. We start with the set of all functions that return a known product datatype (these are the initial unboxing candidates). Note that we can always find the actual "return tag" for a (non-polymorphic) function that returns a product datatype using *tag information* (see section 4.4.4 on page 160). We then disqualify all functions that tailcall a function outside the candidate set (i.e., it tailcalls a function that will not be unboxed). Without this we would lose tailcalls. A similar situation appears when a function outside the candidate set tailcalls a function inside the set. However, here we have decided to allow such calls. By restricting tailcalls "out of" the set, we have eliminated the most important situation where it might be dangerous to lose a tailcall, when the call can be part of a "recursive loop" of successive tailcalls, going "in and out of" the candidate set. We can still lose some tailcalls, but we believe that to be less important. In "stack space" terms, using this method we can never transform a "constant stack" computation into a computation where the stack usage increases by the size of the "input". For each "iteration" in a loop, we will never increase the stack usage by more than a constant amount compared to the original program. After this, the candidates that still remain in the set will be unboxed.

Step 2: transform function returns. After the functions to unbox have been found in step 1, we will examine all return sites inside those functions. In GRIN, most values are returned using unit operations. This together with the fact that the unboxing is run after the *vectorisation* transformation (see section 4.2.4 on page 97), means that the actual tag is always visible in the returning unit operation, so we can simply remove it. If we assume that a function *foo* returns a boxed integer, the transformation will create a new function *foo′*, identical to *foo* except that it returns an unboxed integer instead. I.e., for all return sites in *foo* that match:

$\langle m_0 \rangle$
unit (CInt x')

we will create a corresponding untagged return site in *foo′*:

$\langle m_0 \rangle$
unit x'

If it had been the case that the vectorisation had not been run, we would still be able to untag the return site. In this case the function might have returned using a node variable, i.e., the return site would match:

$\langle m_0 \rangle$
unit v

where "v" is a node variable. Now we could instead create an extra binding to get an untagged return (in *foo′*):

$\langle m_0 \rangle$
unit v ; λ (CInt x') $\rightarrow$
unit x'

The "extra copy" will later be removed by the *copy propagation* transformation (after the *vectorisation* is done).

If we were to unbox the return value of a function where the returned value had more than one component, e.g., a pair, we could still do this with a slight abuse of the GRIN syntax. Similarly to the above, we could do:

$\langle m_0 \rangle$
unit v ; λ (CPair $a\, b$) $\rightarrow$
unit (() $a\, b$)

The "()" denotes the special GRIN *empty value* (see section 2.3.4 on page 32), and is used to signal that we have a node value without a tag. This kind of usage has not yet been implemented, and it is not shown anywhere else in the GRIN transformations, but it would certainly be possible to do this.

Besides unit operations, return sites can also be tailcalls. However, if a tailcall is found during the unboxing process, the decision we made in step 1 means that both the caller and the callee in such a situation must be in the "unboxed set". A tailcall will be created in the unboxed version of the function to simply tailcall the unboxed version of the callee. E.g., if *foo* tailcalls *bar*, the tailcall we create in *foo′* will be to *bar′*.

Step 3: transform function calls. After having created "unboxed versions" of all functions to be unboxed, we will go through the entire program and change all calls to the unboxed functions. Continuing the previous example, calls to *foo* anywhere in a GRIN expression:

foo $a_1 \ldots a_n$

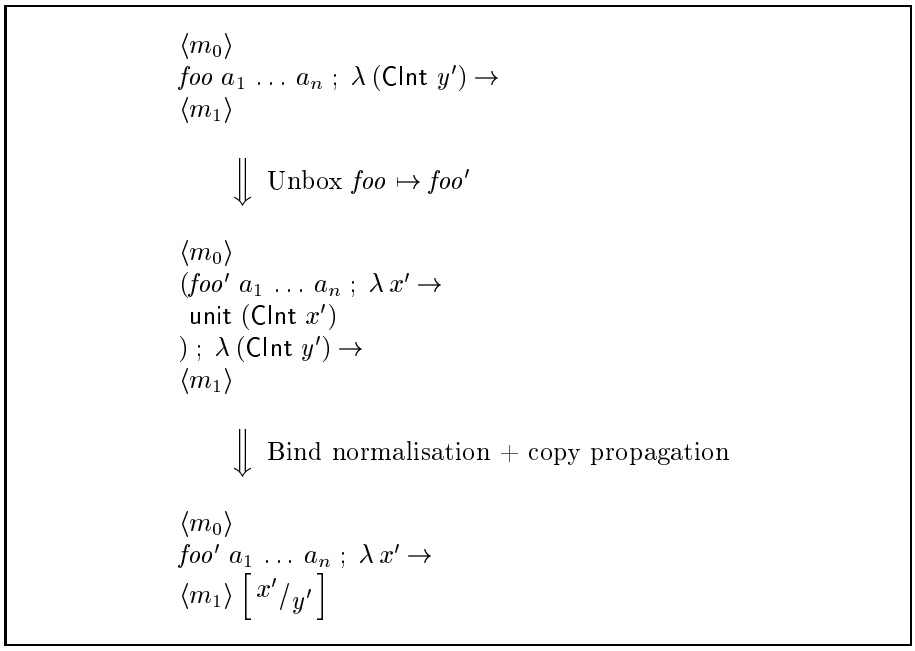

Figure 4.21: Unboxing of function return values.

will be changed to:

foo′ $a_1 \ldots a_n$; $\lambda\, x' \rightarrow$
unit (CInt x')

Note that the above call can not be a tailcall, due to the handling of tailcalls in step 2 (the call would have been changed to *foo′* if it had been a tailcall). As above, the extra unit operation can in most cases be eliminated by the copy propagation. Function calls are typically followed by a lambda binding, in which case the copy propagation directly applies. An example of this is shown in figure 4.21. Before the copy propagation can be applied we must also apply the GRIN *bind normalisation* (see section 4.4.1 on page 156).

The figure shows how the resulting code have completely eliminated the "tagging overhead" associated with the return value of *foo*.

As said earlier, this unboxing of return values and the *arity raising* (see section 4.3.13 on page 144) are the two special cases of our generalised unboxing that has been implemented in the GRIN back-end. More work is needed to find methods to take advantage of the other possibilities discussed above.

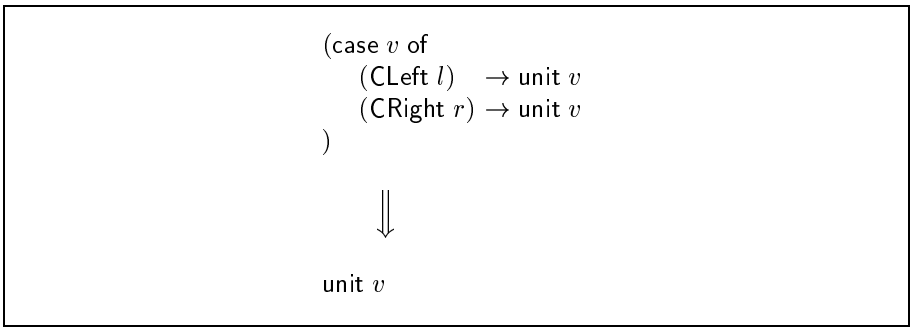

Figure 4.22: Evaluated case elimination.

4.3.4 Evaluated case elimination

This transformation is the first in a series of transformations optimising different aspects of case expressions. Recall the special GRIN *eval* procedure, e.g., in figure 4.2 on page 85. During the *eval* inlining transformation described earlier we will *specialise* the body of the *eval* procedure, and in particular the case expression inside *eval*. In the resulting code, the case expression will only include tags that really can occur during an execution. All this is done according to the results of the heap points-to analysis (see chapter 3).

Sometimes the specialisation will result in a case expression where all tags are so-called C-tags (representing whnf data structures, see section 2.3.3 on page 30). The right hand sides of all C-tag alternatives in the *eval* case expression will always consist of GRIN unit operations that simply return the examined value. Since all alternatives return the scrutinised value, the whole case expression is unnecessary and can be removed. In the presence of higher order functions the same phenomenon occurs for P-tags (representing partial applications, which are also whnfs, see section 2.3.3 on page 30).

To summarise the transformation: A GRIN case expression without F-tags and where all right hand sides are equal to "unit v" where "v" is the scrutinised expression, is removed and replaced with a single unit operation returning the same value. An example of this is shown in figure 4.22.

Note that the transformation will not change the semantics of the code, since case expressions in GRIN does not "force evaluation" like for example in the STG language.

Note also that we can not simply delete the case expression, the transformed code must "return" the same value as the original case expression. The resulting unit operation will however in most cases be trivial to remove. Normally, as in the body of the *eval* procedure, the case expression is followed by a lambda binding, which means that a simple *copy propagation* (see section 4.3.2 on page 113) can eliminate the unit operation. This is illustrated in figure 4.23.

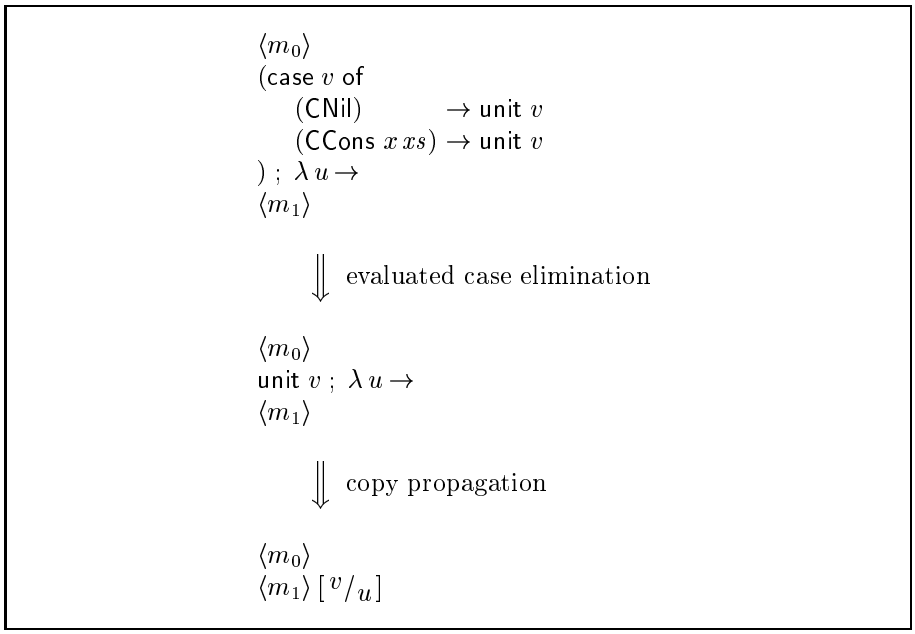

Figure 4.23: Evaluated case elimination with context.

The variable "u" in figure 4.23 can in fact be any kind of GRIN value, e.g., an explicit node. However, if it is an explicit node, like "(CInt x')", we must be a bit careful so that we do not lose *tag information* that can not be recovered. This was discussed in the section about copy propagation (see section 4.3.2 on page 113).

After this transformation possibilities for *update elimination* (see section 4.3.8 on page 132) will often appear.

4.3.5 Trivial case elimination

The *trivial case elimination* is very similar to the previous transformation, the *evaluated case elimination*, in that it will remove "unnecessary" case expressions, but of a slightly different kind.

Sometimes, the *eval* and *apply* inlining give rise to case expressions with a single alternative, but which is not caught be the previous transformation. There can be two reasons for this, either the tag is an F-tag (representing a closure), or the right hand side of the alternative does not match the simple unit operation that the evaluated case elimination requires. Which reason it is depends on if the case expression originates from an inlined *eval* or an inlined *apply* procedure. It is also possible that due to some other transformation we

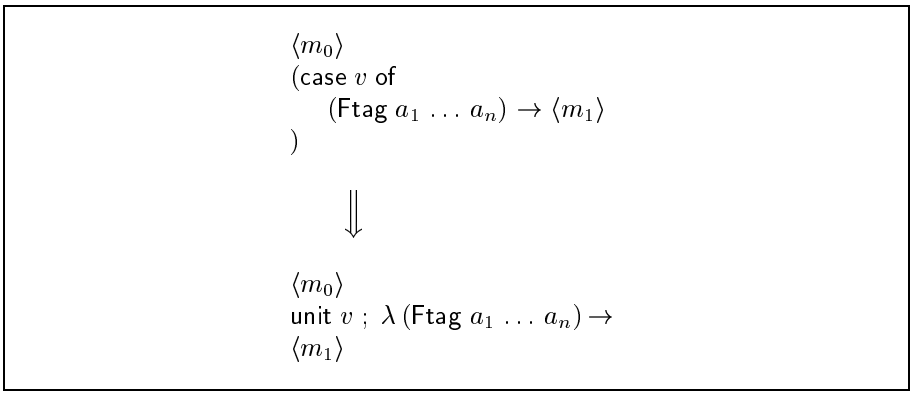

Figure 4.24: Trivial case elimination.

end up with a single alternative case expression, independently of *eval* and *apply* inlining. In any case, there is no need to perform a case "test" when there is only a single alternative, and the transformation can remove the case expression as shown in figure 4.24.

Note that this transformation is slightly trickier than the previous one. We must make sure that the *bindings* to the variables "$a_1 \ldots a_n$" are not lost by inserting an extra unit operation followed by a pattern matching lambda binding with the same pattern as in the original case expression. However, as in the previous transformation the extra unit operation can often be eliminated by the copy propagation.

Yet another form of "unnecessary" case expressions exist, and that is when the scrutinised tag (or the contents of a scrutinised variable) is completely known. This will be taken care of by the *constant propagation* optimisation (see section 4.3.12 on page 143).

4.3.6 Sparse case optimisation

The third optimisation regarding case expressions is closely related to the heap points-to analysis (see chapter 3). The previous two optimisations could in principle transform arbitrary case expressions, that matched the requirements, but they were both mainly intended to attack case expressions originating from inlined *eval* and *apply* procedures. This transformation does instead concentrate on "normal" case expressions, i.e., case expressions that were written by the programmer (or possibly inserted by the compiler). The idea is to use information about possible tags derived by the heap points-to analysis to delete "impossible" alternatives in case expressions. This is very similar to the *specialisation* done as part of the *eval* inlining (see section 4.2.1 on page 86), but applied to

"normal" case expressions instead. This may sound a bit strange at first, when considering case expressions written by a programmer. A programmer would normally not write code with "impossible" alternatives. However, it is considered good programming practice to take even "impossible" situations into account, and at least produce an error message rather than having the program crash in an uncontrolled way (the compiler can also insert "no match" error messages). Also, it is common to write "general" functions and then reuse them as much as possible, even if that might result in a general function being called only with a "limited" set of possible data for some programs. This is especially true after optimisations and program transformations. In particular, optimisations such as *cloning* and *inlining* can make "impossible" alternatives appear much more often.

The *sparse case* optimisation is currently run only once, just after the heap points-to analysis itself (i.e., it is run before the *eval* and *apply* inlining). The reason for this is just an implementation issue. The analysis result is currently returned as information for each *eval* and *apply* "point". This information consists of a mapping from a variable name (the argument in the *eval* or *apply* call) to the actual information. For an example, see the analysis result in figure 3.4 on page 68. Unfortunately this means that after the *eval* and *apply* inlining and other optimisations it is very difficult to track exactly what information is available at a certain point (variable names might change, etc.).

Due to the above reason, the current implementation of the sparse case transformation will have to match a compound GRIN expression: a call to *eval* followed by a scrutinising case expression. An example of this is shown in figure 4.25 on the next page. Here we have assumed that the analysis has deduced that the variable "v" containing the result of evaluating "l" can never be a CNil node. Since the case expression after the transformation has only one possible alternative, the previous *trivial case elimination* applies. This step, and a succeeding *copy propagation*, are also shown in the figure.

A careful reader might have observed that the sparse case optimisation can be seen as a kind of *constant propagation*, where a *set* of possible values is propagated rather than a single constant. In the current GRIN framework we have implemented constant propagation (see section 4.3.12 on page 143), but only a simple version propagating single constants. If we had implemented a more general set based variant, it would probably have been easy to make it subsume the sparse case optimisation.

4.3.7 Case copy propagation

The *case copy propagation* is in some sense a kind of "unboxing" of case expressions, and in some sense a more expensive variant of the normal *copy propagation* (see section 4.3.2 on page 113). In particular, the result of this transformation is very similar to the copy propagation *left unit law*, where a unit operation to the left of a binding is eliminated. In this case though, we will not have a

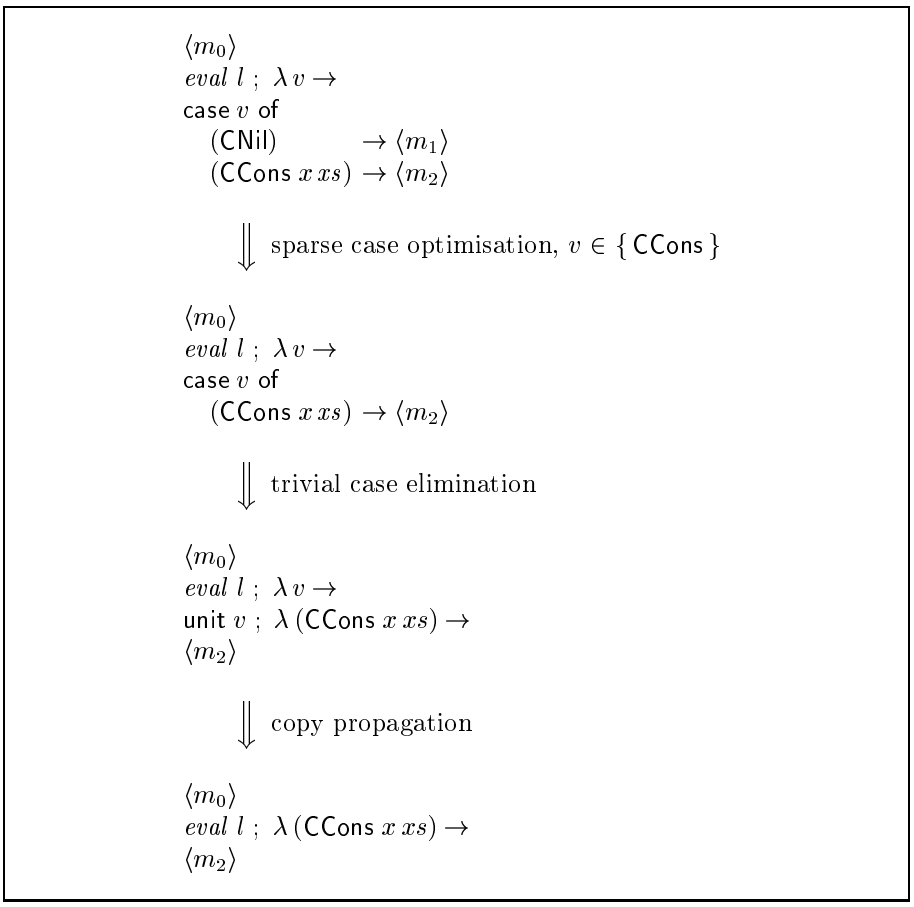

Figure 4.25: Sparse case optimisation.

single left unit, but instead "many left units". This can happen when a GRIN case expression is put on the left hand side of a binding, and each alternative in the case expression "returns" to the outside binding using a unit operation. As an example, consider a case expression where all alternatives return an explicit integer node value:

```
case v of
  ⟨pat_1⟩ → ⟨m_1⟩
            unit (CInt x'_1)
     ⋮
  ⟨pat_n⟩ → ⟨m_n⟩
            unit (CInt x'_n)
```

Here, we can do a trick very similar to what we did when unboxing function return values transformation (see section 4.3.3 on page 121). We "unbox" (or really "untag") all the unit operations that "return" from the case expression and instead insert a single unit after the case expression. For the above example, we would do:

```
(case v of
   ⟨pat_1⟩ → ⟨m_1⟩
             unit x'_1
      ⋮
   ⟨pat_n⟩ → ⟨m_n⟩
             unit x'_n
) ; λ y' →
unit (CInt y')
```

The transformed code will behave exactly as the untransformed code did, it will return an entire integer node. Note that the transformation can only be applied when *all* the alternatives of the case expression return exactly the *same tag*, using a unit operation. Of course, this is more likely if the value returned is of a product datatype, like the integer case above.

Fortunately, this is a quite common situation in GRIN programs, and it happens especially for case expressions that are the result of inlined calls to *eval* (see section 4.2.1 on page 84) combined with the unboxing of return values (see section 4.3.3 on page 121). The example in figure 4.26 on the next page tries to illustrate this using a sequence of transformations.

The case expression in the figure originates from an inlined call to *eval*. The functions *foo* and *bar* return boxed integers, and are in the first step unboxed, and becomes *foo'* and *bar'*. After this, the case expression matches the condition of the case copy propagation, all the alternatives "end" with a unit operation returning a CInt node. After the case copy propagation, we also show a step of the "normal" copy propagation, to show how the code can be "cleaned up".

$\langle m_0 \rangle$
(case v of
  (Ffoo a) $\rightarrow$ $foo\ a$
  (Fbar b) $\rightarrow$ $bar\ b$
  (CInt x') $\rightarrow$ unit (CInt x')
) ; $\lambda u \rightarrow$
$\langle m_1 \rangle$

$\Downarrow$ unbox return values: foo and bar

$\langle m_0 \rangle$
(case v of
  (Ffoo a) $\rightarrow$ $foo'\ a$; $\lambda y' \rightarrow$ unit (CInt y')
  (Fbar b) $\rightarrow$ $bar'\ b$; $\lambda z' \rightarrow$ unit (CInt z')
  (CInt x') $\rightarrow$ unit (CInt x')
) ; $\lambda u \rightarrow$
$\langle m_1 \rangle$

$\Downarrow$ case copy propagation

$\langle m_0 \rangle$
((case v of
  (Ffoo a) $\rightarrow$ $foo'\ a$; $\lambda y' \rightarrow$ unit y'
  (Fbar b) $\rightarrow$ $bar'\ b$; $\lambda z' \rightarrow$ unit z'
  (CInt x') $\rightarrow$ unit x'
) ; $\lambda v' \rightarrow$
unit (CInt v')
) ; $\lambda u \rightarrow$
$\langle m_1 \rangle$

$\Downarrow$ bind normalisation + "normal" copy propagation

$\langle m_0 \rangle$
(case v of
  (Ffoo a) $\rightarrow$ $foo'\ a$
  (Fbar b) $\rightarrow$ $bar'\ b$
  (CInt x') $\rightarrow$ unit x'
) ; $\lambda v' \rightarrow$
$\langle m_1 \rangle \left[(\text{CInt } v')/u \right]$

Figure 4.26: Case copy propagation.

The copy propagation is able to eliminate all the unnecessary unit operations, both inside the alternatives and after the case expression. Actually, before the copy propagation can be applied we must also use the *bind normalisation* transformation (see section 4.4.1 on page 156), to restructure the code slightly before it matches the requirements of the copy propagation.

Given the final result of the transformation, an alternative name for the case copy propagation could have been "case unboxing".

Implementation

When implementing the transformation we need to, for each case expression: first examine the return sites of all the alternatives, then decide if the transformation applies (all returns must be similar units), then possibly transform all the return sites, and finally continue transforming the code in each alternative. Note that even if the transformation applies to a particular case expression, there can still be more "candidates" inside the alternatives of the case expression. In a naïve implementation, this can easily result in a quadratic algorithm, but it can be made linear using several passes. The transformation can even be done in a single pass and linear time using techniques for *circular algorithms* [Bir84, Joh87a].

4.3.8 Update elimination

The *update elimination* is yet another in the line of transformations that is intended to "clean up" after the *eval* inlining (see section 4.2.1 on page 84). The case expression resulting from an inlined *eval* procedure can sometimes be completely eliminated by the *evaluated case* transformation (see section 4.3.4 on page 125). This happens when the argument to *eval* was already evaluated, so the call to *eval* was really unnecessary. And if the *eval* was unnecessary, then so must the update operation be, since all update operations originate from inlined calls to *eval*. For an example, see the *eval* procedure in figure 4.2 on page 85. When the *eval* case expression has been eliminated, it will result in a fetch operation immediately followed by an update, i.e., the heap node is immediately overwritten with itself. Clearly, this is unnecessary, and we can simply remove the update operation without changing the behaviour of the code. An example of the transformation is shown in figure 4.27 on the facing page.

In addition to the update elimination, we describe two other transformations that deal with update operations, the *update specialisation* (section 4.2.3 on page 90) which handles updates where the written value is unknown, and the *whnf update elimination* (section 4.3.9 on the facing page) which tries to avoid updating values that are already on weak head normal form.

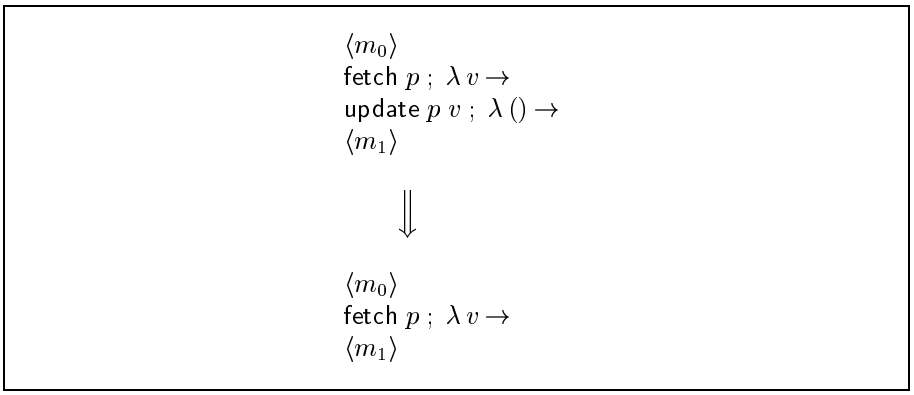

Figure 4.27: Update elimination.

4.3.9 Whnf update elimination

The intention of the *weak head normal form update elimination* is to avoid doing an update when a value which was the subject of a call to *eval* turned out to be already evaluated. The transformation is the third in the line of transformations involving update operations, after the *update specialisation* (see section 4.2.3 on page 90) and the "normal" *update elimination* (see section 4.3.8 on the facing page). It is meant to catch situations where the other two did not apply. To recapitulate, the "normal" update elimination is used when a case expression resulting from an inlined *eval* has been completely removed, so that the update is clearly unnecessary (see figure 4.27). The update specialisation on the other hand is used when the update operation is located "inbetween" two case expressions, the first being an inlined *eval* and the second being an "ordinary" scrutinising case expression. In this situation, the update must be specialised, i.e., moved into the branches of the second case expression (see figure 4.6 on page 93).

If none of those transformations apply, we have a situation where the following holds:

- the value *before* the evaluation can represent a closure (but also a whnf), i.e., the *eval* case expression could not be eliminated,
- the value *after* the evaluation has a known tag, i.e., there is no scrutinising case expression after the update, just a lambda binding.

An example of this is shown in figure 4.28 on the next page (see the code before the transformation).

The inlined *eval* in the figure will eventually result in an integer node, i.e., the only possible tag is CInt. The case expression includes three cases, an already

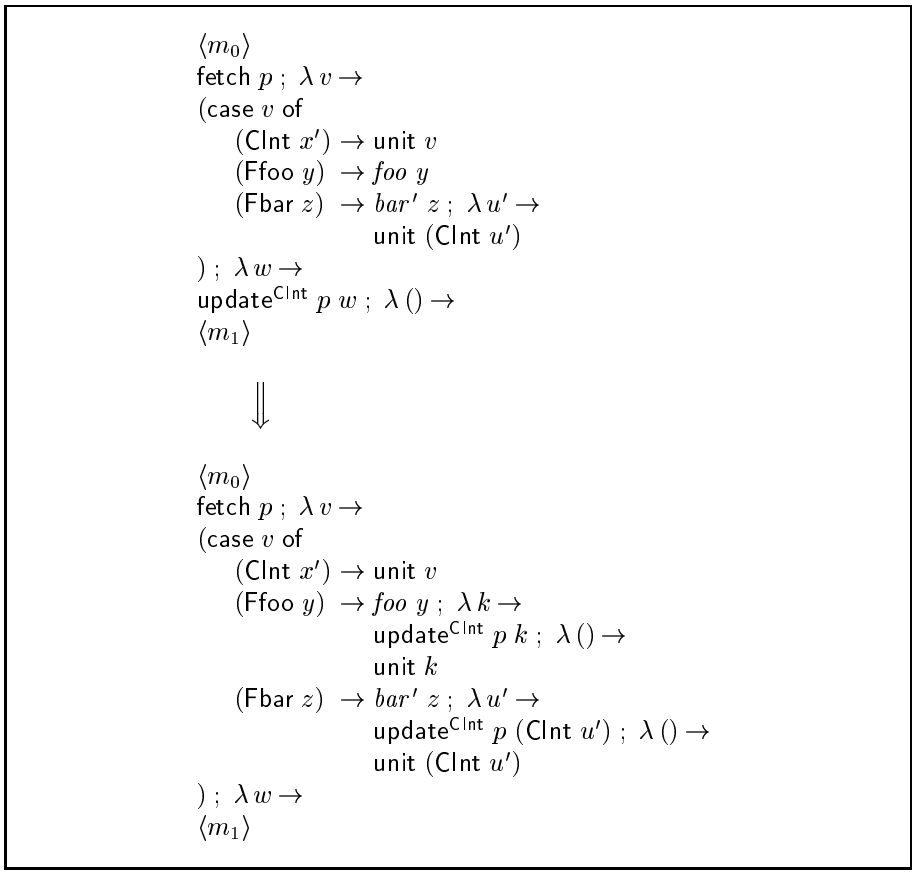

Figure 4.28: Whnf update elimination.

evaluated integer and two closures. The function *bar* has had its return value *unboxed* (see section 4.3.3 on page 118), which explains the right hand side of that alternative.

Keeping the update after the case expressions means that we will also perform an update when the value was already a CInt node. To optimise this, the transformation will move the update into the preceding case expression, but only into the closure alternatives, as is shown in the figure. Note that in general it would *not* be possible to already from the beginning put the updates in only the necessary branches of the *eval* case expression, since some of them must be *specialised* or we will not be able to generate code for them later (see the discussion in section 4.2.3 on page 90). It is only in cases where the update specialisation was not needed that the transformation in figure 4.28 can be applied.

Sharing analysis

Above, we can do even better when moving updates into a preceding case expression. If the heap points-to analysis (see chapter 3) deduces that some closures are *unique* (not shared), then we do not need to insert `update` operations for those alternatives.

GRIN control flow restrictions

Unfortunately, this transformation is not able to eliminate all updates on weak head normal forms. The reason is that in situations where the update specialisation transformation has already moved the `update` into a following scrutinising case expression, we can not also move it into the preceding case expression. With the current GRIN design, there is nothing (viable) we can do about this on the GRIN level, and the problem has to be solved when translating to RISC machine code. Fortunately, this can rather easily be done by giving special treatment to sequences of case expressions in the RISC code generator (see section 5.3.2 on page 189).

4.3.10 Late inlining

Technically our *late inlining* transformation is not really any different from traditional inlining. What is interesting about it is instead the "late time" of the compilation process when we apply it. In particular, the GRIN late inlining is applied *after* inlining the *eval* and *apply* procedures (see sections 4.2.1 on page 84 and 4.2.2 on page 89, respectively). This means that all control flow (or really, all function calls) have been made explicit in the code, i.e., turned into direct calls instead of being indirected through *eval* and *apply*. After some further GRIN transformations, this will often result in inlining opportunities that were not visible earlier during the compilation process. Compared to other implementations of lazy functional languages, that do not eliminate calls to *eval* (or its equivalent), the GRIN back-end will see many more inlining opportunities arise, simply because many more function calls are "visible" to the optimiser. Or in short: the GRIN late inlining will be able to inline function calls that did not even look like function calls when the more normal "front-end inlining" were done.

To motivate the above claim we will go through two examples. The first is very simple and the second a bit more involved. First, consider a very simple functional expression:

$$x + 1$$

When translating this into "real code", in any implementation of a lazy functional language, it will have to result in the evaluation of the variable "x" (unless that

was already done). In GRIN we would get:

$$\begin{array}{l} \mathit{eval}\ x\ ;\ \lambda\,(\mathsf{CInt}\ x') \rightarrow \\ \mathit{intAdd}\ x'\ 1\ ;\ \lambda\, y' \rightarrow \end{array}$$

The interesting thing happens when the call "*eval* x" is inlined. For any suspended computation (closure) that "x" can be bound to there will appear a function call to that function (see the example *eval* procedure in figure 4.2 on page 85). Of course, these function calls (that were not visible before the *eval* inlining) can also be inlined. Most of the calls should probably not be inlined, but at least now we have a choice.

Our second example of how "late" inlining opportunities arise is taken from a real program, the *queens* program discussed earlier (see figure 3.7 on page 72). The function *foo*, shown in figure 4.29 on the next page, is an auxiliary function created by the compiler during the *lambda lifting* phase [Joh85]. It has a (functional) definition that is just an application of the first argument to the second:

$$\mathit{foo}\ f\ x = f\ x$$

The reason for creating this rather strange function is that a function application of an *unknown function* (e.g., a higher order argument) appeared in a *lazy context* in the original code. Our implementation, as many other, can not build closures of unknown functions. This is taken care of already by the compiler front-end which creates the *foo* function to take care of the application, and then build closures of *foo* (which is now a known function).

When translated to GRIN the function will result in a call to *eval* followed by a call to *apply*, as shown in figure 4.29. For the queens program, however, the heap points-to analysis (see chapter 3) will be able to deduce that the variable "f" is always the same. Or really, the analysis will deduce that "f" always points to a node in the heap with the same tag, a P-tag representing a partial application of the *ok* function (see figure 3.7 on page 72). I.e., the *eval* will be unnecessary, since a P-node already represents a weak head normal form, and consequently the *apply* will always call the *ok* function. As a result, after inlining *eval* and *apply* and some further simplifications we will be left with just a call to the *ok* function. This is shown in figure 4.29.

At this point, *ok* can be inlined, something that was *not* possible in the original functional code! In this particular case, the *foo* function should probably also be inlined, but that is an independent problem. The unused "f" parameter will be removed later by the *dead parameter elimination* (see section 4.3.16 on page 155).

Implementation

Hopefully, the above examples showed how new inlining opportunities arise "late" in GRIN. The actual inlining is done in a fairly standard way, though.

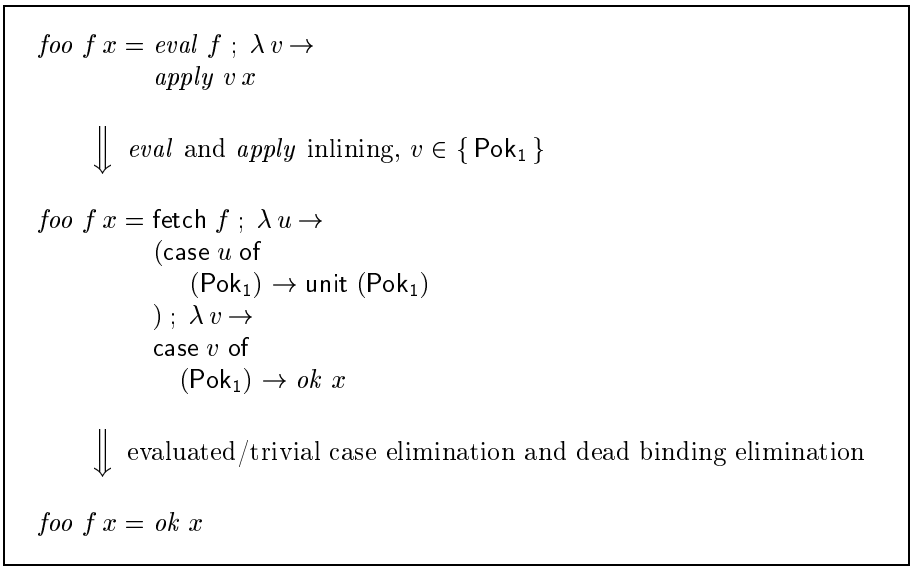

Figure 4.29: An example of how late inlining opportunities arise.

We inline functions that satisfy at least one of the following:

- the function is *called only once* (and non-recursively),
- the function body is *smaller* than a certain "size" limit.

The "size" of a GRIN expression is defined to be the number of "binds" inside it.

As for traditional inlining, one of the most useful things about the GRIN late inlining is that it will often create new opportunities for other transformations, and in particular the *case hoisting* (see section 4.3.11), the *constant propagation* (see section 4.3.12 on page 143), and the *arity raising* (see section 4.3.13 on page 144). In fact, these optimisations taken together can sometimes produce really amazing results, similar to the effects of *deforestation* [Wad88] (see the transformation example in appendix A).

4.3.11 Case hoisting

The purpose of the *case hoisting* transformation is to avoid doing case expression "tests" on values that are already known. As a special case, this transformation will be able to avoid repeated tests on the same value. This situation often appears after *inlining*, both the inlining of the special *eval* procedure and of "normal" GRIN functions (see sections 4.2.1 on page 84, and 4.3.10 on page 135, respectively).

The problem

Consider the following very common situation, a function call returning a node value followed by a scrutinising case expression:

$$
\begin{array}{l}
\mathit{foo}\ ;\ \lambda\, v \rightarrow \\
\mathsf{case}\ v\ \mathsf{of} \\
\quad (\mathsf{CNil}) \qquad\ \ \rightarrow \langle m_1 \rangle \\
\quad (\mathsf{CCons}\ x\ xs) \rightarrow \langle m_2 \rangle
\end{array}
$$

Now assume that we are going to inline the call to *foo*. Also assume that it turns out that *foo* will always return a CCons node, using a unit operation. If that is the case, inlining *foo* would result in something like:

$$
\begin{array}{l}
(\langle m_{foo} \rangle \\
\ \mathsf{unit}\ (\mathsf{CCons}\ a\ as) \\
)\ ;\ \lambda\, v \rightarrow \\
\mathsf{case}\ v\ \mathsf{of} \\
\quad (\mathsf{CNil}) \qquad\ \ \rightarrow \langle m_1 \rangle \\
\quad (\mathsf{CCons}\ x\ xs) \rightarrow \langle m_2 \rangle
\end{array}
$$

After this, the code can be normalised using the *bind normalisation* (see section 4.4.1 on page 156), and the unit operation removed by the *copy propagation* (see section 4.3.2 on page 113). Once that is done, the *constant propagation* (see section 4.3.12 on page 143) can easily find that the case expression is unnecessary and remove it.

However, in the general case it is not very likely that a function returns a single possible tag. At least not if it is returning a value of a *sum* datatype (the only situation where there will be a scrutinising case expression following the call, as above). Instead, most functions can return a number of different tags. Note however that these tags must be the same as (or a subset of) the tags in the scrutinising case expression.

For an example of this more general situation, consider the code in figure 4.30 on the facing page. The first case expression in the figure is typically the result of an inlined call to *eval*. Code like this is very common, and can appear as the result of either an *eval* followed by a scrutinising case expression, or an inlined ordinary function call followed by a case expression. The important thing to note in the figure is that two of the alternatives of the first case expression return a "known" tag, either CNil or CCons. This tag is then bound by the variable "*u*" and examined by the second case expression. Clearly this is not optimal, and we should try to avoid it. Note also that the Fbar alternative can return either a CNil or a CCons node, which means that the second case expression is not always "unnecessary".

In some sense, this can be seen as the problem with the *control flow* inside GRIN expressions which we discussed earlier (see section 4.3.9 on page 135).

```
⟨m0⟩
(case v of
    (CNil)          → unit (CNil)
    (CCons x xs)    → unit (CCons x xs)
    (Fbar a)        → bar a
) ; λ u →
case u of
   (CNil)          → ⟨m1⟩
   (CCons y ys)    → ⟨m2⟩
```

Figure 4.30: An inlined function followed by a scrutinising case expression.

We could imagine the CNil and CCons alternatives of the first case expression "jumping" directly into the corresponding alternatives of the second case expression, to avoid the extra test. But that is not possible in GRIN, since it would completely break the structure of GRIN code. At least not if we want to keep the code as "one GRIN expression". It would be possible to split up the code into several functions, and use calls to *continuation functions* as "jumps", but this has other disadvantages (see also the discussion in section 5.3.2 on page 189).

Fortunately, there is another solution to the problem in figure 4.30, a simple GRIN program transformation. This transformation will be able to eliminate unnecessary tests, as described above, possibly at the price of some code duplication. However, it is easy to control the amount of code that is duplicated by restricting the transformation.

The transformation

The *case hoisting* transformation is a special case of a more general GRIN monad law, shown in figure 4.31 on the next page. In the figure, part of a "continuation" is moved inside a preceding case expression, i.e., the binding of "u" and the code "$\langle m_1 \rangle$" are duplicated and put at the end of the code inside each case alternative. To keep the GRIN static single assignment property (see section 2.3.7 on page 34) we must also rename the moved code (this is not shown in the figure).

Note that we can arbitrarily choose "how much" code to move, i.e., the sizes of "$\langle m_1 \rangle$" and "$\langle m_2 \rangle$" can be chosen freely. Of course, we can also move all the code, in which case both "$\langle m_2 \rangle$" and the binding of "w" would disappear.

We have not yet found a good way to use this general monad law as a program transformation in the GRIN framework. It would need some guiding rules to determine when and how it should be applied, and for what reasons. However, in the special situation when the first part of "$\langle m_1 \rangle$" is a case expression, i.e., we

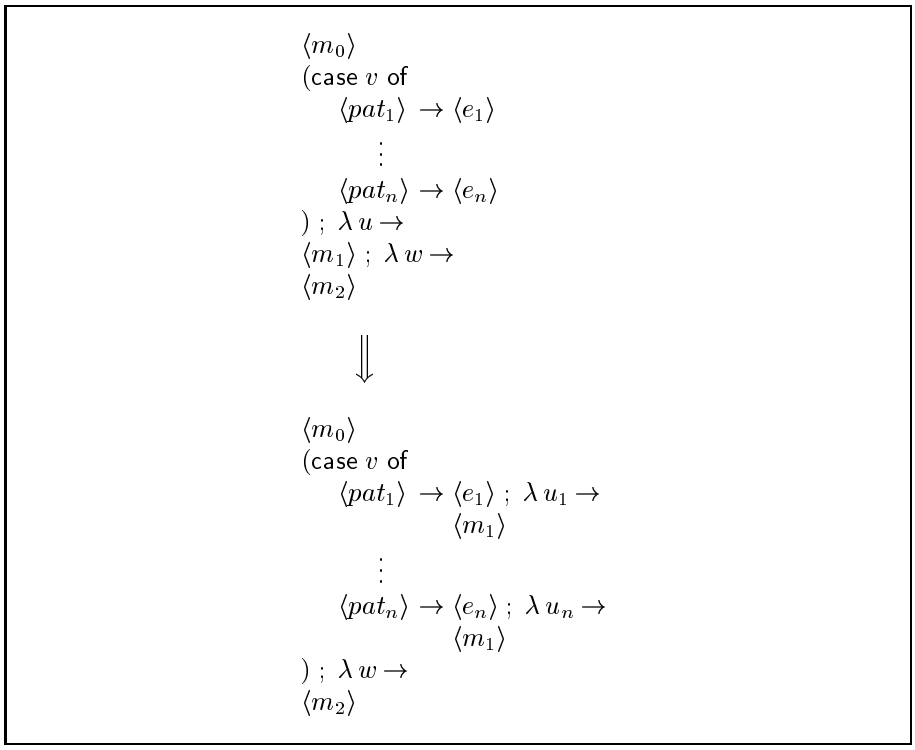

Figure 4.31: Monad based code motion in GRIN.

have two case expressions in a row, the law is very useful as a transformation. We call this *case hoisting*. If we return to figure 4.30 on the page before, we have exactly this situation. If we apply case hoisting and move the second case expression into the first we get the code in figure 4.32 on the facing page.

As above, we must also rename all variables in the moved code, inside "$\langle m_1 \rangle$" and "$\langle m_2 \rangle$" (this is not done in the figure). We also show a step of *copy propagation* (see section 4.3.2 on page 113) and *constant propagation* (see section 4.3.12 on page 143), because it is typically these transformations that makes the case hoisting worthwhile. In the final result the two "already evaluated" alternatives (CNil and CCons) have been short-circuited, and only one "test" is performed.

Known returns

The previous example (figures 4.30 and 4.32) originated from an inlined call to *eval*, but similar situations will also appear after inlining ordinary functions. It is important to note that the reason that the case hoisting succeeds (or really that the following constant propagation succeeds) is *not* that the two case expressions

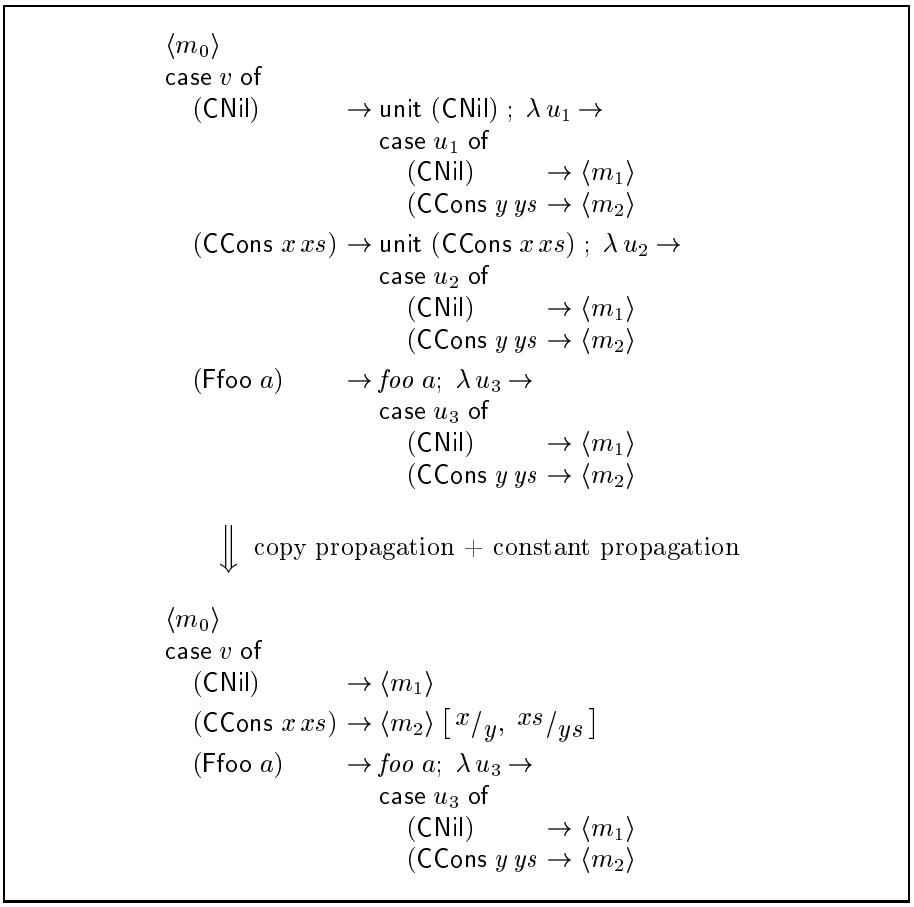

Figure 4.32: The result of case hoisting.

scrutinise the same tags (CNil and CCons). The reason is instead that some of the alternatives of the first case expression "return" a *known tag*!

In our example the first case expression just happened to return the same tag as it had scrutinised (in two of the cases), but that was a coincidence (because the case expression originated from an inlined *eval*). That need not be the true in the general case.

Limiting code duplication

As can be seen from the final result of case hoisting, in figure 4.32, both "⟨m_1⟩" and "⟨m_2⟩" have been duplicated. This may be ok in some situations but is not always acceptable.

The key concept that determines how much code that will be duplicated by the transformation is the number of alternatives with *unknown returns* in the first case expression. In the previous example (see figure 4.30), the Ffoo alternative had an unknown return value, and resulted in code duplication in figure 4.32, the entire second case expression had to be copied to the resulting Ffoo alternative. In general, also alternatives with known return values can result in code duplication, if some alternatives of the first case expression return the same known tag. However, the duplication in this case is smaller than for unknown returns, only the particular alternative of the second case expression will be duplicated, not the entire second case expression.

This means that is is quite easy to predict code duplication by examining all alternatives of the first case expression, to see what they return. In doing so we need to track the value of variables similarly to in the constant propagation. Currently the default setting in our back-end is to not allow any code duplication at all. It would probably be a good idea to combine case hoisting with the creation of *continuation functions*, to be able to do more aggressive case hoisting and still avoid excessive code growth, but we have not yet done this. One of the optimisations that we do in the RISC code generator also decreases the need to do case hoisting in situations where it would lead to code duplication (see section 5.3.2 on page 189).

When we have GRIN code with more than two case expressions in a sequence, it is not obvious in which order to apply the case hoisting for best results. Currently, we will perform it as a kind of "last first" strategy, by doing a bottom-up traversal of the GRIN expression tree, but there are situations where this is non-optimal. The problem gets harder when we have *nested* case expressions (which are not handled at all by the above algorithm). More work is needed to find the best way to attack this.

Related work

Our case hoisting is more or less the same as the "case of case" transformation reported by Santos in [San95], although it may look a bit different since in GRIN the two case expressions are sequentialised. Santos also have the ability to insert a kind of cheap continuations to avoid code duplication.

As a special case, the case hoisting can be used to implement *boolean short-circuiting* [Aug87] (where it is implemented as a code generation technique rather than as a program transformation).

The overall effect of the case hoisting transformation is similar to the *vectored return* convention of the STG machine implementation [PJ92]. The idea is that if a function returns a known value, then the caller should not immediately have to examine the value using a case expression. In the STG machine, a function can have multiple "return addresses", and it will return values in predetermined registers to the correct location, avoiding the extra test. In GRIN we can achieve a similar effect using a program transformation on the intermediate code. This

is in some sense "nicer" than having to rely on an implementation convention. However, our method requires that the callee is inlined (we need to find the two case expressions "together"), something which is not always possible (or desirable) for other reasons.

4.3.12 Constant propagation

The GRIN *constant propagation* [ASU86, section 10.2] is implemented in a very traditional way. Despite that, it is an important part of the GRIN back-end. In particular, it has one especially important task: it must be able to track constant tag values, in order to eliminate case expressions that scrutinise an already known tag. This situation often appears as the result of *inlining* (see section 4.3.10 on page 135) and *case hoisting* (see section 4.3.11 on page 137), so it is important to handle it well.

The constant propagation is implemented as a single top-down traversal of the GRIN expression tree. Currently, only single constants are propagated. I.e., a variable that can have more than one constant value will be considered "unknown". This simple method seems to work ok for our purposes.

As a "starting point" for the constant propagation, we use the *tag information* table that is continously kept updated throughout the transformation process (see section 4.4.4 on page 160). In particular, we take all "singleton tags" from the table, i.e., variables that are known to contain a single and known tag value. In some sense the tag information is a *static* and *invariant* version of constant information. A variable that has a singleton set in the tag information table will definitely have that tag as value in the entire program (where it is in scope of course). In addition to this static information, the constant propagation tracks *dynamic* information. In GRIN this is especially important for tag values and case expressions. As an example, consider the following:

$$
\begin{array}{l}
\mathsf{case}\ t'\ \mathsf{of} \\
\quad \mathsf{CNil} \ \ \ \rightarrow \langle m_1 \rangle \\
\quad \mathsf{CCons} \rightarrow \langle m_2 \rangle
\end{array}
$$

Here, if "t'" is already a known constant, the constant propagation can remove the case expression and only keep the relevant alternative. On the other hand, if "t'" is not known before the case expression, it can still be treated as "known" in each alternative of the case expression. Inside "$\langle m_1 \rangle$" it will have the value CNil, etc.

In addition to tag values, the constant propagation takes care of the usual situations with constant operations, like calculating primitive operations where all operands are constants, etc.

It should be noted that the GRIN constant propagation does not do *constant folding*, i.e., substituting all known constants instead of variable names in the code. The GRIN constant propagation is only applied to "fully constant" calcu-

lations. The reason for this is that constant folding could lead to problems with the confluence of the transformations system (see section 4.3.2 on page 117).

Note also that since constant propagation is done as a single top-down "walk" over the GRIN expression tree, no information is propagated from an "internal" case expression (i.e., a case expression on the left hand side of a bind operator, see section 2.3.2 on page 30) to the code after the case expression. It would be possible to take the *meet* of information found in the alternatives of the case expression, to propagate at least some information out of the case expression. In practice however, it seems that the way GRIN is structured makes this less important, probably much due to the *static single assignment* property (see section 2.3.7 on page 34).

4.3.13 Arity raising

The GRIN *arity raising* transformation is an example of a well-known traditional optimisation that acquires additional power in the GRIN framework. In GRIN we are able to also non-strict function arguments, i.e., arguments that are closures, without forcing them to be evaluated (and thus changing the argument into a strict one). The reason is that normal GRIN values are used to represent closures (so-called F-tags and F-nodes).

Traditionally, arity raising is a transformation used to transform a single function parameter into several parameters, when the parameter is known to contain a value of a known *product datatype* (only one constructor). In [App92] the transformation is called *argument flattening*. A typical example is when a function argument that is a pair is instead passed as two separate arguments, the two components of the pair. In a non-strict language, arity raising can only be applied to strict arguments since it forces the evaluation of the "top-level structure" of the value.

Due to the increased control of evaluation and of tags and nodes in GRIN, the only restriction we will put on when arity raising can be applied is that the argument must be bound to a node value with a single possible, and known, GRIN tag. This will include all values of product datatypes, but can also include values of *sum datatypes* (when we know for some reason that only a single tag can appear), and also nodes representing closures (when we know that a single F-tag can appear).

Normally, arity raising is performed only by examining a function and its arguments, without examining the *callers* of the function. Of course, callers that want to take advantage of the arity raising must be changed to call the new version of the function, but this need not be done at the same time as the arity raising. On the other hand, the program-wide interprocedural optimisation framework provided by the GRIN back-end makes it possible to also examine all callers of a certain function at the time of arity raising, to see if opportunities for the more advanced forms of arity raising arise.

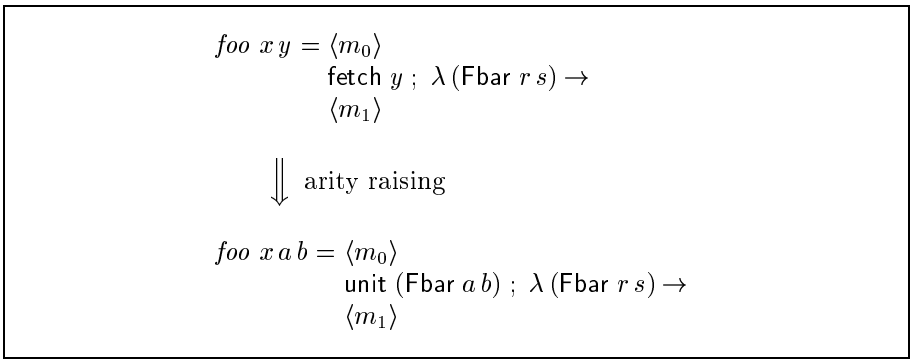

Figure 4.33: The result of arity raising on a closure argument.

An example

The power of the GRIN arity raising is illustrated in figure 4.33. Let us assume that all callers of the *foo* function will send a *bar* closure as second argument to *foo*, i.e., the variable "y" in the figure will always point to a node value with an Fbar tag. Also assume that the arity of the *bar* function is two. If that is the case, we can apply arity raising to the "y" argument of *foo* and turn it into two arguments instead, corresponding to the two arguments of the Fbar tag in the node that represents the *bar* closure. Note also how the fetch operation inside *foo* is changed into a unit operation at the same time. This results in a "copy" that can easily be removed by the normal *copy propagation* (see section 4.3.2 on page 113).

Since the arity of *foo* now is changed, a corresponding change must also be done to all call sites that call *foo*. It is not unusual that such a call is preceded by a store operation that creates the closure to be passed as an argument, in our case a *bar* closure. I.e., a call site could look like the following:

store (Fbar i j) ; $\lambda p \rightarrow$
foo z p ; λ ... $\rightarrow$

After arity raising of the second argument of *foo* this would be transformed to:

store (Fbar i j) ; $\lambda p \rightarrow$
foo z i j ; λ ... $\rightarrow$

After this, the store operation would probably be dead code (if "p" is not used again) and can be removed by the normal *dead code elimination* (see section 4.3.15 on page 153).

A special case, but a very important one, is when we have a call to *foo* with an *invariant* argument, i.e., somewhere inside the body of the *foo* function is a

recursive call:

foo z y

Note how the second parameter of *foo* is passed unchanged in the recursive call to *foo*. When arity raising is applied in this case it would result in:

foo z a b

As a result we now get two invariant arguments (compare with figure 4.33). This may seem as a small detail, but the fact is that this ability to do arity raising in the presence of invariant arguments, combined with the raising of closure arguments, can result in remarkable code improvements. It can sometimes transform "lazy loops" into strict and much more efficient loops (see the discussion below).

The algorithm

The critical question is how to find situations when arity raising can (and should) be applied. Since the GRIN framework offers us to examine all *call sites* where a certain function is called we can rather easily do this. The transformation is done in three steps:

1. Examine definitions of function arguments in all callers.
2. Examine uses of function parameters inside all callees.
3. Do the actual transformation.

Step 1: examine callers. The first part of finding candidates for arity raising is to examine all function calls in the program. E.g., we might have a call like this:

foo a

The first condition for being able to raise the argument of *foo* is that we must know exactly what node value the variable "*a*" is bound to, and that the the node value must be available (already fetched from the heap). One way to know this is the example above, where the result of a store operation is passed as a function argument (in that case we knew that the variable "*p*" was bound to an Fbar node and that the variables "*i*" and "*j*" could be used instead of the entire node). In addition to store operations there are other ways that we can "know" about the contents of a variable, e.g., inside case expressions. *Tag information* (see section 4.4.4 on page 160) can also be used to help finding candidates for arity raising.

After examining all function calls we sum up the information collected. For each function, and each argument position, only a single known tag must have been known in all its callers, for arity raising to be applicable.

Step 2: examine callees. The next step is to examine how the candidates for arity raising found above, i.e., some function parameters, are *used* inside the corresponding function bodies. Parameters that are to be raised are only allowed to be used in two ways: either as an argument to a fetch operation, or, as an invariant argument in a recursive call. The fetch case can be seen in figure 4.33. An example of a situation where arity raising would fail is the following code (assume that "p" is the parameter that is a candidate for arity raising):

$$\begin{array}{l} foo\ p = \langle m_0 \rangle \\ \qquad \mathsf{store}\ (\mathsf{CCons}\ p\, q)\ ;\ \lambda r \rightarrow \\ \qquad \langle m_1 \rangle \end{array}$$

Here, the pointer "p" is itself built into a data structure. If we were to apply arity raising to "p" in this function we would have to insert a new store operation to create the pointer needed. So, in this example "p" will not be considered a candidate for arity raising any further.

Step 3: transform. The last (and simplest) step is to actually perform the arity raising in the code, for the candidates that passed both checks above. We simply change all the raised function parameters, and substitute a unit operation for fetch operations on a raised parameter (see figure 4.33). Invariant arguments are also changed as shown in one of the examples above.

Note that we put no restriction on the actual node value that a raised parameter can point to (in Step 1). This means that if the tag in such a node only has a single argument the arity raising will not really change the arity of the function, it will stay the same.

Note also that arity raising can never duplicate any work, due to a closure not being updated. In fact, an update operation operating on a function parameter will prevent arity raising from being applied to that parameter (it will not pass the condition in step 2 above).

Discussion

In combination with other GRIN transformations, the arity raising can sometimes create really amazing results, similar to the use of *deforestation* [Wad88], and even *listlessness* [Wad84] in some situations. In appendix A on page 301 we show a complete transformation example where exactly this happens. A typical "lazy computation", involving two functions (a *producer* and a *consumer*) communicating via a lazy list (and closures), is transformed into a totally strict computation that requires no intermediate list, and creates no closures (in fact, the resulting loop uses neither heap nor stack).

As mentioned earlier, this kind of arity raising is really just a special case of the possibilities offered by our *generalised unboxing* (see section 4.3.3 on

page 118). In the example above the arity raising we did was equivalent to doing both a *load* and an *untag* (to use our terms from the unboxing section). However, it would also be possible to let the arity raising only do the loading part, if the GRIN tag is not unique. This would effectively transform a function argument that is a pointer to a node in the heap into a function argument that is a node value (including a tag). That node value could then be split into several arguments. In the above example (see figure 4.33) this would result in an additional argument for the tag value, that could be inspected by a normal GRIN case expression (in the original example the tag were always known to be Fbar, so it needed no inspection).

4.3.14 Common sub-expression elimination

Common sub-expression elimination (CSE) is a well known conventional optimisation [ASU86, section 10.2], and is very easy to implement for GRIN. A popular method to do CSE is to first do *value numbering* [AWZ88, RWZ88], and then in some way propagate these value-numbers, for all *available* values. In the presence of cyclic flow graphs (loops) this process usually involves iteration. In GRIN it is much simpler. Since all GRIN flow graphs are DAGs (see section 2.3.2 on page 30), we can propagate available expressions using a single forward traversal of the code.

Common sub-expressions in GRIN

Also, the possible common sub-expressions themselves are very simple and uniform in GRIN. All "computations" in GRIN (e.g., arithmetic calculations) are done as sequences of GRIN primitive operations. The "splitting" of complicated expressions into sequences of primitives is in fact done already by the compiler front-end. This means that all GRIN expressions have the form:

$$\langle op \rangle \; ; \; \lambda v \rightarrow \langle m \rangle$$

where "$\langle op \rangle$" normally is a "simple" operation (it could also be a case expression or a function call, see the syntax in figure 2.1 on page 28). If we assume that "$\langle op \rangle$" is a potential sub-expression it is trivial to do CSE. We can simply remember that the result of "$\langle op \rangle$" is already available in the variable "v", if "$\langle op \rangle$" is ever performed again inside "$\langle m \rangle$". To do this we add the sub-expression to an environment (which is used to hold all available expressions). I.e., if the previous environment was "*env*" it will be extended as follows:

$$env \; [\, \langle op \rangle \mapsto \mathsf{unit}\; v \,]$$

This means that for all GRIN operations that we find (like "$\langle op \rangle$" above) we can simply do a lookup in the environment to see if the operation has already been found. If it has, we reuse the old value instead of redoing the computation.

A value in a variable is "reused" by inserting a GRIN unit operation. As can be seen above we actually put the unit operation already in the environment, rather than just the variable. This is somewhat more general, as it makes it possible to add other things than just simple variables.

To continue the example, if we find "$\langle op \rangle$; $\lambda v \to \langle m \rangle$" and instead assume that "$\langle op \rangle$" had already been available in the variable "u", then the CSE would result in:

$$\mathsf{unit}\ u\ ;\ \lambda v \to \langle m \rangle$$

The extra unit operation will be eliminated later by the copy propagation (see section 4.3.2 on page 113).

The "computations" that are interesting for CSE ("$\langle op \rangle$" above), can be of the following kinds:

- function calls,
- primitive operations, e.g., *intAdd*,
- unit operations,
- memory fetch operations,
- memory store operations.

Since GRIN is a functional language, side-effects is not a big problem (although see the note about updates below). All the above can just be added "as is" to the set of available expressions, with one small exception.

Memory operations

For store operations we will also add a corresponding fetch to the set of available expressions. Consider the following example:

$$\mathsf{store}\ (\mathsf{CCons}\ x\ xs)\ ;\ \lambda p \to \langle m \rangle$$

Here, the store operation writes a node value to the heap. The store itself will of course be available inside "$\langle m \rangle$", but since we know exactly what was written we can do even better. A fetch operation that loads the same node back from the heap (i.e., using the variable "p") can also be made available. To accommodate this, the actual environment extension for the above would be:

$$env\ [\,\mathsf{store}\ (\mathsf{CCons}\ x\ xs) \mapsto \mathsf{unit}\ p,\ \mathsf{fetch}\ p \mapsto \mathsf{unit}\ (\mathsf{CCons}\ x\ xs)\,]$$

Then, assume that we actually find such a fetch inside "$\langle m \rangle$":

$$\mathsf{fetch}\ p\ ;\ \lambda u \to \langle m_1 \rangle$$

Now, we know that the node returned by that fetch will be identical to what we earlier wrote, so we can transform to:

$$\mathsf{unit}\ (\mathsf{CCons}\ x\ xs)\ ;\ \lambda\, u \to \langle m_1 \rangle$$

Note that this is always safe because no variables can have been redefined, GRIN variables are single assignment (see section 2.3.7 on page 34). There is only one thing that we need to be a bit careful with, the node in the heap must not have been updated inbetween the store and the fetch. This means that when the CSE transformation finds an update operation, it must also update the contents of its environment, to reflect the new node value written. This is easy to do, because the same variable name is used in both the store and the fetch operation ("p" above). Note also that we need only bother with this if the stored node value represents a closure (if a whnf is ever updated it must be with an identical value). Function calls that risk updating nodes in the environment must be treated in a similar way, resulting in that some fetch operations may not be available after a call.

The above situation, a store followed by a fetch, may seem a bit exotic, but is actually quite a common situation. It can appear after *inlining* (see section 4.3.10 on page 135). Typically, an "outgoing" argument is first stored by the caller, and then fetched by the callee. When the call is inlined these two operations appear inside the same GRIN expression, i.e., visible to the CSE.

A larger example of CSE is shown in figure 4.34 on the next page (continued in figure 4.35). To fully eliminate all common sub-expressions it is necessary to run the CSE twice, with an intervening *copy propagation* (to eliminate all the inserted unit operations). This is also shown in the figures.

Tag information

While doing the CSE transformation it is important to know the values of as many tag variables as possible. To do this, we use the GRIN *tag information* (see section 4.4.4 on page 160). In addition to that, we also track tag values when transforming the alternatives of case expressions, similarly to in the constant propagation. This allows us to catch situations such as the one in figure 4.36 on page 153. The "unit CCons" is unnecessary since the tag is already available in the "t'" variable.

Limiting CSE

It can be discussed if it really is profitable to eliminate all possible common sub-expressions, even in situations where it would be very cheap to recompute. E.g., the "unit CCons" in figure 4.36 is relatively cheap to recompute, it is just a matter of putting a constant value in a register. It might even seem strange to keep such a value in a variable for a long time, and thus tying up a register. However, we believe it to be important to utilise all "potential register usage"

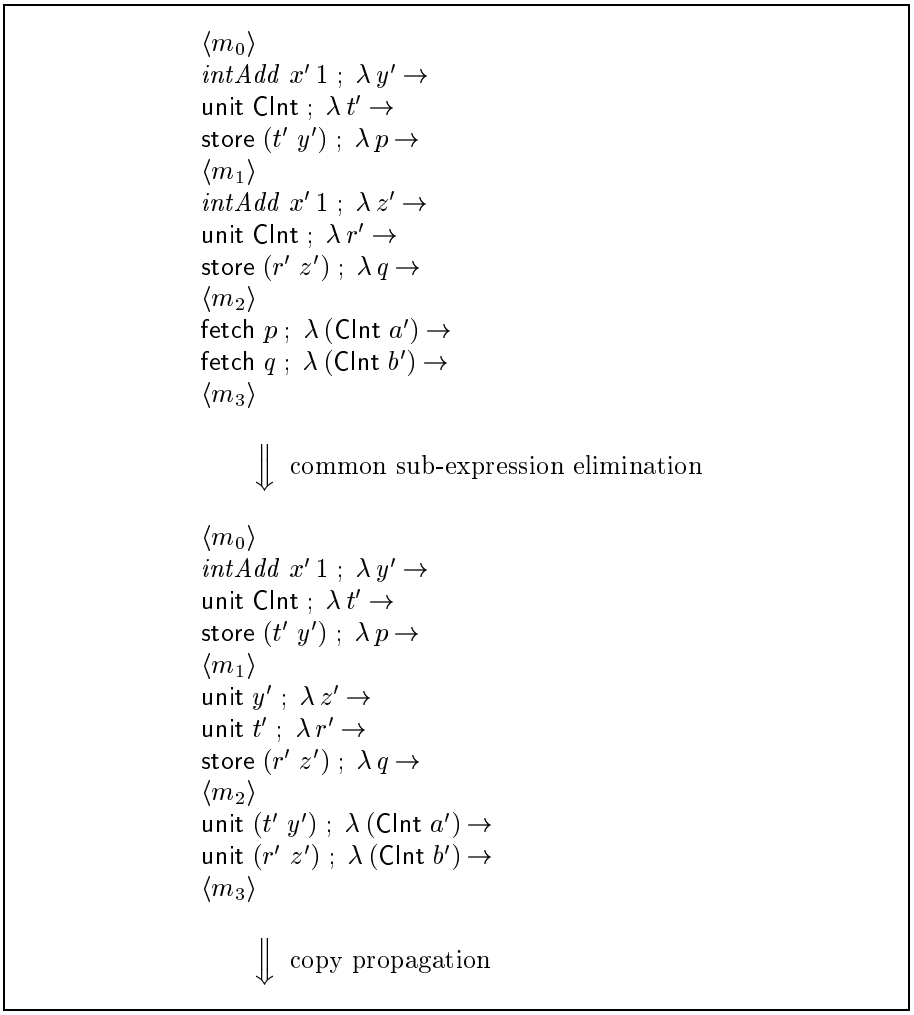

Figure 4.34: Common sub-expression elimination (*part 1*).

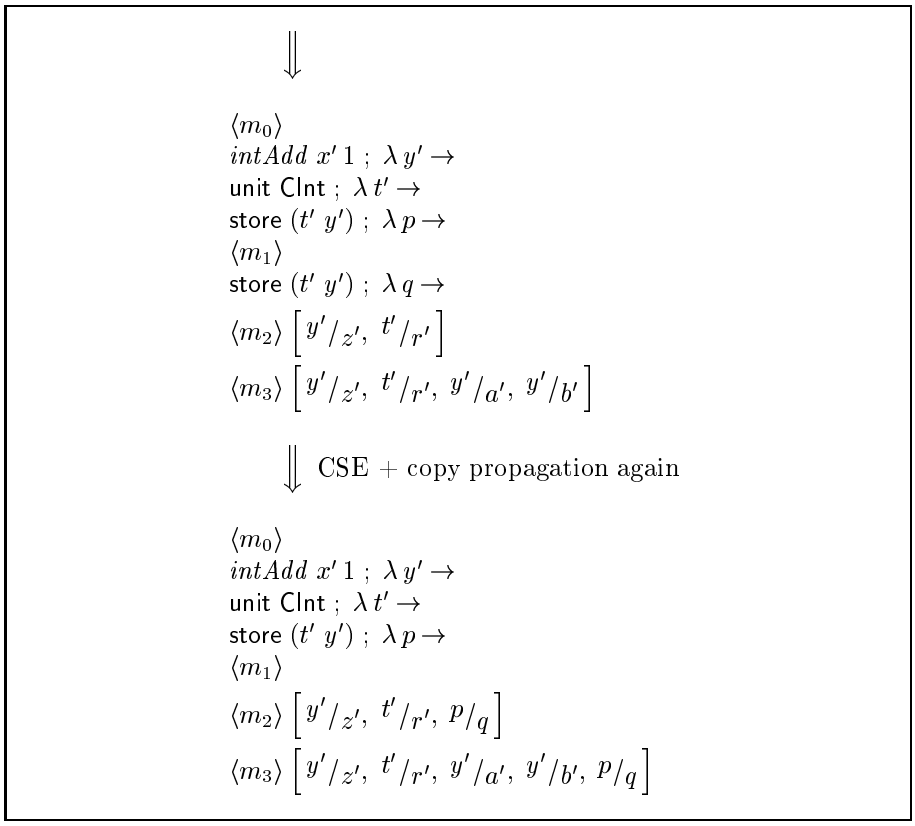

Figure 4.35: Common sub-expression elimination (*part 2*).

at the intermediate code level. We can then leave it to the register allocator (see chapter 6) to decide what should be held in registers. After all, it "knows more" about the *register pressure* at different points in the code than we do at the GRIN level. The register allocator can also be made aware that some values are cheaper to "recompute" than to keep in memory and take advantage of that for values such as the tag above, a technique called *rematerialisation* [BCT92]. Furthermore, the above example of loading a constant value into a register might actually require several machine instructions on some modern RISC architectures (e.g., the Alpha), because the instruction length is too small to encode large constants in a single instruction. So keeping the constant in a register may be quite beneficial after all. See also the discussion in section 4.2.7 on page 106.

Chitil reported disappointing CSE results for *Core Haskell* in [Chi98], with not very many common sub-expressions available for optimisation. In GRIN more opportunities ought to arise since it is a more low level language, which

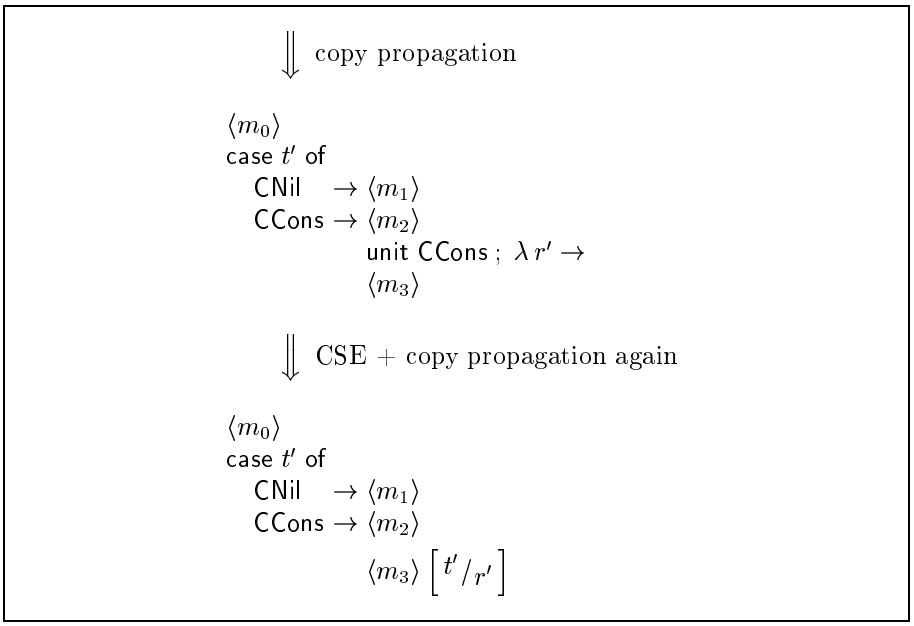

Figure 4.36: An example of CSE: case tags.

exposes more details than Core Haskell.

4.3.15 Dead code elimination

This section will describe two variants of *dead code elimination*:

- dead procedure elimination,
- dead variable elimination.

Both these variants are quite simple to implement in the GRIN framework. As in conventional compilers, the dead code elimination is an important complement to the more "ordinary" optimisations.

Dead procedure elimination

In the GRIN back-end one of the basic assumptions is that the entire program is available at compile time. This makes it easy to check for functions that can never be used. It is basically a question of *reachability* in the function call graph. We start with the *main* function and create a closure of reachable functions, following the function call arcs in the graph. This transformation will only be run after the *eval* inlining (see section 4.2.1 on page 84), when all

function calls in the program are direct calls. So, by following function calls from *main* we will reach all functions that can ever be used. Note that it is very important to do this *after* the *eval* inlining, otherwise we will not be bale to remove any procedures at all, since the *eval* procedure itself contains calls to all other procedures.

To be complete we must also treat "calls" to CAFs (*Constant Applicative Forms*, globally defined constants [PJ87]) as ordinary calls, i.e., we must look "inside" CAFs. Alternatively, we could include all CAFs in the starting set together with *main* (to be on the safe side), but that would risk including CAFs that are not used. This could actually prove to be a real problem in practice, for a language like *Haskell* [Hud92] where the standard implementation of *overloading* [WB89, Aug93] may produce many unused *dictionaries* that are CAFs.

Apart from ordinary functions we will also try to delete unused tags, F-tags and P-tags (see section 2.3.3 on page 30). Recall that for each function (say *foo*) the GRIN back-end will create an F-tag (Ffoo), if the function is ever used in a non-strict context, and a set of P-tags (Pfoo_1, Pfoo_2, etc.), if the function is ever partially applied. A tag that does not appear anywhere in the set of used functions is dead and can safely be deleted. Of course, "deleting" a tag that does not appear in the code may seem a bit strange, but is actually quite useful. For practical purposes, being able to delete tags and thereby reducing the size of compiler symbol tables etc. can be a big win. And also, for optimisation purposes, it is an advantage to know exactly what tags are used by the program (see the discussion in section 4.4.3 on page 159).

Interestingly, the elimination of a particular function and its corresponding tags are largely unrelated. We can have situations where a function is dead but not its F-tag, and vice versa. If a function is only used in strict contexts the F-tag will never be used and can be removed. On the other hand, we can also have a situation where a function originally appears in lazy contexts, i.e., the F-tag is used in store operations and in the *eval* case expression (see figure 4.2 on page 85), and the function itself is called inside *eval*. If that is the case and the function is non-recursive, and we *inline* all calls to the function (see section 4.3.10 on page 135), then we will get a situation where the F-tag is still used but the function is dead. I.e., we still create closures of the function, but instead of calling the function when a closure is forced we simply execute the inlined function body.

Dead variable elimination

The second form of dead code elimination in GRIN is to delete bindings where the bound variable is never used. This is the transformation that is normally called *dead code elimination* [ASU86, section 10.2]. However, to distinguish this from the other variants of dead code elimination that we describe, we call this *dead variable elimination*.

Recall the basic structure of GRIN expressions:

$$\langle op \rangle \ ; \ \lambda\, v \rightarrow \langle m \rangle$$

Here, if the variable "v" is not used inside "$\langle m \rangle$" the binding is useless and can be removed. There is only one restriction to this simple rule. If "$\langle op \rangle$" is a side-effecting operation, like update, or a side-effecting primitive then we must not delete it. However, this situation is always easily detected because "v" in such cases will be the empty value, "()".

In general, since lambda patterns can bind more than one variable (in explicit node patterns), we will use *free variables* to check each binding. For the above, if the intersection of "$\mathit{freeVars}(v)$" and "$\mathit{freeVars}(\langle m \rangle)$" is empty it means that none of the variables in the binding is used, so it is safe to remove the operation.

Note that it is important to implement the dead code elimination "bottom up" on GRIN expressions. Otherwise we would not be able to handle things like the following (assume that neither "p" nor "q" are used inside "$\langle m \rangle$"):

$$\begin{array}{l} \mathsf{store}\ (\mathsf{CNil})\ ; \ \lambda\, p \rightarrow \\ \mathsf{store}\ (x'\ p)\ ; \ \lambda\, q \rightarrow \\ \langle m \rangle \end{array}$$

If we do the removal "top down" is is not possible to delete the first store operation since "p" is used in "the rest" of the code. Of course, this is not a big catastrophe in the GRIN back-end because all optimisations are repeated (see figure 4.1 on page 80), but if we can do more in a single pass by going "bottom up" that seems like a better choice.

The dead binding elimination is implemented as a single pass over the GRIN expression tree, at the same time annotating expressions with free variables and removing dead bindings.

As a special case, we will also consider constant patterns as dead bindings. E.g., the following:

$$\mathsf{fetch}\ p\ ; \ \lambda\, (\mathsf{CNil}) \rightarrow \langle m \rangle$$

Constant patterns can often arise as result of transformations that do *substitution*, replacing tag variables with constant tags, e.g., the copy propagation (see section 4.3.2 on page 113).

4.3.16 Dead parameter elimination

The third GRIN variant of dead code elimination (see the previous section) is to drop unused parameters from functions. In the same way as for the dead procedure elimination, this is possible in GRIN because the entire program is available at compile time. A good example of how dead parameters can arise is shown in figure 4.29 on page 137, where the heap points-to analysis has made the first argument unnecessary.

Implementing dead parameter elimination is a simple two step process. First we check the *free variables* of all function bodies, and remove parameters that are not in the free variables set of the function body. Then, we go through the program and remove arguments from all calls to changed functions. We also remove the corresponding arguments from all applications of F-tags and P-tags for changed functions. This is not strictly necessary, but it is very simple to do, and there is not really any reason to keep unnecessary arguments in the tags. In fact, we may save a lot of space (and time) by removing them. It may also enable further possibilities for the "normal" dead code elimination (removing dead variable bindings, see the previous section).

4.4 Miscellaneous

In this section we will describe a few transformations and other things that do not really fit into either the *simplifying* or the *optimising* transformation categories, but still need to be mentioned to complete the picture of the GRIN transformation process. E.g., a pass to collect information that is used later by some transformation.

4.4.1 Bind normalisation

The *bind normalisation* transformation may seem pointless at first, since it neither simplifies nor optimises the code, it just "changes" it a little. However, the transformation is actually extremely useful when implementing many of the other GRIN program transformations. It is used to transform GRIN expressions into a kind of *normal form*, so that it becomes easier to perform *pattern matching* when implementing transformations that search for a certain "GRIN pattern" before they can be applied.

Consider the following, the basic structure of all GRIN expressions:

$$\langle op \rangle \; ; \; \lambda v \rightarrow \langle m \rangle$$

Here, "$\langle op \rangle$" is normally a monad operation or a function call, i.e., a GRIN expression can be seen as a sequence of "simple operations" ("$\langle op \rangle$" could also be a case expression, see the syntax in figure 2.1 on page 28). However, an interesting thing happens when "$\langle op \rangle$" is initially a function call that later is inlined. If that happens, the resulting GRIN expression will get a "non-flat" structure. This is not a problem in the GRIN syntax and semantics, but it makes implementing GRIN program transformations more difficult. In a typical transformation we "match" an expression against a "code pattern" to see if the transformation can be applied. Such patterns often involve sequences of operations, which means that if we allow non-flat structures there might be several expressions, with different structure, that ought to match the same pattern. This makes implementing the pattern matching much more difficult.

Basically, what we need is a *normal form* for GRIN expressions. Fortunately, it turns out that it is enough if we always keep the GRIN expression tree "right skewed". I.e., we do not allow a bind operator inside "$\langle op \rangle$" above, unless "$\langle op \rangle$" is a case expression. An example of a right skewed syntax tree was shown in figure 2.2 on page 30. Whenever a transformation has resulted in code which is not on normal form, we can restore it with a simple program transformation, which makes the syntax tree right skewed again.

This *bind normalisation* transformation is exactly the same as the *monad associativity law*, that all monads must obey [Wad92]. Written in GRIN, the associativity law would read:

$$(\langle m_0 \rangle \; ; \; \lambda a \rightarrow \langle m_1 \rangle) \; ; \; \lambda b \rightarrow \langle m_2 \rangle$$

$$\equiv$$

$$\langle m_0 \rangle \; ; \; (\lambda a \rightarrow \langle m_1 \rangle \; ; \; \lambda b \rightarrow \langle m_2 \rangle)$$

By repeated applications of this law (going from the top and down) we can make sure that we get a totally right skewed GRIN expression.

After all transformations that "insert new code" (or really new bind operators), like inlining, the bind normalisation must be run to "fix" the expression tree.

4.4.2 Argument reordering

The current GRIN implementation adopts a *node layout* (see section 5.2.3 on page 178) where unboxed arguments (non-pointers) must be stored "before" boxed arguments (pointers) whenever a node value is written to the heap. The reason for this is that it makes it simpler to implement the garbage collector (see chapter 8). This means that at some point during the compilation some form of *argument reordering* must be done. We have chosen to implement this as a GRIN program transformation, instead of as an implicit "mapping" in the RISC code generator, because we want the code generator to be as simple as possible. I.e., we will once during the compilation run a transformation that changes the order of arguments for all tags that have unboxed arguments. Note that this can involve all our different kinds of tags, F-tags, P-tags and C-tags (see section 2.3.3 on page 30).

The transformation is very simple, and is done in two steps. In the first step we identify all tags that have unboxed arguments (unless they already are "before" all boxed arguments). As an example, consider a function *foo* with three arguments, and assume that only the second is unboxed. This means that both the tags Ffoo and Pfoo_1 (representing a partial application with one missing argument) have one unboxed argument, but not the tags Pfoo_2 and Pfoo_3. The first step of the transformation would be to create new versions of

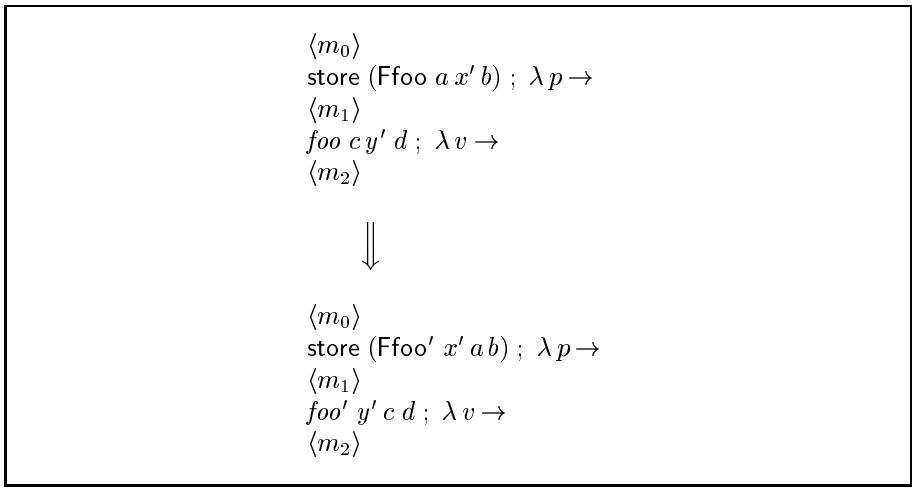

Figure 4.37: Argument reordering.

the tags that need to be changed:

$$\begin{array}{lcl} \mathsf{Ffoo} & \mapsto & \mathsf{Ffoo}' \\ \mathsf{Pfoo}_1 & \mapsto & \mathsf{Pfoo}'_1 \end{array}$$

The two new tags have the unboxed argument first.

When doing this we also create a new version of the function (*foo*), an identical clone except that its argument order is changed in the same way as for the F-tag. This is not strictly necessary, but we still do it because it is useful to keep the correspondence in argument positions between a function and its tags. For each function and tag that is changed, we remember a *permutation* that tells how the arguments were changed.

In the second step of the transformation we go through the entire program once, and for each occurrence of a tag (or a function) that was changed in the first step, we apply the corresponding permutation and replace the application with the new version of the tag (or function). An example of this, that transforms both the building of a closure and a call to the *foo* function, is shown in figure 4.37.

After the argument reordering transformation we run the *dead procedure elimination* (see section 4.3.15 on page 153). It will remove all the "original" functions and tags (*foo* and Ffoo above) since they are now unused (not reachable from *main*).

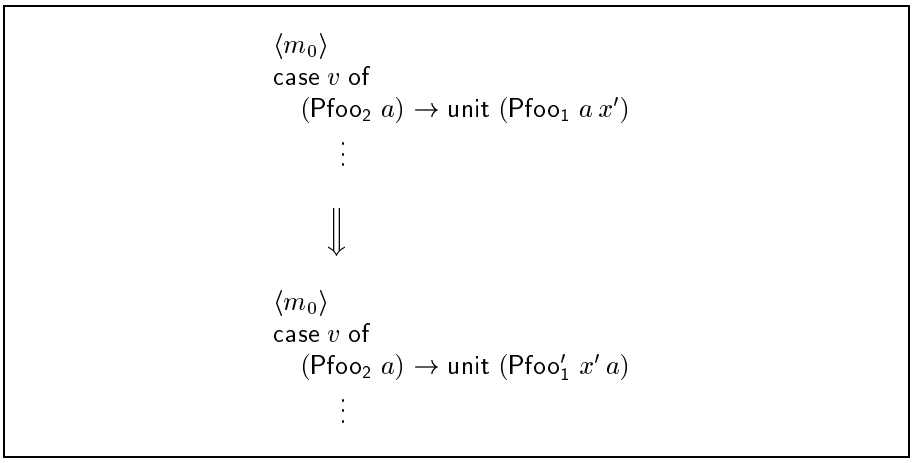

Figure 4.38: Another example of argument reordering, part of *apply*.

Partial applications

Note that argument reordering using the above method is not in any way complicated by partial applications (which it would probably have been if we were to do a corresponding transformation on the "functional" level). The reason is that in GRIN there are not really any partial applications, all tags are always fully saturated (or kept as separate tag values). What is normally called a partial application in a functional language is in GRIN represented as a fully saturated application of a P-tag (see section 2.5.6 on page 54). As an example, consider the special GRIN *apply* procedure (see figure 2.12 on page 55). Part of an *apply* procedure for a program involving partial applications of the *foo* function discussed above is shown in figure 4.38. Note that the Pfoo$_1$ tag is changed by the argument reordering transformation but that the Pfoo$_2$ tag is not.

4.4.3 Tag number assignment

In the current GRIN implementation (see section 5.2.3 on page 178) tags are implemented as small integers. Since GRIN code is very nice to work with (from a compiler implementors point of view), we have chosen to assign the integer tag numbers already at the GRIN level (rather than in the RISC code generator). Recall that all GRIN tags must be unique, i.e., distinguishable from all other tags in a program. Actual tag numbers will be assigned only to tags that remain in the final GRIN code, after all program transformations. Hopefully, many of the initially constructed tags (for function closures and partial applications) have been removed by the GRIN optimisations (e.g., the *dead code elimination*, see section 4.3.15 on page 153).

Currently, we assign a number to each tag according to a simple enumeration of all tags that remain in the program, in no particular order. However, given that the entire program is available we could actually do much better than this. We could in principle examine all case expressions, to see what tags they include in their alternatives, and then use that information to "optimise" the tag number assignment. Large case expressions are usually implemented using either *comparison trees* or *jump tables*, or using combinations of the two [Sal81], which means that the actual tag numbers are quite important. For a jump table implementation we need to create a mapping from each tag number to an index in the jump table. This index function should fulfil the following:

- it must be cheap to compute,
- its range must not be too *sparse*, since this will make the table unnecessarily large.

For very sparse ranges, the case expression can be split into a number of intervals, and then implemented as a combination of comparison trees and jump tables [HM82, Spu94].

It certainly seems possible that the tag number assignment could be optimised in some way, to make the code generation work well. However, we have not yet explored this route any further.

4.4.4 Tag information

The GRIN *tag information* is no more than a big table containing information about the tag usage of the program. Any compiler will use a number of "tables" (sometimes called symbol tables) to keep different kinds of information needed for the compilation, and this is also true for the GRIN back-end. However, we feel that the GRIN tag information table deserves to be described here, for two reasons. First, it is not very standard, and second, the information in the table is used by (and is very important to) many of the GRIN program transformations.

A typical example of a transformation where the tag information table is used, see the *copy propagation* (in section 4.3.2 on page 113). When a "copy" is removed it can sometimes be very important to remember an explicit GRIN tag that was part of the copy, e.g., we might need to record that a certain variable contains a certain known tag although that is no longer directly visible in the GRIN code. That kind of information is exactly what we call tag information.

The information in the tag information table is a mapping from GRIN variable names to a set of GRIN tags (see section 2.3.3 on page 30). If a variable is present in the table the set of tags describes the possible contents of that variable during any execution of the program. This set is always *safe*, i.e., if a variable is present in the table it will never be assigned a tag value that is not in the set given by the table. In particular, this means that if some variable has a singleton tag set then that tag is the only one ever assigned to that variable

(the variable is a known constant). On the other hand, if the tag information for a variable is a set containing more than one tag, it means that the variable is inspected by a case expression somewhere in the program, a case expression that contains exactly the tags found in the tag information table. This situation is the only one where the tag information can yield a set with more than one member. Remember that GRIN is single assignment (see section 2.3.7 on page 34), so a variable can never be "assigned" to in more than one place in the code.

Note that not all tag variables are included in the tag information table. To include a variable we must "know" exactly what tags it may have as values. If we only know some of the tags, but not necessarily all, for a certain variable, that variable can not be included in the table. This property is very important, without it we would never be able to "trust" the information in the table.

Tag information is collected and maintained as a "side effect" during many of the GRIN transformations (typically transformations that can remove GRIN tags that appear in the code, like the *copy propagation*).

4.4.5 Structure of the final GRIN code

The final GRIN code, after all transformations, is very simple compared to the initial GRIN code. There are no calls to the general *eval* and *apply* procedures. All variables containing entire node values have been eliminated, only pointers and basic values remain. All GRIN operations are very simple, e.g., all fetch operations use *offsets* (see section 4.2.7 on page 102) and load only simple values (not entire node values). All operands to GRIN operations, e.g., to the store operation, are variables ("registers"), due to the register introduction transformation (see section 4.2.9 on page 110).

GRIN case expressions are also simple compared to in the initial code, they will scrutinise only basic values (which include GRIN tags), and they bind no variables. The alternatives of a case expression are not required to be *exhaustive*, but a case expression is not allowed to fail to match at runtime. Note that not even a programmer bug should be able to cause a failed match at runtime. This can be guaranteed since we expect the compiler to fill in special "no match" alternatives producing an error message. These (and other impossible alternatives) can hopefully later be removed by the compiler if it can somehow prove that they can never happen for a particular program (see the *sparse case optimisation*, in section 4.3.6 on page 127).

The control flow inside each function will follow a directed acyclic graph, as in the initial GRIN code (see the discussion in section 2.3.2 on page 30). The program *call graph* however will be completely different compared to the initial GRIN code, due to the *eval* and *apply inlining*. There are also other transformations that affect the call graph of the program, e.g., the *late inlining* of ordinary functions.

An example of how GRIN code gradually becomes simpler and simpler can be seen in the transformation example in appendix A on page 301.

The intention with all of the above is that the actual machine code generation should become rather simple, with each GRIN operation corresponding to a single, or a few, machine instructions. This is described in chapter 5.

4.5 Related work

Previous work related to the GRIN language were discussed in section 2.6 on page 58. Some work related to specific transformations have already been mentioned in the descriptions of the different transformations, so we will only mention a few additional things here.

The idea of "compilation by transformation" is not new, it has been used in many compilers before, in particular compilers for functional languages. See for example [KH89, App92]. The idea of using a very large number of each very simple source-to-source transformations that is repeatedly applied is similar to what is used by the *simplifier* [PJ96, San95, PJ96] in the Glasgow Haskell Compiler (ghc). However, the nature of most of our GRIN transformations are quite different compared to those used by ghc, although some are similar. The main reason for these differences is that the GRIN language is a low level code compared to the Core language used by ghc.

Part III

RISC
the low level code

Chapter 5

RISC code generation

This chapter will describe RISC, the low level code that is used by the "latter half" of the GRIN back-end and show how GRIN can be translated into RISC. RISC is in essence a machine code (or really an assembler code) for a hypothetical *Reduced Instruction Set Computer* (RISC) processor. Our RISC code is very simple and the RISC code generation is mostly very straightforward. The organisation of the RISC part of the back-end can be seen in figure 7.1 on page 248. Compare this to figure 1.3 on page 17 that shows the organisation of the entire compiler.

In order to explain the RISC code generation we also need to show parts of the runtime system, e.g., how the stack and the heap are handled, how GRIN data structures are mapped onto memory, etc. The RISC code we use is actually quite close to the SPARC architecture, so we will start by giving an overview of that.

5.1 A hypothetical RISC machine

Our RISC code is closely modelled after the *SPARC* architecture [SPA92]. This means that we assume a RISC architecture, or as it is also known, a *load-store architecture*. This should be seen in contrast to the other main direction in computer architecture, CISC machines (*Complex Instruction Set Computer*). The properties that we are interested in, and which are normally associated with a RISC style processor, are:

- a relatively large register set (typically 32 or more registers) of (mostly) general purpose registers,
- simple and uniform instructions,
- most instructions can not take memory operands, only *load* and *store* can reference memory.

Note that the word "Reduced" in RISC refers mainly to the word "Instruction", not to the words "Instruction Set". In fact, many RISC processors have very large instruction sets, but with each instruction being very simple (at least most of them).

The above properties have one consequence that we consider the most important for the RISC code that is used in the GRIN back-end: the large register set makes it possible to use quite aggressive optimisation methods (in particular register allocation). Indeed, for most RISC machines it is vitally important to make good use of registers, since this is normally much faster than to use memory.[1] As a minor detail it can also be mentioned that the code generation, or more specifically the *code selection*, is much simpler for a typical RISC processor than it is for a typical CISC. This makes our code generator (from GRIN to RISC) quite simple, which is nice because code selection has not been a primary research target in the GRIN back-end project.

A detailed description of the actual RISC instruction set is given in section 5.1.3 on page 171.

5.1.1 RISC and machine dependence

The fact that we have chosen to generate code for a "SPARC like" architecture is not very significant. The RISC code should be seen simply as an example of a machine code for some RISC architecture, and the code is used as a "test bed" for the optimisations presented in the RISC part of this thesis. The RISC code should *not* be seen as a general approach to making a *machine independent* compiler back-end! That issue could easily take up a thesis on its own. If machine independence is desired, which it of course is for "real" compilers, some other technique should be used to attack that problem, e.g., the ML-RISC framework [GGR94, Geo96]. Hopefully though, most of the methods described in this thesis are general enough to be applicable even in a different low-level back-end framework.

5.1.2 The SPARC architecture

Since our RISC code is so closely modelled after the SPARC we will first explain some of the main properties of the SPARC architecture. In particular, we will describe the SPARC *register windows* and explain why that feature is *not* used in our RISC code. Readers already familiar with the SPARC can safely skip this section.

The SPARC architecture is really a family of architectures. There are a number of different "versions" of the architecture specification, and a number of different implementations of the architecture. Our RISC code is modelled

[1] Note that even if all memory accesses could be made without delay, a kind of "perfect cache", memory based code would still be slower than register based code because most instructions can not take memory operands (load and store instructions would still be needed).

after SPARC V8 [SPA92]. This means that it is a uniform 32 bit architecture, the *word size*, the *instruction size*, and the *register size* are all 32 bits. The instruction set consists mostly of standard RISC style instructions, with only load and store instructions to reference memory. A *condition code register* is used to hold boolean results, i.e., the results of comparisons and some logical and arithmetic operations. For a more elaborate description of the basic features of the architecture the reader is referred to some SPARC textbook, e.g., [Pau94].

A few properties of the SPARC architecture deserve to be explained in a bit more detail here: the *register windows*, *delayed loads* and (user visible) *branch delay slots*.

Register windows

The SPARC architecture uses a windowed register set. This means that most of the (integer) registers are organised as a stack, where only a small *window* of all registers is "visible" at any given time. The visible window can be "moved" by the software using special *save* and *restore* instructions. This is done in such a way that two adjacent windows will always be overlapping, i.e., have some common registers. The idea is to provide hardware support for a simple kind of *register allocation* (and thus making it simpler for the compiler). In short: the register windows helps passing procedure arguments (and results) in registers, and also provide new registers for local variables to each called procedure.

This is achieved by a division of each register window into three parts, each of which consists of eight registers: *ins* (incoming arguments), *locals* (local variables), and *outs* (outgoing arguments). The *ins* and *outs* overlap between adjacent windows, as shown in figure 5.1 on the following page.

The intention is to switch to a "fresh" window at each procedure call. Outgoing arguments that are put in *outs* registers before a procedure call, by the *caller*, will after the call and a window switch (going right in figure 5.1), be available in the *ins* registers of the *callee*. In addition, the callee now has a new set of *locals* registers, to use for local variables and temporaries, and a new set of *outs* registers, to use for outgoing arguments. The *locals* of the caller are preserved in the previous window and are not available to the callee. When the procedure is about to return, it will switch back the register window one step, returning to the caller's registers.

Note that the "register switching" is fully controlled by software. Normally, a compiler will insert a *save* instruction at entry to each procedure, and a *restore* instruction at each return point of the procedure. However, the compiler does not have to insert *save* and *restore* instructions for all procedures. Typically, *leaf procedures* (that make no function calls) can often be compiled without switching to a new register window, unless the procedure has a very large register demand.

Two of the *ins* registers (and consequently also two of the *outs* registers, since they are the same), are meant to be used as a *stack pointer* and a *return address* register. This works quite well together with the windowed register

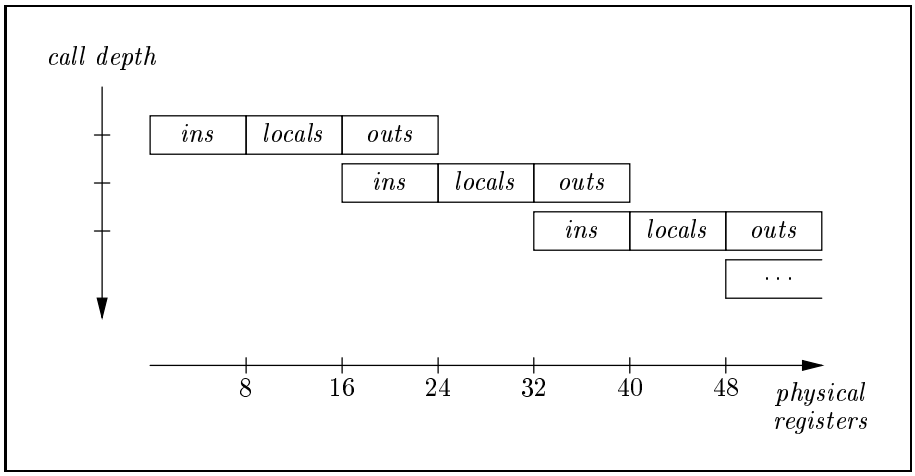

Figure 5.1: SPARC register windows.

set. When a window switch is done, just after a procedure call, the caller's stack pointer (in an *outs* register) will automatically become the callee's *frame pointer* (now in an *ins* register). A new stack pointer is set up for the callee (a stack frame size is always part of the *save* instruction).

A similar thing happens for the return address. The SPARC uses a *jump-and-link* style instruction to do procedure calls. I.e., when a procedure call is done the return address is saved in a register by the hardware. For SPARC this is one of the *outs* registers. After a window switch has been done, the return address will end up in an *ins* register of the callee, i.e., it is "safe" for further function calls by the callee and need not be saved to the stack at all!

Independently of the windowed register set, SPARC has eight *global* registers. These are always available.

The register windows may seem extremely nice and useful at first, but of course, such convenience does not come for free. The problem is that the actual number of windows (and physical registers) must be limited in an implementation of the SPARC architecture (a typical number is 8 windows, which means about 128 registers in total). The consequence of this is that if a window is "pushed" (a *save* instruction) and the end of the physical registers is already reached, the processor must in some way "fake" new registers. This is done by saving old windows to stack memory, thus freeing a couple of new windows. Analogously, if a window is "popped" (a *restore* instruction) the processor may have to reload registers from the stack to fill a window that was saved earlier. These two situations are called *window overflow* and *window underflow*, respectively. What happens is really that the processor generates a *trap* when this happens, and jumps to special routines in the operating system. The whole

process is in some sense "transparent" to the program, but it still has to pay the price in execution time of the saving and restoring of registers on the stack.

Note that it is not obvious "how many" windows should be saved when a window overflow trap occurs. It depends completely on if the window operations to follow will be "pushes" or "pops". I.e., it depends on the procedure call and return behaviour of the program (the nesting depth of procedure calls). Since this is very hard to predict (dynamically) in hardware, it has to choose something which works well most of the time.

The windowing works relatively well when the procedure call nesting depth has a "flat profile", i.e., there are not too many calls without intervening returns. If that is the case window overflows and underflows can largely be avoided. This is actually the main motivation for using register windows from the start, studies of "typical programs" showed that they often had such a behaviour (see [Sta88] for a good discussion of this).

A typical sub-optimal situation for register windows is when the call nesting is very deep (many calls without intervening returns), such as a deep recursive loop (not tail recursive). This will result in a number of window overflows, when all the recursive calls take place, followed by a number of underflows, when the recursion is unwinded. This is sub-optimal in two ways. First, there may be a large overhead associated with taking the traps into the operating system. Second, the window save and restore routines must save *all* the registers in a window, it has now way of telling which ones actually contain values that will be used later. Unfortunately, deep recursion is very common in functional languages, where recursion is used to do all "loops". This fact has caused many compiler implementors (for functional and other languages) to mostly avoid using the register windows, even if that means that a large part of the physical register set becomes unavailable. This is also the approach taken by the GRIN back-end (see more about this below). As a result, we will *not* use any *save* and *restore* instructions in the code generated from GRIN.

The SPARC is currently the only major general purpose processor to provide a windowed register set.

Delayed loads

The *delayed load* instruction is something that the SPARC shares with many other RISC architectures, and is a simple consequence of the fact that the memory subsystem often is slower than the processor. This means that the result of a load instruction will not be immediately available for use by the succeeding instruction. For the SPARC the delay is a single instruction, i.e., the instruction following a load should not use the result of the load. If it does, the processor *pipeline* will *stall* and execution be halted for one cycle, i.e., the code will slow down, but still produce a correct result. On some other architectures, e.g., the MIPS, we would instead get the wrong result if we tried to use the result of a load too early.

This means that for good performance the compiler should *schedule* instructions in such a way that a load instruction is not immediately followed by an instruction using the result of the load, i.e., at least one *independent* instruction should be put inbetween (this is called *instruction scheduling*, see section 7.8 on page 256).

Branch delay slots

The last special property of the SPARC that we will describe, *branch delay slots*, is similar to the previous. As for delayed loads (or load delay slots), branch delay slots exist for architectural reasons, but this time not so much because of the memory subsystem, but more because of *pipelining* and *instruction pre-fetch* reasons. In short: branch delay slots offer a simple way (for the hardware) to find an instruction to execute while the target of a conditional branch is still being calculated, i.e., when the processor is deciding whether to take the branch or not. Consider the following SPARC code:

```
    ⋮
bne   Label1
mov   1,%g2
    ⋮
```

Here, "*bne*" is a conditional branch instruction, and "*mov*" is a simple instruction used to load a constant value into a register (in this case a SPARC register called "*%g2*"). The move instruction is in the *branch delay slot*, i.e., the instruction immediately following a branch instruction. This means that the move instruction will be executed *before* the branch happens, and independently of whether the branch is taken or not. Again, the reason to do this is that it is easy for the hardware to execute the move instruction while we are still deciding if the branch is going to be taken or not, it avoids "bubbles" in the processor pipeline.

This technique has in some sense been made obsolete by more recent techniques such as *speculative execution* and *execution in parallel* of branch targets, but the SPARC still has this feature (visible to the user).

Note that the delay slot instruction must be chosen such that the branch instruction does not depend on its result. Also, a delay slot instruction should not be an instruction that itself would require a delay slot, otherwise unpredictable things might happen. The compiler is responsible for finding suitable instructions that can fill delay slots. If it can not find an independent instruction that can be moved to fill a delay slot, it must insert a "*nop*" (no operation) instruction.

The SPARC also has a possibility to *annull* a delay slot instruction, by appending a ", *a*" to the branch instruction name. This means that all effects of

the delay slot instruction will be annulled (cancelled) if the branch is not taken. Annulled branches can be used to improve the instruction scheduling.

In addition to all branch instructions, all other kinds of "jumps" (including procedure calls and returns) require a delay slot instruction.

5.1.3 RISC code

Given the above description of the SPARC architecture it is now quite simple to define the hypothetical RISC machine that we will use.

RISC registers

Since the RISC code is used during different kinds of optimisations we need a couple of different types of registers. In particular, RISC code is used before the register allocation phase, when "real" machine registers have not yet been assigned. The registers types in RISC are (see also figure 5.2):

- *virtual* registers, starting with the letter "*v*", are used before the register allocation,
- *real* registers, starting with the letter "*r*", are used (mostly) after the register allocation,
- *special* registers, starting with the letter "%", are reserved registers (stack pointer, return address register, etc.).

Note that these are all integer registers. RISC does not yet include floating point registers (these are not very important for the kinds of experiments that we have been mostly interested in).

The number of *virtual* registers is unlimited. Whenever the RISC code generator or a RISC optimisation (before the register allocation) needs new registers it can create new virtual registers. The idea is that the register allocator (see chapter 6) will replace all uses of virtual registers with real machine registers (possibly combined with the use of memory).

The *real* registers are intended to be exactly the machine registers that are available on the target architecture (except the ones that are reserved for special purposes). As the final step in the back-end, the RISC code will be "pretty printed" as SPARC assembler (see figure 7.1 on page 248). At this point the RISC "*r*" register names will be translated to real SPARC register names. The actual mapping between RISC real machine registers and the SPARC machine register names is not important (since they are all allocated by the register allocator), with one exception. The six RISC registers "r_8" up to "r_{13}" correspond to the first six *outs* registers of the SPARC. This is important to know since these registers are also used when doing *external calls* (see section 5.2.1 on page 175).

The *special* registers are named as to coincide with the SPARC names, if possible. These are mainly the stack pointer ("%*sp*") and the return address

Registers =		
$v_1 \mid v_2 \mid v_3 \mid \ldots$		virtual registers
$\mid r_1 \mid r_2 \mid r_3 \mid \ldots$		real machine registers
$\mid \%sp \mid \%o_7 \mid \%hp \mid \%hp_{gc} \mid \ldots$		special registers
Instructions =		
load	$[reg_1 + op], reg_2$	load word
$\mid$ *store*	$reg_1, [reg_2 + op]$	store word
$\mid$ *move*	op, reg	register move
$\mid$ *sethi*	$const_{22}, reg$	fill high bits
$\mid$ *or/and/xor/...*	reg_1, op, reg_2	logical instructions
$\mid$ *not*	reg_1, reg_2	inverse
$\mid$ *sll/srl/sra*	reg_1, op, reg_2	shift instructions
$\mid$ *add/sub/addcc/...*	reg_1, op, reg_2	arithmetic instructions
$\mid$ *smul/sdiv/wr*	reg_1, op, reg_2	multiplication and division
$\mid$ *cmp*	reg_1, op	comparison
$\mid$ *ba/blt/ble/beq/...*	*label*	branch instructions
$\mid$ *call/ecall*	*global*	procedure calls
$\mid$ *return*		return from procedure call
$\mid$ *jmpl*	$reg_1 + op, reg_2$	general jump
$\mid$ *save/restore*	$[reg_1, reg_2, \ldots]$	move to/from stack
$\mid$ *spill/reload*	$[(reg_1, reg_2), \ldots]$	move to/from stack

Note: The rightmost operand is always the destination.
op is either *reg* or $const_{13}$.

Figure 5.2: The RISC instruction set.

register, called "$\%o_7$" (the reason for this name is that the register is in fact one of the *outs* registers of the SPARC, see above). In addition, we create two special registers for the graph reduction: the heap pointer ("*%hp*") and a heap limit for garbage collection ("$\%hp_{gc}$"). Inherited from the SPARC are also the "$\%g_0$" register (always zero when read and discards all writes), and the "*%y*" register (used during multiplication and division).

RISC instruction set

Most of the instructions available in RISC are taken directly from the SPARC. The instructions are shown in figure 5.2. Similar instructions, that take the same kinds of operands, have been grouped together. The operand convention is that the *destination* is always written as the rightmost operand. An arbitrary

(integer) register is written "reg_i" and an N bit constant is written "$const_N$". The special pattern "*op*" denotes either a register or a short constant (13 bits).

Memory instructions. As in the SPARC, load and store instructions is the only means of accessing memory. The RISC "*load*" and "*store*" instruction operate on complete memory words (32 bits), and there is currently no instructions for smaller units of memory. This has been due to the very uniform memory layout decisions made in the GRIN back-end (see section 5.2.3 on page 178), but this might change in the future. Similarly to SPARC, load instructions are delayed a single cycle (see section 5.1.2 on page 169).

ALU instructions. Most of the arithmetic and logical instructions are hopefully self explanatory (see figure 5.2). As for SPARC, the operands of most instructions are either registers or small constants (often 13 bits). This means that in order to fill a register with a word sized constant, i.e., 32 bits, two instructions must be used. E.g., assuming that we need the value of the global symbol "*foo*" in register "r_3", we would do:

$$\begin{array}{ll} sethi & \%hi(foo), r_3 \\ or & r_3, \%lo(foo), r_3 \end{array}$$

The "%*hi*" and "%*lo*" are special assembler directives to get at the highest 22 and the lowest 10 bits of a 32 bit constant, respectively. The "*sethi*" instruction sets the highest 22 bits of a register and clears the lowest 10 bits.

Our RISC machine also includes instructions for multiplication and division, in the same way as SPARC V8 (earlier versions of the SPARC did not have these instructions in hardware). The "*wr*" instruction is a special SPARC instruction related to multiplication and division. E.g., it can be used to implement *reminder* functionality (see [Pau94]).

The comparison instruction ("*cmp*") and all instructions ending in "*cc*" set *condition codes* in the standard way. The condition codes are tested by conditional branching instructions. There is no easy way to translate a condition code into a "boolean value" in a general purpose register. If such a transfer is needed it has to be done using a conditional branch instruction. This is the same as in the SPARC (see also the discussion in section 4.2.6 on page 101).

Branch instructions. All branch instructions have a single delay slot, as in the SPARC (see section 5.1.2 on page 170). All branches except "*ba*" (*branch always*) use the condition code register to decide if the branch should be taken or not, in the standard way. All branches can also be *annulled* by appending ", *a*" to the branch instruction name. The semantics of annulled branches are the same as for the SPARC. If no "useful" instruction can be found to put in a

delay slot it is always possible to insert a "*nop*" instruction[2] that does nothing.

Procedure calls. Normal GRIN procedure calls, i.e., when both the caller and the callee are GRIN procedures, are done using the RISC "*call*" instruction. This instruction follows the standard SPARC linkage convention of putting the return address in register "$\%o_7$". This register is also what is used by the "*return*" instruction, to return from procedure calls. External procedures, i.e., procedures following the standard C calling convention for SPARC, can be called using the "*ecall*" instruction.[3] To help the compiler, all "*ecall*" instruction must be annotated with argument and result registers, and also a set of *clobbered* registers (registers that does not survive the call). Finally, a general "*jmpl*" instruction exists for calls to an arbitrary address (in a register, the other call instructions all jump to a label). For the "*jmpl*" instruction an arbitrary return address register can also be specified, e.g., "$\%g_0$" if no return address is needed. All call instructions and the return instruction have a single delay slot.

Special instructions. To simplify the implementation, we have included special instructions for saving and restoring registers in the current *stack frame*. In RISC, these are called "*save*" and "*restore*". They take as argument a list of registers, and will write (or load) the registers to (or from) "home locations" in the current stack frame. The exact offset in the stack frame need not be specified, it will be calculated by the implementation in such a way that two saved registers will never clobber each other (this is done at the end of the register allocation phase when all stack usage is known, see section 6.2.11 on page 230). Typically, the "*save*" and "*restore*" instructions are used to save registers around function calls that would otherwise risk clobber the registers. Note that these instructions are not the same as the SPARC *save* and *restore* instructions used to manipulate register windows.

During the register allocation phase, a slightly different variant of stack instructions are used, called "*spill*" and "*reload*". These instructions are similar to "*save*" and "*restore*" but take an additional register argument for each register to save, which specifies a location in the stack frame. This is used to allow several live ranges to spill to the same home location if they all originate from the same spilled live range (see section 6.2.9 on page 226).

Finally, RISC includes some "assembler directives" in the same way as SPARC assembly language: to manipulate constants ("*%hi*" and "*%lo*"), to put arbitrary values in the code ("*.word*", "*.ascii*", ...), and to switch between different executable segments ("*.text*", "*.data*", ...).

[2] There are numerous ways to *synthesise* a *no-operation* instruction, it does not have to be part of the "real" instruction set, e.g., as "*add* $\%g_0, 0, \%g_0$".

[3] Currently there is no way to "call back" into the GRIN program from external code.

5.2 Runtime system issues

Before showing the actual RISC code generation we need to describe some conventions for the mapping of the "graph reduction machinery" onto the hardware. One of the objectives of the GRIN back-end is to avoid a traditional "standalone" runtime system, i.e., we want as little "fixed" runtime system code as possible. Instead, we want as much code as possible to take part of the normal optimisation. The only exception to this in the current GRIN back-end is the garbage collector, which is written entirely in C. In this section we will describe the runtime model used by the generated RISC code, i.e., how it uses the heap, the stack, the registers, etc. This model provides only the minimal support needed to execute a single program, there is no support for advanced features such as *threads* or *exceptions*. All issues related to garbage collection will be described separately (see chapter 8).

5.2.1 Register usage

As noted earlier, the current RISC code does not make use of the SPARC register windows. Or more specifically, once inside GRIN code (i.e., RISC code produced from GRIN functions), we will use exactly *one* register window. In this *internal* RISC convention we do not follow the standard SPARC conventions of how registers are used. Instead, most of the 24 windowed SPARC registers (*ins*, *locals* and *outs*, see section 5.1.2 on page 167) are mapped directly onto the RISC real machine registers, and hence their uses will be determined by the register allocator. Only two of the SPARC windowed registers are special and not handled by the register allocator: the return address register ("$\%o_7$") and the stack pointer ("$\%sp$"). For simplicity we use the same register names for these two as in the standard SPARC convention.

The two special RISC registers used to handle the heap ("$\%hp$" and "$\%hp_{gc}$") are mapped to SPARC *globals* registers.

In the initial code that is generated by the RISC code generator not all RISC registers will be used (see the different types of RISC registers in figure 5.2 on page 172). The important thing to note is that the code generator mainly uses *virtual* registers, rather than real machine registers. The latter are introduced later, after the register allocation. The exception to this is a few of the real machine registers that must be used to do external procedure calls (see below). Of course, the code generator can also use the *special* RISC registers (all starting with a "%"), if they are needed to implement a certain GRIN operation.

External calls

In one particular situation it is important to strictly follow the standard SPARC convention, and that is when we do *external calls* (the RISC "*ecall*" instruction), i.e., calls from the GRIN program to external code (e.g., written in C and

compiled by a standard C compiler). In particular we must make sure that we pass outgoing arguments to external procedures in the *outs* registers. We must also avoid keeping any values in the *outs* registers that are needed after the call, since the registers may be clobbered by the callee (they are *caller-saves*). Fortunately, it is ok to keep values in the *ins* and the *locals* registers over an external call, these are *callee-saves* in the standard SPARC convention (because of the windowing). Normally, an external callee will do a register window switch to create new *ins* and *locals* registers. When doing an external call, the actual *save* and *restore* instructions for registers that need to be saved around the call will be inserted by the register allocator. This is handled in the same way as recursion inside the GRIN program (see section 6.2.3 on page 202).

5.2.2 Heap and stack layout

The graph reduction *heap* is handled in a straightforward way. The heap is considered to be a continuous amount of memory. A *heap pointer* that points to the first unused memory location is always kept in a special RISC register ("%*hp*"). The heap pointer is incremented to allocate more heap memory, in the usual way. It may also be necessary to include tests to check if the heap is full and need to be garbage collected. This is explained in detail later (see section 8.3.3 on page 269).

Functions that need to temporarily save registers (function arguments, local variables, and all other kinds of compiler temporaries, etc.) will allocate a *stack frame* (*activation record*) on the RISC stack, which on the SPARC is the same as the standard system stack. Note that the allocation of stack frames in SPARC code normally is done in conjunction with a register window switch, but since we do not use the register windows we will adjust the stack pointer manually ("%*sp*"). Because we are using the standard system stack we must obey the following SPARC conventions:

- the stack grows from high to low addresses,
- the stack pointer must always be double word aligned.

Apart from this, the actual stack frames created by the GRIN back-end will not follow the standard SPARC stack frame layout. Our format is very simple, the only reserved location in a stack frame is the top-most location (highest address) which is used to save the return address (if it needs to be saved). Other than that, the back-end (or really the register allocator) is free to save registers anywhere in a stack frame. The size of a frame is chosen depending on the maximal number of variables that may need to be saved (this is determined after the register allocation, see section 6.2.11 on page 230). The stack frame organisation is illustrated in figure 5.3 on the facing page.

An example of a RISC instruction to read the return address out of a stack

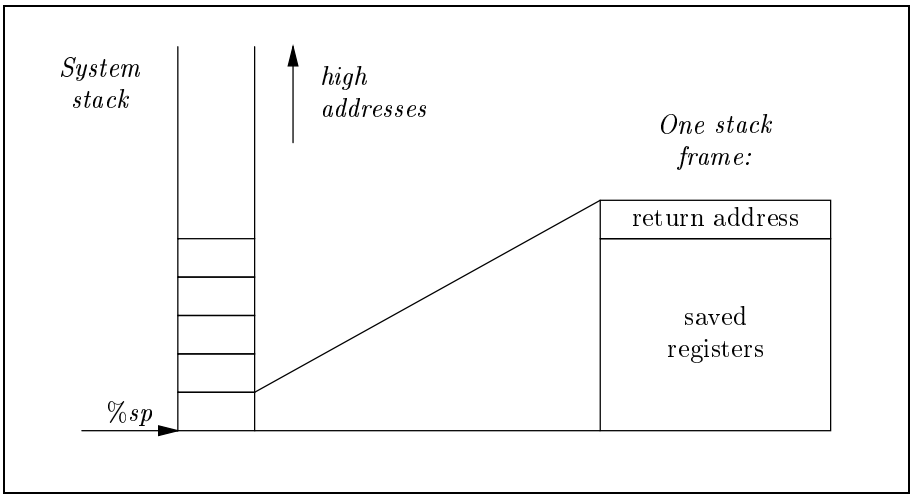

Figure 5.3: RISC stack frame layout.

frame that is 6 words large could be:

$$load \quad [\%sp + 5 * 4], v_1$$

Here, the "5" selects the top-most frame entry, and the "4" is the word size (addresses are always in bytes).

One thing to note is that a RISC stack frame has no special sections for *pointers* and *non-pointers*. Different kinds of values can be freely mixed, and it is up to the compiler to make sure that it "knows" what is stored in each entry. Such information is needed by the garbage collector, and it will be encoded in special *GC descriptors* that is stored elsewhere (not on the runtime stack). This is explained in the chapter about garbage collection (chapter 8). Another thing to note is that a stack frame can be allocated anywhere inside a procedure, and in general different "paths" through a procedure can have different stack usage.

Currently there is no way to pass arguments or return values on the stack. The default convention in the GRIN back-end is to pass all arguments of a function in registers, and to return node values from a function in registers. This needs to be changed though (there are no fundamental reasons preventing arguments on stack, it simply has not been implemented yet).

Stack reserved space

The above description of how the system stack is used is actually not completely true. There is one small detail that we will explain here, and then ignore for the rest of this thesis, because it is not very important, and it can be very confusing.

The problem is that according to the standard SPARC calling convention, the 24 words right above the stack pointer are reserved and must never be used by an application. This is really an operating system convention that can not be ignored (at least on Solaris/SPARC). The extra space is used by the operating system to save registers during certain kinds of kernel traps. In normal SPARC code, this extra space is supposed to be allocated in every stack frame. However, it seems like a waste do this for GRIN code, since we do not follow the SPARC stack frame layout anyway. To achieve this, the trick is to allocate the extra 24 words once and for all at the beginning of the execution and then *offset* all stack references by 24 words. I.e., the constant offset that is always present in an instruction that either reads or writes to the stack is incremented to "skip over" the 24 words. E.g., the example instruction above to load the return address from the stack would in real SPARC code become the following:

$$load \quad [\%sp + (5 + 24) * 4], v_1$$

But as we said above, from this point and on we will completely ignore this extra stack gap and just show stack references as in the first example (and as shown in figure 5.3, without any reserved space).

5.2.3 Node layout

Everything that is stored in the heap during the execution of a GRIN program will be in the form of GRIN *nodes*. Recall that GRIN nodes are used to represent both weak head normal forms and suspended computations. The notions of GRIN nodes and tags are discussed in section 2.3.3 on page 30.

The actual memory layout of node values that is used by the GRIN back-end is very simple. However, as in every compiler back-end all the low level details must "fit together", to make it possible to implement all the different operations that are needed. In an implementation of graph reduction, typical details that tend to interact closely with the memory layout of values are: how "eval" is done, how "update" is done, requirements of the garbage collector, etc.

Implementation requirements

The GRIN part of the back-end puts two requirements on the underlying implementation of nodes and tags:

- the representation of each GRIN tag must be *unique*,
- it must be "easy" to extract the tag and each argument from a node value.

The first requirement may seem very strict at first, but is not really very different from other implementations. Memory addresses (either of code or data) are often used to represent function closures, and sometimes also to represent all "types" of data. This means that the "uniqueness" is taken care of at program

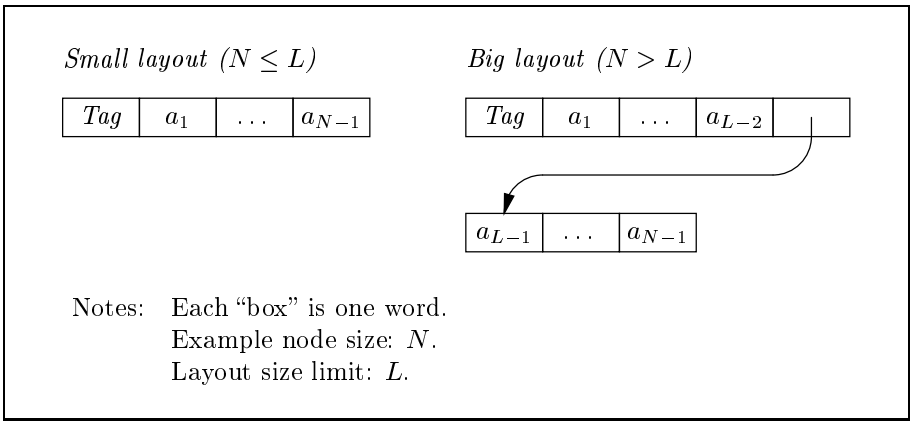

Figure 5.4: Nodes in memory: *small* vs. *big* layout.

link time. In the GRIN back-end we move this step into the compiler. However, one issue related to the implementation of case expressions still exists (see the discussion in section 5.3.2 on page 189).

The second requirement above is intended to make it efficient to implement GRIN expressions of the kind "fetch p [2]" (load the second argument from the node value pointed to by "p"). This means that it would probably not be a good idea to have an implementation where a function call is required to deliver the different parts of a node value (like in the STG machine).

Implementation

All GRIN tags are represented by program-wide unique integers. We simply enumerate all tags that appear in the program and assign a *tag number* (see section 4.4.3 on page 159). This is easy to do in the GRIN back-end since we rely on having the entire program available at compile time.

The node layout used by the GRIN back-end has two variants: *small* node layout and *big* node layout (see figure 5.4). The *small* layout is the simplest possible mapping of GRIN nodes onto memory, with one memory *word* used for each part of the node. I.e., the first word represents the GRIN tag, the second word represents the first node argument, etc.

For nodes with a *size* larger than a certain limit we must instead use a different layout, called *big* node layout (see figure 5.4). The *size* of a GRIN node is defined as "1 plus the number of node arguments", see section 2.3.3 on page 30. The reason that we need this alternate layout is that we use a *copying update* operation rather than *indirection nodes* (see section 5.3.2 on page 186). In the *big* layout a node is split into two parts, that can reside in different parts of the heap, and the last word of the first part is set to point to the second part.

This makes it possible to update a "small" closure with a larger node and still use copying (the first part of the node is copied onto the closure and the second part is allocated in new storage).

The detailed rules for which layout is used are as follows. Note that these rules depend only on the tag of the node in question, its type (F-tag, C-tag or P-tag) and its number of arguments. E.g., different constructors (C-tags) of the same datatype are not related at all in this regard. The size limit between small and big layout is depicted "L" below (it is selectable in the compiler, but defaults to 3 words).

- Values representing function closures (F-nodes) use *small* layout (we never update *with* an F-node, only *onto* an F-node[4]). All nodes must be at least "L" words (may require padding).
- Values representing whnfs (C-tags and P-tags) with "$size \leq L$" use *small* layout. All nodes must be at least 2 words (need space for a GC *moved indirection*).
- Values representing whnfs with "$size > L$" use *big* layout.

Note that there is no need to "tag" the second part of a big layout node, we will make sure that the pointer to it is always unique! The main implication of this is that the update operation must also copy the second part of big nodes, which in turn means that updates may require heap allocation in some cases.

Note also that no explicit information about node size, argument boxity, etc., is stored in a node. This is instead put in special tables that is used by the garbage collector (see chapter 8).[5]

Constant Applicative Forms

Some implementations of graph reduction treat CAFs very differently from ordinary functions, e.g., keep "CAF nodes" (containing the top-most constructors of all CAFs) outside of the ordinary heap. In the GRIN back-end we instead try to keep everything as uniform as possible, all CAF nodes will be stored in the normal heap. Initially, they will be closures (F-nodes without arguments, but padded to the minimal closure size, "L" above), and once evaluated will be updated just like any other closure. To keep track of all CAFs we use a table of pointers to the CAFs, called "*caf_roots*". This table is kept outside the heap and is initialised once at program startup (when all the initial CAF nodes are created in the heap), and then kept updated by the garbage collector. The important thing to note though is that all "data" that belong to CAFs are kept in the normal heap.

[4] The original G-machine [Joh84] could actually do that (and then continue with an UNWIND).

[5] Arrays are not yet supported by the GRIN back-end, but to do that it is probably necessary to add a third layout type with explicit size information.

Constant data

For practical reasons, to reduce the memory needs, we use special tables of constant nodes representing characters and small integers (the GRIN "CChar" and "CInt" tags). The nodes are "pre-built" in the generated code.

In addition to that, some GRIN store operations are completely constant, i.e., all their arguments are known constants at compile time. Nodes that such operations would "store" are also pre-built at compile time. Both these techniques are used by many other compilers.

These constant nodes are the only exception to the rule that all GRIN nodes are allocated inside the normal heap.

5.3 RISC code generation

The core of the RISC code generator, the actual code selection, is quite simple and standard, with a few but important exceptions. We do not use any advanced code generation technique, such as *tree pattern matching* [AGT89, FHP92], or try to produce optimal code in any sense. Instead we follow the philosophy of the GRIN part of the back-end, i.e., we use a simple code generator producing rather naïve code and then apply a number of optimisations to make the code more efficient. In fact, since most RISC instructions are very simple there is often no choice in what instructions to emit for a particular GRIN operation (the final GRIN code is also in a very simple form, see section 4.4.5 on page 161).

Below we will show how the most important GRIN constructs are translated into RISC code, in the form of examples rather than giving a complete implementation of a code generator (which would take a lot of space due to all the bookkeeping that is needed, and be mostly non-interesting). In particular we will describe the two most important features of the code generator:

- the handling of function calls (and registers),
- an optimisation of sequences of case expressions.

The main idea when generating code for function calls is to prepare for the *interprocedural coalescing* in the register allocator, the method used to achieve tailor-made calling conventions. The optimisation of case expressions is used as a remedy for a slight problem in the GRIN language, the inability to express "non-structural control flow". This optimisation is the only exception to the above claim that the code generation is very naïve.

5.3.1 RISC code structure

For each procedure, the code generator will produce RISC code organised as a *flow graph* of *basic blocks* [ASU86, section 9.4]. Each procedure will have a single *entry block* (this will later change when tailcalls are optimised, see section 7.4

on page 250), and one or more *exit blocks*. We will call this "per-procedure" flow graph the *intraprocedural flow graph*, to distinguish it from the *interprocedural* (program-wide) flow graph that is used later during the register allocation. Each basic block in the intraprocedural flow graph can have zero or more *predecessors* and zero or more *successors*. Each successor relation will always be explicit in the RISC code, i.e., a basic block with one or more successors will always end in a branch instruction (or some other control transfer instruction) explicitly jumping to the successor block.

Each basic block is uniquely identified by a RISC *label* and we will use the terms basic block and label alternatingly to denote a basic block together with its associated label.

The intraprocedural flow graphs that we generate will always be *directed acyclic graphs* (DAGs), i.e., there will be no *back edges* [ASU86, section 10.4] in the flow graphs. However, as with the entry blocks above, this will also change later during the tailcall optimisation. The structure of the generated flow graphs is discussed in more detail below (see section 5.3.3 on page 192).

5.3.2 Code selection

Two basic principles in the code generator are:

- create new virtual registers whenever a new register is needed,
- create new basic blocks whenever it can be useful.

The code generator will always use virtual registers rather than real machine registers for all kinds of values including function arguments and results, except in cases where an external calling convention forces the use of real machine registers. It will never try to reuse a virtual register for a different value. All such decisions are left to the register allocator. The ability to create a lot of basic blocks makes the code generator simpler, and it is easy to later eliminate all the unnecessary blocks that have been introduced (and the branches).

The basic RISC code generation procedure

The core of the code generator is a single procedure, called $\mathcal{R}$, that generates RISC code from a GRIN expression, and also performs all the bookkeeping that has to be done in a code generator (keeping track of various environments and tables, etc.). As an example, to generate code for the GRIN expression "e", we would call:

$$\mathcal{R}\ [e]\ \rho\ vs$$

Here, "ρ" is an environment mapping GRIN names to RISC registers, and "vs" is a list of *result registers*, where the result of the expression should be put. In the real implementation the code generation function is more complex, but the

above should be enough to demonstrate the most important parts of the code generation.

Function calls

The key idea that we use for function calls is to generate a lot of, seemingly unnecessary, register to register copy instructions for function arguments and return values. The purpose is to give the register allocator "maximal freedom" in deciding what registers to use at various points in the code. Hopefully it will be able to remove most of the copy instructions, and the ones that it leaves behind will actually be useful (decreasing the register pressure). This process is called *coalescing*, see section 6.2.5 on page 214. The result of this code generation method followed by the coalescing will be a tailor-made calling convention (argument and result registers) for each function.

The code generator part of this works as follows, the net effect of this is to insert register copies on "both sides" of function calls and returns:

1. Create a mapping *argRegs*, such that for each function f, $argRegs(f)$ returns a list of unique *virtual registers* to be used as argument registers when calling f.

2. Create an analogous mapping *retRegs*, for return registers. The number of return registers for each function will be the *maximal* number needed (compare with GRIN: the maximal size of a node value that the function can return).

3. In each function prelude (*entry point*): emit instructions to copy all argument registers (from *argRegs*) to fresh virtual registers (local temporaries).

4. In each function postlude (*exit point*): emit instructions to copy the value to be returned into the return registers (from *retRegs*).

5. Immediately *before* each function call (*call site*): emit instructions to copy all *outgoing* arguments into the argument registers of the function to be called (from *argRegs*).

6. Immediately *after* each function call (*call site*): emit instructions to copy the result of the call (from *retRegs*) to fresh virtual registers (local temporaries).

The above will result in a lot of register copies, many of which may seem unnecessary, but again, the point is to give maximal freedom to the register allocator to decide which ones can be removed (it is in a much better position to decide this than the code generator is).

As an example, we will show the code generated for a *call site* (the last two items in the list above). The resulting flow graph is illustrated in figure 5.5 on the following page. In addition to all the register copies, we also create two

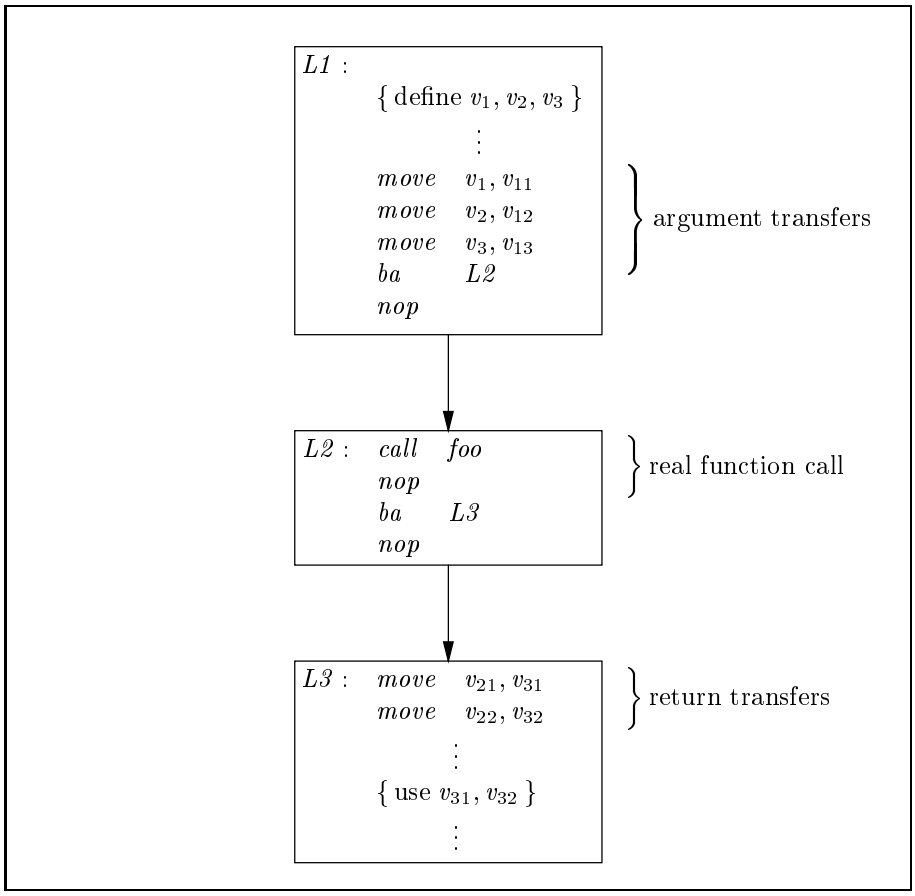

Figure 5.5: A function call site – argument and return transfers.

new basic blocks for each call site. This makes it easier to set up *data flow equations* in the register allocator (both the call point and the return point will be associated with distinct labels). The extra branches that are inserted are trivial to remove later (see section 7.7 on page 255).

In figure 5.5 we have assumed that the function *foo* takes three arguments in the virtual registers "v_{11}", "v_{12}" and "v_{13}" (from *argRegs*), and delivers a result in "v_{21}" and "v_{22}" (from *retRegs*). The other virtual registers used are temporaries that are local to the calling function. Note also that we must insert "*nop*" instructions to fill the *delay slot* of all control flow transfer instructions (see section 5.1.3 on page 172). This will be optimised later during the *instruction scheduling* (see section 7.8 on page 256).

No registers are "saved and restored" around function calls in the initial

RISC code. This is all taken care of later by the register allocator (hopefully many calls will need no saves and restores, due to the interprocedural register allocation). This means that the initial RISC code is not "runnable", registers can be clobbered by recursive function calls.

Tailcalls. Function calls that are tailcalls is currently treated exactly as above, with argument transfers before the call and return transfers after the call. Note that this means that the tailcall is no longer a tailcall (due to the return transfer)! This is intentional however, since we want to avoid a situation where all procedures that can potentially tailcall each other are forced to use the *same return registers*. Again, we want the decision of return registers to be left to the register allocator. Some tailcalls will be "reintroduced" later if the register allocator has managed to coalesce all the return transfer copies. However, this is probably not an ideal solution, some (but not all) tailcalls are really vital to keep as tailcalls and we have little control over that using this technique (see also the discussion in section 7.4 on page 250).

Sequencing – the GRIN bind operator

One of the most important GRIN constructs is the GRIN sequencing operator (written ";", called *bind*). The code generation for a GRIN bind expression is illustrated below, using the code generation procedure "$\mathcal{R}$" (see above) and a kind of pseudo-code that hopefully is self-explanatory. Here, "$getVars(x)$" is used to return all variables in "x" (if it is a GRIN node) and "$newTempRegs(n)$" is used to create "n" new virtual registers ("n" here equals the *size* of "x"):

$$
\begin{array}{lll}
\mathcal{R}\ [\, m\ ;\ \lambda\, x \rightarrow k\,]\ \rho\ vs & = & \mathit{let}\ \ [\, x_1\ \ldots\ x_n\,] = \mathit{getVars}(x) \\
 & & \quad\ [\, v_1\ \ldots\ v_n\,] = \mathit{newTempRegs}(n) \\
 & & \mathit{in} \\
 & & \text{— generate code for the lhs:} \\
 & & \mathcal{R}\ [\, m\,]\ \rho\ [v_1\ \ldots\ v_n] \\
 & & \text{— "bind" the result:} \\
 & & \mathit{let}\ \ \rho' = \rho\,[\, x_1 \mapsto v_1\,, \ldots\,,\ x_n \mapsto v_n\,] \\
 & & \mathit{in} \\
 & & \text{— generate code for the rhs:} \\
 & & \mathcal{R}\ [\, k\,]\ \rho'\ vs
\end{array}
$$

The bind operator does not really result in any code in itself. The only thing we need to do is to create a set of new virtual registers (according to the size of "x") and pass them on as *result registers* when generating code for the left hand side.

GRIN unit operations

The GRIN unit operation is used to "return" values, both as a normal function return and as an "internal return", e.g., to return a value out of a case expression to the code that follows. The code generator keeps track of if it is on the "returning spine" of a function, and will produce slightly different code for a unit operation depending on the context it is in.

Function return. If we assume that the return registers (from the *retRegs* table) for the current function are "$[\, v_0 \ \ldots \ v_k \,]$", a unit return of a node value would result in the following:

$$\mathcal{R}\ [\,\mathsf{unit}\ (tag\ a_1\ \ldots\ a_n)\,]\ \rho\ [\,v_0\ \ldots\ v_k\,] \quad = \quad emit \begin{bmatrix} move & \rho(tag), v_0 \\ move & \rho(a_1), v_1 \\ & \vdots \\ move & \rho(a_n), v_n \\ return & \\ nop & \end{bmatrix}$$

Note that "*tag*" is always a tag variable in the final GRIN code, and that "k" must not necessarily be the same as "n" (in general "$k \geq n$"). The copies that we generate at function return points are part of the overall code generation strategy for function calls discussed earlier (see page 183).

Internal returns. For unit operations that are "internal" to a procedure we will generate copy instructions in the same way, but instead of a "*return*" instruction we will emit a short branch to the current *exit label*. The concept of exit labels has not yet been discussed, but it is used internally in the code generation procedure "$\mathcal{R}$" in a similar way to how the *result registers* (the last argument of "$\mathcal{R}$") are used to signal what registers the result should be put in. The exit label instead signals if the code should jump to some label after it is done. Typically, an exit label will be created and passed down into the alternatives of case expressions that appear on the left hand side of a bind operation (all the alternatives should join when the case is done). The use of an exit label is not needed in most of the examples that we give here, so we have left it out of the presentation (it would simply have been an extra argument to $\mathcal{R}$).

GRIN memory operations

At the point of RISC code generation, all GRIN memory operations (fetch, store and update) will be annotated with a concrete GRIN *tag*, in as much as a tag is needed to actually implement the operation (this is discussed in section 4.2.3 on page 90). A consequence of this is that it is quite easy to generate RISC code for all GRIN memory operations, although it becomes a bit involved since

we must obey all the memory layout rules shown earlier (see section 5.2.3 on page 178).

Fetch operations. As an example, to load the tag word from a node in memory we would get a single *load* instruction (no tag annotation is needed in this case). The code generation rule becomes (the result should be in "v_0"):

$$\begin{array}{lcl} \mathcal{R}\ [\mathsf{fetch}\ p[0]]\ \rho\ [v_0] & = & \mathit{let}\ v_1 = \rho(p) \\ & & \mathit{in} \\ & & \mathit{emit}\ [\ \mathit{load}\ [v_1], v_0\ ] \end{array}$$

To instead load a node argument (a non-zero offset) we would have to obey all the node layout rules. As long as the node followed the *small* node layout, or the argument were located in the "first part" of a node with *big* node layout (see figure 5.4 on page 179), we would get the same code as above, but with an offset in the resulting *load* instruction. Assuming a word size of 4:

$$\begin{array}{lcl} \mathcal{R}\ [\mathsf{fetch}\ p[i]]\ \rho\ [v_0] & = & \mathit{let}\ v_1 = \rho(p) \\ & & \mathit{in} \\ & & \mathit{emit}\ [\ \mathit{load}\ [v_1 + 4 * i], v_0\ ] \end{array}$$

If the word to be loaded instead appears in the second part of a node with big node layout, the result will instead be two *load* instructions, where the first loads the indirection pointer and the second follows it. A trivial optimisation is to cache the load of the indirection pointer since it is the same for all words of the second part of the node. All but the first fetch operation can then reuse the indirection pointer.

Store operations. GRIN store operations are handled analogously to fetch operations, making sure to obey all the node layout rules. As an example, let us consider a simple store of a CCons node using the default *small* node layout. Recall the GRIN *register introduction* transformation (see section 4.2.9 on page 110), which makes sure that all arguments of a store operation are GRIN variables. This means that the actual CCons tag is not shown in the store operation other than as an annotation. The tag will already have been put in a register by a GRIN unit operation (or reused from somewhere else by

the common sub-expression elimination):

$$\mathcal{R}\ \left[\mathsf{store}^{\mathsf{CCons}}\ (t'\ x\ xs)\right]\ \rho\ [v_0] \quad = \quad \begin{array}{l} let\ \ v_1 = \rho(t') \\ \qquad\ v_2 = \rho(x) \\ \qquad\ v_3 = \rho(xs) \\ in \\ emit \left[\begin{array}{lll} store & v_1, [\%hp + 0] \\ store & v_2, [\%hp + 4] \\ store & v_3, [\%hp + 8] \\ move & \%hp, v_0 \\ add & \%hp, 12, \%hp \end{array}\right] \end{array}$$

No "heap full tests" are emitted at this stage, such tests will instead be inserted later (see section 8.3.3 on page 269).

As a simple optimisation, we detect store operations where the node value is a compile time *constant*. The only thing we need to do in such a situation is to load the address of a pre-built constant node, i.e., no real memory load is needed (see also section 5.2.3 on page 181). E.g., for a constant small integer node we would get the code:[6]

$$\mathcal{R}\ [\mathsf{store}\ (\mathsf{CInt}\ 5)]\ \rho\ [v_0] \quad = \quad \begin{array}{l} let\ \ l = lookupConstantNode(\mathsf{CInt}\ 5) \\ \qquad\ v_1 = newTempReg() \\ in \\ emit \left[\begin{array}{lll} sethi & \%hi(l), v_1 \\ or & v_1, \%lo(l), v_0 \end{array}\right] \end{array}$$

Above, "l" will be a symbolic address (an assembly *label*), so we must load its value into a register in two steps (see section 5.1.3 on page 172). Note also that we create a new temporary register to hold the intermediate value rather than to reuse the current result register ("v_0").

Update operations. A GRIN update operation is almost identical to a store operation. The only difference is that an update will overwrite an old node in memory (the closure) instead of allocating a completely new node.

The interesting thing happens when the node to be written follows the *big* node layout. In that case, the "first part" of the node (see figure 5.4 on page 179) will overwrite the node to be updated, and the "second part" of the node will be allocated in new heap (i.e., we get a heap allocation just as for a store).

Primitive operations. GRIN primitive operations will translate into either a few machine instructions or an *external* function call. All primitive operations

[6] Actually, this store operation does not obey the above claim that all operations have only variables as arguments. The reason is a "pre pass" to the code generation where we apply constant folding (and undo some of the work done by the *register introduction*, for constant node values), to make it easier for the RISC code generator.

in GRIN take only unboxed basic values as arguments, so they need not care about heap pointers or node values. E.g., a simple addition:

$$\begin{array}{lcl} \mathcal{R}\ [\,intAdd\ x'\ y'\,]\ \rho\ [v_0] & = & let\ \ v_1 = \rho(x') \\ & & \quad\ \ v_2 = \rho(y') \\ & & in \\ & & emit\ \ [\ add\ v_1, v_2, v_0\ ] \end{array}$$

Some more complicated primitives are implemented using function calls to external functions (usually written in C). An external function call will result in code that is very similar to the code emitted for normal function calls, inside the GRIN program (see figure 5.5 on page 184). The argument will be copied to the argument registers of the callee (which in this case are the fixed argument registers of the SPARC calling convention). After the call the result will be copied back into temporary virtual registers. The actual RISC instruction becomes an "*ecall*" instead of a "*call*". New basic blocks are created exactly as in figure 5.5.

Case expressions

We will describe the code generation for GRIN case expressions in two steps:

1. "Normal" case expression code: select the matching alternative.
2. An optimisation: short-circuit sequences of case expressions.

Normal case expression code. GRIN case expressions are rather simple to implement. Only "known tags" are allowed in the alternatives and all tags are constants that are known at compile time (see section 4.4.3 on page 159). This makes it quite simple to either build a *decision tree* or a *jump table*, depending on the size of the case expressions, to select the matching alternative. Currently we generate either a decision trees or a jump table, but it should be fairly straightforward to apply standard techniques to also build combinations of jump tables and decision trees [Sal81, Spu94]. A small reservation must here be made for the fact that all GRIN tags must have unique tag numbers (at least in the current implementation), which means that the actual tags that appear in a certain case expression can form a very "sparse interval". This makes the building of jump tables more difficult, but methods exist also to handle this [HM82].

Note also that currently "default alternatives" are not allowed in GRIN case expressions, but that will probably be allowed in the future (in some form).

Optimising sequences of case expressions. As discussed during the GRIN transformations there is no way in GRIN to express "non-structured" jumps. As an example, consider the GRIN code in figure 5.6 on the next page.

```
fetch p[0] ; λ t' →
fetch p[1] ; λ a →
(case t' of
    CLeft  → unit (CLeft a)
    Ffoo   → foo a
    CRight → unit (CRight a)
) ; λ (s' b) →
case s' of
   CLeft  → ⟨m1⟩
   CRight → ⟨m2⟩
```

Figure 5.6: A sequence of GRIN case expressions.

The sequence of two case expressions in figure 5.6 is a very typical example of GRIN code. The fetch operations and the first case expression is probably what remains of an inlined call to *eval* ("*eval p*"), where "*p*" turned out to point either to a closure of the *foo* function, or to an already evaluated node. The second case expression is the normal code, that the programmer wrote to inspect the value. Even without an initial *eval* code similar to this can appear when using general inlining on the GRIN level (see section 4.3.10 on page 135). Inlining can also create sequences of more than two case expressions.

The problem with this code is that the natural implementation would risk examining the same value twice (when the original value is already evaluated). Some instances of this problem can be fixed on the GRIN level (see the *case hoisting* transformation, in section 4.3.11 on page 137). Unfortunately there is no general and good solution to this problem on the GRIN level[7], so we must make sure that the RISC code generator can handle this kind of code well.

Our solution is to keep track of which alternatives of the first case expression that result in a return of a *known tag*! If the first case expression returns a known tag, then the test in the second case expression is really unnecessary. Note that the "scrutinised tag" in the first case expression is totally irrelevant (for the case optimisation). What is interesting is only the *return values* of the first case expression, and the scrutinised tag of the second case expression. The key observation is that when an alternative of the first case expression returns a known tag, the code can jump directly into the corresponding branch of the second case expression, bypassing the extra test (on "*s'*").

The result of this is a kind of *case short-circuit* optimisation. The flow graphs that result from the code in figure 5.6, with and without the optimisation, are shown in figures 5.7 and 5.8, respectively.

[7] We do not consider continuation functions a very good solution.

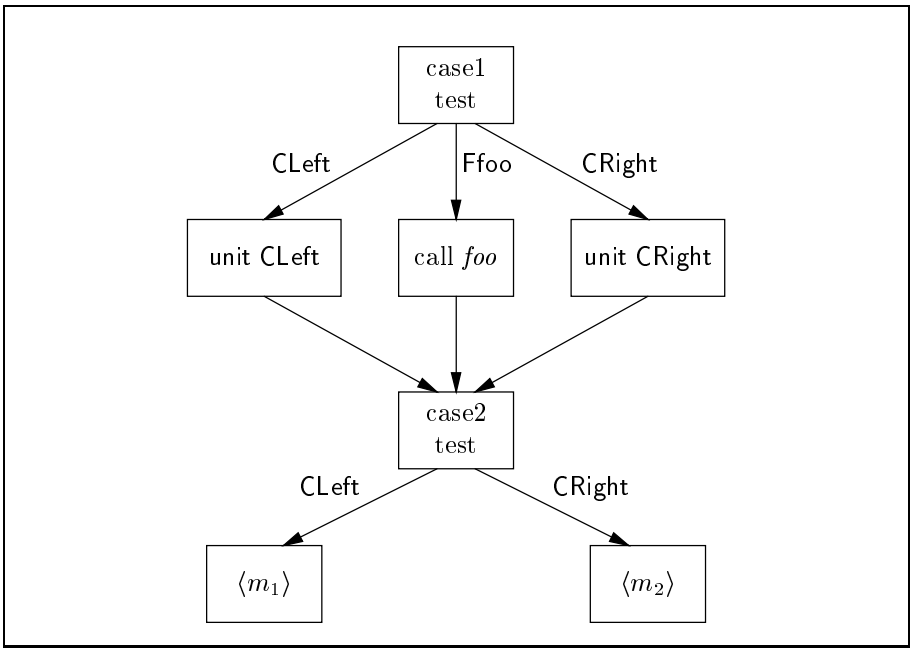

Figure 5.7: Naïve case implementation (of the GRIN code in figure 5.6).

The implementation of this optimisation is very similar to the GRIN *case hoisting* transformation (see section 4.3.11 on page 137), but here we only insert different branch instructions instead of duplicate code like in the GRIN transformation. In fact, the case short-circuit optimisation should be seen as a complement to the GRIN case hoisting transformation. The case short-circuiting done in the RISC code generator will optimise all case expressions where the GRIN case hoisting did not apply (because it risked duplicating too much code).

The actual bookkeeping that needs to be done for this transformation is rather involved, to handle sequences of more than two case expressions, nested case expressions, etc., and we will not explain the details here.

Update operations. To further optimise situations with sequences of case expressions, update operations can also be considered. Returning to the GRIN code in figure 5.6, it is possible that the first operation in both "⟨m_1⟩" and "⟨m_2⟩" is an update (of the node pointed to by "p"). It may not be, if the closure "Ffoo" was deduced to be non-shared by the heap points-to analysis (see chapter 3), or if the first case expression is not at all an inlined call to *eval*. In any case, if the first case expression finds an already evaluated node, clearly an update must be unnecessary and can safely be skipped. This is easy to do

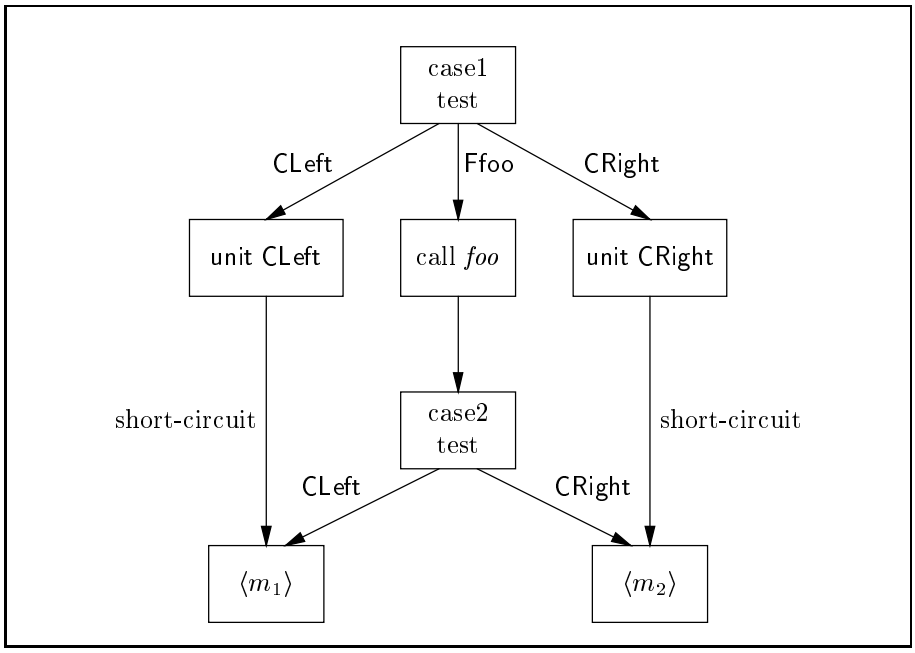

Figure 5.8: Case implementation with case short-circuit.

in conjunction with the normal case short-circuit optimisation, simply let the inserted branches jump a bit "longer" and skip both the test and the update operation.

It could be argued that this whole optimisation should be done as an independent pass, after the RISC code generation, but that would probably be much harder. Note that this is *not* just a "branch short-circuit" optimisation, we also eliminate an unnecessary test and possibly an unnecessary update. To detect such things on RISC code seems very difficult.

5.3.3 Structure of emitted code

As explained earlier, the RISC code generation for each GRIN function produces an intraprocedural flow graph of basic blocks. The intraprocedural flow graphs in the initial RISC code are always *directed acyclic graphs*, DAGs. Intraprocedural loops (self-recursive tailcalls) appear first after tailcalls are optimised (see section 7.4 on page 250).

Function calls and recursion

All function calls are treated as explained above, i.e., for a function call we generate code to setup arguments, do the call, and then take care of the result. No registers are "saved and restored" around function calls. In some sense this would be almost enough, since we generate new virtual registers all the time, if it had not been for recursion. In the presence of recursion it is sometimes necessary to save and restore around function calls to avoid registers being clobbered. The consequence of this is that the initial RISC code is not "runnable", not even on an ideal machine with an infinite number of virtual registers.

In fact, in the initial RISC code we do not even save and restore the return address register ("$\%o_7$") around function calls. This is really the same problem as saving other "clobbered registers" around function calls, and all that will be taken care of by the register allocator (see section 6.2.3 on page 202).

Values and live ranges

It is interesting to note the distinction between the two notions of *value* and *virtual register* (or rather *live range* as the virtual registers are called in the description of the register allocator). A new *value* is usually said to arise for each definition in the code. For imperative languages, the consequence of this is that a *live range* consists of several values connected by *common uses*, also known as *def-use chains*. Given the "single assignment nature" of functional languages (GRIN is *static single assignment* (SSA), see section 2.3.7 on page 34), we might expect the two notions of value and live range to coincide, since each "variable" is only assigned once. But unfortunately that is not at all the case in the RISC code generated from GRIN. Instead we will get code where the same live range (virtual register) can be assigned at different points in the code, i.e., RISC does not have the single assignment property that GRIN has. E.g., a GRIN case expression on the left hand side of a bind operator (we earlier called this an "internal" case expression), will result in code where all the alternatives of the case expressions assign the same virtual registers and then jump to the basic block where the control flow *joins* (the "bind point").

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 (we might have hoped for some things being "easier" due to the functional intermediate language). Actually, we are a little better off in that we do not need a separate *renumber* phase in the register allocator to find live ranges, the single assignment property of GRIN is enough to ensure that.

Of course, if we really wanted SSA code it would be quite easy to achieve that during the RISC code generation. E.g., we would need to insert *phi nodes* [CFR+91] after each "internal" case expression. But that is not the approach taken by the GRIN back-end, our RISC code is more like conventional machine code in this respect. Which, seen in retrospect, might have been a

mistake, but too much work on the GRIN back-end had already been done (designed and implemented) before we realised the real benefits of SSA also at the machine code level.

Chapter 6

Interprocedural register allocation

This chapter will describe two register allocation algorithms, i.e., algorithms to map all the *virtual registers* used by the RISC code generator into uses of real machine registers, possibly combined with the use of memory locations. The first algorithm, called the *base* algorithm, uses interprocedural (and program-wide) *graph colouring*, and is the algorithm that is implemented in the current GRIN back-end. The second algorithm, called the *extended* algorithm, is a variant of the first aimed at being more practical (it is still interprocedural, and uses graph colouring, but is not necessarily program-wide).

6.1 Background

Two of the most important goals for the GRIN back-end have been:

- achieve good register utilisation,
- make function calls cheap.

Unfortunately, these two goals are somewhat conflicting when it comes to programs that are *call-intensive*. The reason is that register allocation is normally done on a per-procedure basis (called *intraprocedural* register allocation, and in more detail *local* or *global* register allocation, see section 1.3 on page 10). Intraprocedural register allocation will normally force all registers in use by a procedure to be saved to memory before a call can be made, and then restored into registers at return from the call.[1] This has the undesirable effect that the better (intraprocedural) register allocation we achieve, i.e., the more registers

[1] This problem can be reduced by using *callee-saves* registers, but that is not a general solution, it has other drawbacks.

that are in use, the more expensive will procedure calls be, due to the increase in memory-traffic at each call site. Obviously this is not very good for code generated from functional languages where there is often very "little code" between function calls.

One solution to this dilemma is to use a very simple intraprocedural register allocator and not try very hard to use a lot of registers, thus automatically avoiding most of the save and restore overhead for function calls. But of course, the resulting code will not be very efficient, at least on modern machines where good register utilisation is very important.

6.1.1 Interprocedural register allocation

A better solution to the save and restore problem than to use no registers at all is to start from the following observation:

> *use a register allocation algorithm that "knows" what happens on the "other side" of function calls.*

I.e., if the algorithm can guarantee that a certain register will not be clobbered by a function call, then the register need not be saved and restored around the call. By definition, such an algorithm is called *interprocedural* (see the discussion in the introduction, on page 14).

In addition to reducing the save and restore overhead at procedure boundaries, which is how the benefits of interprocedural register allocation are normally characterised (e.g., see [Cho88] or [Ste91]), there is yet another way that we can use it to even further reduce the function call and return overhead. And that is by making more effective use of registers for passing function arguments and returning results from functions. This may in fact be even more important for functional languages than to avoid the save and restore overhead.

One possible drawback of interprocedural register allocation is that it requires rather precise information about the function call behaviour of the program to work well, i.e., it needs know the *call target* for as many function calls as possible at compile time. With many "unknown" (or *indirect*) calls, the possibilities for good interprocedural register allocation become greatly reduced. This conflicts in a way with things like traditional *separate compilation* and the "one procedure at a time" optimisation model that many compilers use. Fortunately, in the GRIN back-end the entire program is available, and we have access to very precise information about function calls, due to the use of the program-wide control flow analysis (see chapter 3). One example of this precise information is the GRIN *call graphs* (e.g., see figure 3.9 on page 74), which makes it possible for interprocedural register allocation to work really well.

6.2 The base algorithm

We will now describe an interprocedural register allocation algorithm that we believe is particularly well suited for *call-intensive code*. This algorithm is implemented in the current GRIN back-end, and we call it the *base* algorithm, to distinguish it from the *extended* algorithm that we will describe later (see section 6.3 on page 233). Figure 7.1 on page 248 shows how the register allocator fits into the organisation of the RISC part of the GRIN back-end.

The *base* algorithm was first described in [Boq95a, Boq95b] and is specifically designed to minimise the function call and return overhead. To summarise its features:

- 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 many cases, no extra "register shuffling" will be necessary at the call site.
- Likewise with procedure *return values* (we will normally use more than one register to return a result).
- Local variables that are *live*[2] across a call site, can often be kept in a register during the call, thus avoiding the save and restore overhead.
- Some save and restore instructions can not be avoided (due to recursion), but we optimise the placement of these instructions to avoid unnecessary memory references.

6.2.1 Overview

We will first give an overview of the main parts of the register allocator (see figures 6.1 and 6.2), and then describe each part in greater detail.

The basic idea of the algorithm is to build an *interference graph* (or *conflict graph*, see the introduction, on page 11) for the complete program and then *colour* it using a number of colours equal to the number of available machine registers. We initially assume that all values are allocated to registers (called *virtual registers* or *live ranges*). If the allocator fails to find a colour for some value, 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.

The starting point of our allocator is the *optimistic* version of Chaitin's graph colouring algorithm described by Briggs *et al.* [BCKT89, Bri92]. The main differences between Briggs' allocator and ours are:

[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.

- It is *interprocedural*, both Chaitin and Briggs discuss only global register allocation.
- We use *interprocedural coalescing*, which is very effective in handling parameter passing and functions returning results in several registers. It achieves a kind of *targeting* beyond procedure boundaries.
- 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 optimise the placement of save and restore instructions for caller-saves registers (around recursive calls) using a technique related to Chow's *shrink-wrapping* [Cho88].

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* [CH84], even though the latter has live range splitting as one of its basic operations. As a consequence of the lower accuracy, *coalescing* is not possible in Chow and Hennessy's framework (see the discussion in section 6.4.2 on page 239). Instead they use *copy propagation* earlier during the compilation, but that is insufficient for our purposes.

The algorithm

The main phases of the register allocator are shown in figure 6.1 on the next page. It consists of:

- *ICFG* (*Interprocedural Control Flow Graph*): connect all intraprocedural flow graphs into a program-wide flow graph using *call* and *return edges* (see section 6.2.2 on page 200).
- *Save locals*: insert save and restore instructions around (recursive) calls. The actual placement of the instructions is optimised using a *shrink-wrap* technique, to avoid unnecessary saves and restores (see section 6.2.3 on page 202).
- *Build-colour cycle*: perform the actual graph colouring. The build-colour cycle is iterated until all virtual registers have been either coloured or spilled to memory (see figure 6.2).
- *Post process*: modify the RISC code for the program according to the result of the graph colouring, i.e., change all uses of virtual registers into uses of real machine registers, assign stack slots to all spilled live ranges, etc. (see section 6.2.11 on page 230).

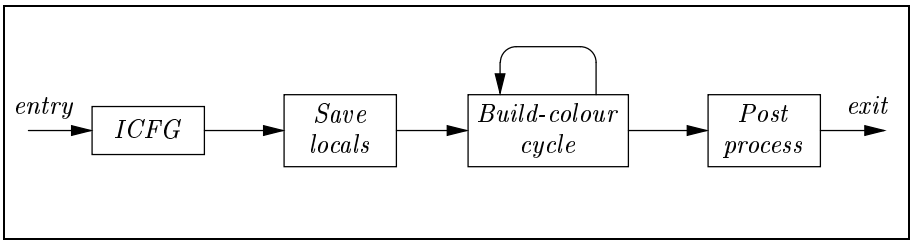

Figure 6.1: The main phases of the register allocator.

The so-called *build-colour cycle* is the core of any Chaitin-style graph colouring algorithm. Our variant of this is shown in figure 6.2 on the next page (contrast this to the standard Chaitin allocator in figure 1.2 on page 13).

The main parts of our build-colour cycle are:

- *build*: performs an interprocedural live variable data flow analysis and builds the interference graph (see section 6.2.4 on page 210).
- *coalesce*: a kind of low level *copy propagation*, eliminates unnecessary copies by combining live ranges (see section 6.2.5 on page 214).
- *spill costs*: approximates the spill cost for each live range (see section 6.2.6 on page 219).
- *simplify*: the main colouring heuristics, simplifies the interference graph, possibly choosing spill candidates (see section 6.2.7 on page 222).
- *select*: assigns colours to live ranges in the reverse order as they were chosen by simplify (see section 6.2.8 on page 225).
- *spill*: inserts spill code for spilled live ranges. This means creating a number of very short live ranges instead of a few long ones (see section 6.2.9 on page 226).
- *split*: inserts code to split a long live range into shorter live ranges (see section 6.2.10 on page 228).

The overall structure of our build-colour cycle is very similar to the one by Briggs, and especially the ordering of the *select* and the *spill* phases, which is due to our use of the *optimistic* colouring of Briggs. Our addition is the *split* phase (see section 6.2.10 on page 228).

Note also that compared to Chaitin and Briggs we need no *renumber* phase. The purpose of *renumber* is to get "the right number of names" [Cha82], meaning that *disjoint uses* of the same virtual register need not be assigned to the same machine register. This separation is done by identifying *def-use chains* [ASU86,

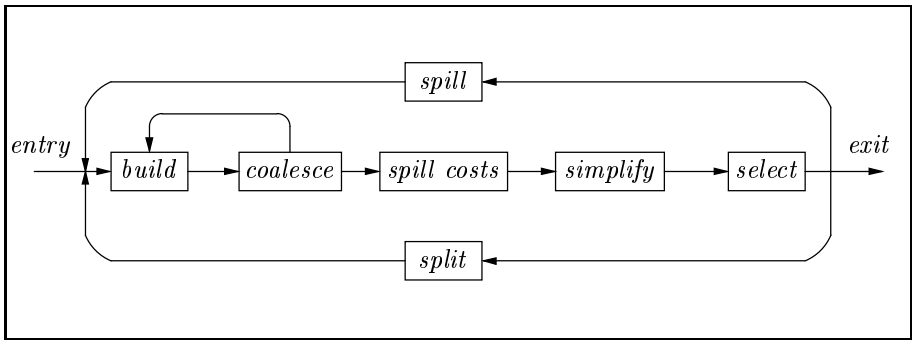

Figure 6.2: Our *build-colour* cycle.

section 10.6]. This is not an issue in the GRIN back-end, due to the GRIN static single assignment property and the way we generate RISC code from GRIN (see the discussion in section 5.3.3 on page 193). We will never get a situation where a virtual register is reused for a different value, and hence need no *renumber* phase.[3]

6.2.2 The ICFG phase

The RISC code that is input to the register allocator is organised as a set of intraprocedural flow graphs (one per procedure, see section 5.3.3 on page 192). However, to prepare for the interprocedural graph colouring we also need information about the function call and return behaviour of the code.

Recall that the GRIN language requires all function calls to be to statically known functions, i.e., we do not allow any indirect calls. Some information about calls can be found in the GRIN *call graphs*, e.g., see figure 3.9 on page 74. However, this call graph is not exact enough to be used during the register allocation (it does not give information for each *call site* and each *return point* of a procedure, which is what we need now).

An interprocedural flow graph

To access function call and return information during the register allocation we will instead build a "super flow graph", a program-wide flow graph where all the intraprocedural flow graphs are connected using *call* and *return edges* [Mye81]. We call the result the *Interprocedural Control Flow Graph* (ICFG) following Landi and Ryder [LR91]. An example of how call and return edges are used to link together the intraprocedural flow graphs are shown in figure 6.3.

[3] There is actually one situation where a renumber phase would be useful, see the discussion in the *split* phase (in section 6.2.10 on page 228).

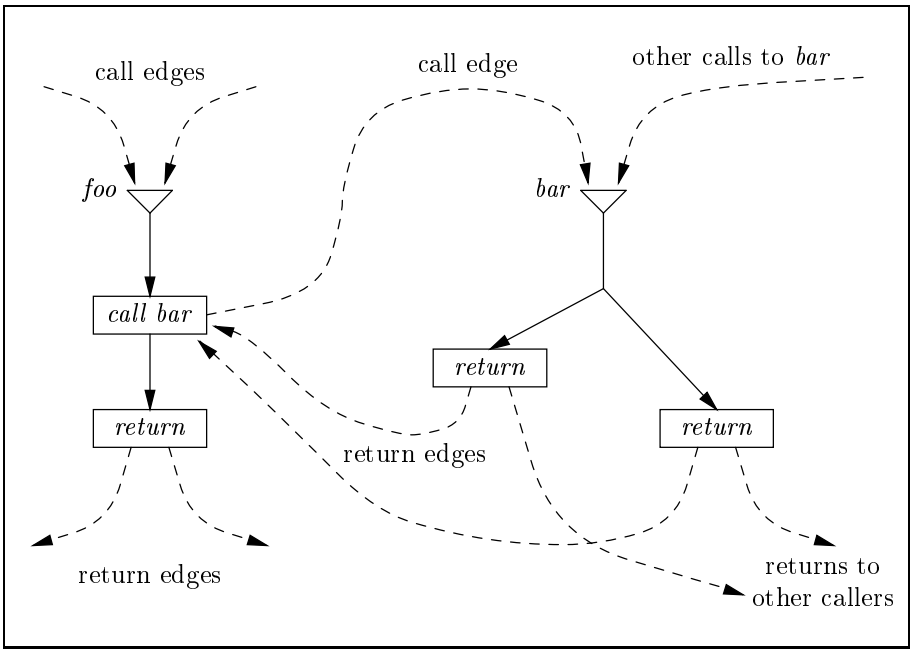

Figure 6.3: ICFG: call end return edges.

Figure 6.3 shows two interprocedural flow graphs, for the functions *foo* and *bar*, and assumes that *bar* is called by *foo* (in addition to being called by others). Note that for each *call edge* there will be a number of *return edges*, one for each *return site* in the flow graph of the called function.

Below, when talking about the *interprocedural successor relation* (or predecessor relation) we will mean the union of the ordinary intraprocedural flow graph edges and the interprocedural edges (call end return edges). This means that a call block will have two successor blocks. The first is the intraprocedural successor, since each call site is split into three basic blocks (see figure 5.5 on page 184), and the second successor is the entry block of the called procedure. Each exit point of a procedure will have as successors all the callers of the procedure (or actually, all intraprocedural successors to the call blocks). Analogously for the predecessor relation.

The algorithm

The ICFG can be constructed quite easily from the RISC program. We give an algorithm to find the *set of all call and return edges*. Each edge is determined by two basic block numbers (labels), the source and the destination of the edge.

1. create three mappings (all returning basic block numbers):
   (a) *entry_block*(f): the entry block for function f,
   (b) *call_blocks*(f): the call blocks for function f,
   (c) *exit_blocks*(f): the exit (return) blocks for function f,
2. for each function f, and for each basic block $i \in$ *call_blocks*(f), do:
   (a) check the function call in block i, assume the call is to g,
   (b) add $\langle i, \textit{entry\_block}(g)\rangle$ as a *call edge*,
   (c) for each block $j \in$ *exit_blocks*(g), add $\langle j, \textit{Succ}(i)\rangle$ as a *return edge*.[4]

Depending on the actual data structures that is used to represent a RISC program, the above can be done in one or two passes over the code. In the current GRIN back-end, the three mappings *entry_block*, *call_blocks* and *return_blocks*, are already part of the RISC data structure, so the whole ICFG can be built simply by examining all call blocks in the program.

6.2.3 The save locals phase

This section will explain why we must insert save and restore instructions around certain function calls, even when using an interprocedural register allocation algorithm which is intended to avoid exactly that problem. We will also show how the actual placement of save and restore instructions can be optimised to avoid unnecessary memory references.

Recursion

Although interprocedural register allocation can often avoid the saving and restoring of local variables around function calls that a global allocator must do, this is not always possible. When a local variable of a procedure is *live* 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 where the caller and the callee belong to the same SCC.

In previous work there are some variations on where the save and restore instructions can be placed. In figure 6.4 on the next page we show two different ways, as used by Steenkiste [SH89] and Wall [Wal86] 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.

[4] Here, "*Succ*" denotes the intraprocedural successor block. Again, this is due to the organisation of call sites in RISC code (see figure 5.5 on page 184).

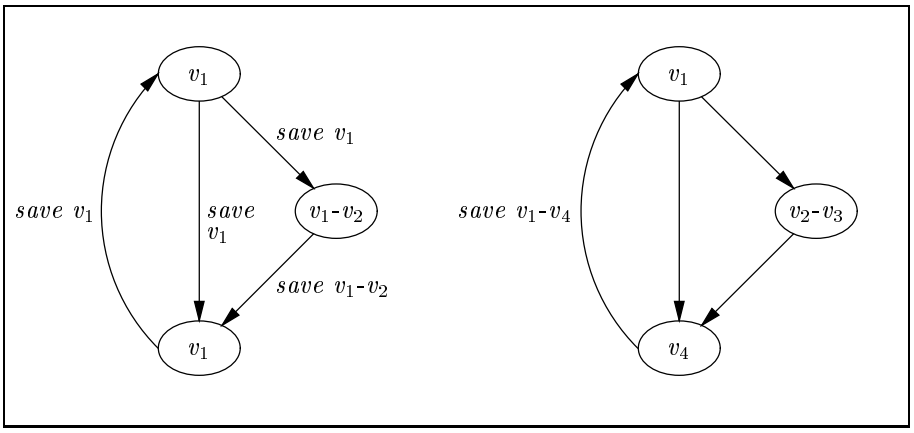

Figure 6.4: Different ways to handle recursion, Steenkiste vs. Wall

In the first solution, the live variables of each procedure 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 [ASU86, section 10.4]. For Steenkiste's bottom-up allocation (see section 6.4.3 on page 241), the incremental method is natural since procedures in the same SCC can use the same registers. Because of this, more registers will be free higher up in the call graph. In Wall's method (see section 6.4.3 on page 242) the allocation process is simplified by removing all back edges from the call graph before the allocation is done. After allocation he inserts saves and restores of all registers at the back edges. In our allocator, we have chosen the incremental method because it seems more natural in our setting, although the choice is not very critical compared to in Steenkiste's algorithm.

The Steenkiste method is also simpler because all register saves will occur in the procedure where the register is defined (all registers that are used for local variables and temporaries "belong" to exactly one procedure). This fits well together with the way the register allocator will *spill* registers, that is always done in the stack frame of the procedure defining a live range (see section 6.2.9 on page 226). It will also make garbage collection easier (see chapter 8).

In addition to saving registers used to hold local variables around potentially recursive calls, we will use the same technique to save and restore the special *return address* register. Of course, the return address is a bit special in that it is always live, and must be saved before all calls (not only recursive calls), but the method we will use works quite well for the return address too (see below). Recall that during the RISC code generation we did not insert any save and restore instructions for the return address at all, because we would optimise the

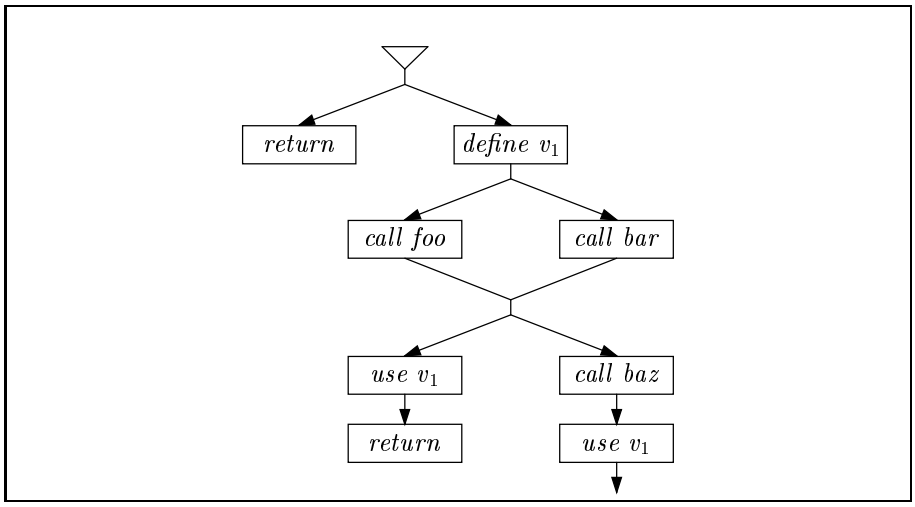

Figure 6.5: A flow graph illustrating the shrink-wrapping problem.

exact placement later (see section 5.3.3 on page 192).

Placing save and restore instructions

A naïve solution to the problem of placing save and restore instructions would be to simply save all live registers immediately before each function call where the caller and the callee are in the same call graph SCC, and then restore the registers again immediately after the call.

However, such a simple method would not result in very good code. As an example consider the procedure flow graph in figure 6.5.

Figure 6.5 illustrates several things. First, we would like to avoid saving the return address until we are certain that a call will occur (i.e., in the "right branch" of the figure). Second, if we assume that all the function calls are potentially recursive, the register "v_1" should be saved as soon as we know that a call will be done (a call where "v_1" is live). Here this means that it should be saved immediately after its definition (it is always better to save as early as possible, to decrease register pressure). After the first call, to either *foo* or *bar*, "v_1" should not be restored if we are also going to call *baz*, since there is no use inbetween the calls. On the other hand, we must restore "v_1" if the returning branch is taken after the control flow join. Similarly, "v_1" should not be saved before the call to *baz* even if it is live over that call. The return address behaves in a similar way, it should be saved before the first calls and then not restored until it is needed again.

The problem gets even harder if we have more complicated *fork* and *join*

patterns in the control flow graph (recall that our intraprocedural flow graphs are DAGs, see section 5.3.3 on page 192).

Shrink-wrapping

Our solution to this problem will be a technique similar in spirit to Chow's *shrink-wrapping* [Cho88]. Chow's technique was originally used to optimise the placement of save and restore instructions for *callee-saves* registers. In short, a callee-saved register need not be saved until it is certain that it will be clobbered. The intent was that saves and restores that were originally put at procedure entry, and exit, respectively, could be moved towards the points where the callee-saves registers became clobbered ("shrink-wrapped"). Chow used standard data flow techniques to place saves and restores. To avoid a possibly unnecessary save or restore can also be seen as a way to eliminate a *partial redundancy* [MR79].

In our setting all registers are effectively *caller saved*.[5] As a result, the technique we will use becomes more or less the "opposite" to Chow's shrink-wrapping. It can be visualised in the following way: all saves and restores are initially put immediately before and after each function call, and are then "floated outwards".[6] The following will hold after the optimisation:

- We do not save until it is necessary, but when it is, we save as quickly as possible.
- We do not restore until it is necessary, but when it is, we restore as quickly as possible.

The word "necessary" has a rather broad meaning in this case. E.g., the reason that it is necessary to do a restore does not have to be that the register is about to be used. It can also be that two control flow paths *join*, only one of which has saved the register (see also the discussion below, and figure 6.8 on page 210).

The two problems of shrink-wrapping callee-saved and caller-saved registers may seem totally symmetric (one floats inwards and the other floats outwards), but unfortunately it seems as if the caller-saves case is a bit more complicated. It is not absolutely clear to us why this is the case, but one reason could be that a caller-saves register may be *restored* several times (if it has multiple uses with calls inbetween), but this will never happen in the callee-saves case. The effect of this is that our data flow equations will be a bit more complicated than Chow's shrink-wrapping equations.

[5] Ignore the return address register for now, we will discuss that later (page 209).

[6] In what follows we will call our technique *shrink-wrapping*, even if a more appropriate name would have been *reverse shrink-wrapping*.

The shrink-wrapping algorithm

Our algorithm works in several steps and uses a number of different intraprocedural data flow attributes.[7] Remember that there are no loops in our intraprocedural flow graphs at this point, which means that most of the attributes can be calculated in a single traversal of the flow graph (with one exception, see below). The algorithm below is applied to RISC code with no save and restore instructions inserted, and gives as a result an optimised placement for the save and restore instructions that are necessary to insert in the program.

For each procedure we do the following (each step is shown in detail in figure 6.6 on the next page, continued in figure 6.7):

1. Calculate *Live*, in a single bottom-up traversal of the flow graph.

2. Calculate *Clob*, i.e., registers that need to be saved around calls using *Live* and the call graph SCCs. This is done by checking all call sites in the current procedure. After this step, all data flow attributes used will be *restricted* to registers that appear in *Clob* ("interesting registers").

3. Calculate *Ant* (a *use* of a register is *anticipated*), and *May* (a register *may* have been *defined* earlier). Implicitly, both these include only the interesting registers (appearing in *Clob*). Note how the flow is "stopped" (killed) by call sites (the use of "\" in Ant_{in} and May_{out}). As with *Live* this can be done without iteration.

4. The *Ant* and *May* attributes give use/def information for the interesting registers, but we need more information about function calls. For that we calculate *NeedSave* (registers that need to be saved, that will be clobbered by an upcoming call), and *NeedRestore* (registers that need to be restored, that have been saved earlier but not restored after a call).

5. Combine steps 3 and 4 to give a resulting optimised placement of save and restore instructions. $Save(i)$ gives the registers that should be saved in basic block i. $Restore(i)$ gives the registers that should be restored in basic block i.

All equations that use the intraprocedural flow graph predecessor or successor relations (*Pred* and *Succ*) should really include a case for when there are no predecessors (or successors), returning an empty set, but we have left that out for space reasons.

The separation of the last two steps above is a slight simplification, since the equations in 4 and 5 are really mutually recursive. The easiest way to solve that is to iterate them together. However, the only need to iterate at all is the

[7] Our data flow attributes are set based, i.e., for each basic block number they will give a set of live ranges (registers to save). Most attributes come in two versions, labelled "*in*" and "*out*", giving the value at basic block entry, and exit, respectively.

1. Find liveness information:

$$Gen(i) = \{ \text{ the set of all registers used before defined in basic block } i \ \}$$

$$Kill(i) = \{ \text{ the set of all registers defined in basic block } i \ \}$$

$$\begin{cases} Live_{out}(i) = \bigcup\limits_{j \in Succ(i)} Live_{in}(j) \\ Live_{in}(i) = Gen(i) \cup (Live_{out}(i) \setminus Kill(i)) \end{cases}$$

2. Find interesting registers (that need to be saved)

$$Clob(i) = \begin{cases} Live_{in}(i), & \text{if } i \text{ is a potentially recursive call block} \\ \emptyset, & \text{otherwise} \end{cases}$$

3. Find how the interesting registers (from 2) are used and defined:

$$\begin{cases} Ant_{out} = \bigcap\limits_{j \in Succ(i)} Ant_{in}(j) \\ Ant_{in} = Gen(i) \cup (Ant_{out}(i) \setminus Kill(i)) \end{cases}$$

$$\begin{cases} May_{in} = \bigcup\limits_{j \in Pred(i)} May_{out}(j) \\ May_{out} = Kill(i) \cup (May_{in}(i) \setminus Clob(i)) \end{cases}$$

4. Relate the interesting registers to function calls:

$$\begin{cases} NeedSave_{out}(i) = \bigcap\limits_{j \in Succ(i)} NeedSave_{in}(j) \\ NeedSave_{in}(i) = NeedSave_{out}(i) \cup Clob(i) \end{cases}$$

$$\begin{cases} NeedRestore_{in}(i) = \bigcap\limits_{j \in Pred(i)} NeedRestore_{out}(j) \\ NeedRestore_{out}(i) = NeedRestore_{in}(i) \cup (Clob(i) \setminus Restore(i)) \end{cases}$$

Figure 6.6: Data flow equations for save and restore placement (part 1).

5. Combine 3 and 4 to find exact save and restore placement:

$$
\begin{aligned}
&Save(i) = \\
&\quad NeedSave_{out}(i) && (1)\\
&\bigcap\ May_{out}(i) && (2)\\
&\bigcap \left(\bigcap_{j \in Pred(i)} (\neg NeedSave_{out}(j) \cup (NeedSave_{out}(j) \cap \neg May_{out}(j))) \right) && (3)
\end{aligned}
$$

(1) a save is needed
(2) the register is defined
(3) a save could not be placed earlier

$$
\begin{aligned}
&Restore(i) = \\
&\quad NeedRestore_{in}(i) && (4)\\
&\bigcap\ (Ant_{in}(i) && (5)\\
&\quad \bigcup \neg (\bigcup_{j \in Succ(i)} NeedRestore_{in}(j))) && (6)\\
&\bigcap \left(\bigcap_{j \in Pred(i)} (\neg NeedRestore_{in}(j) \cup (NeedRestore_{in}(j) \cap \neg Ant_{in}(j))) \right) && (7)
\end{aligned}
$$

(4) a restore is needed
(5) the register will be used
(6) or, a restore could not be placed later (join point)
(7) a restore could not be placed earlier

Figure 6.7: Data flow equations for save and restore placement (part 2).

use of the *Restore* attribute in the equation for $NeedRestore_{out}$. Without that it would be possible to calculate all the attributes in a single, either bottom-up or top-down, traversal of the flow graph. Unfortunately we need that use of *Restore* to prevent a register that has already been restored from being restored again. In practice this is not a problem though, the iteration stabilises very quickly.

The uses of set complement ("¬") might seem a bit frightening since we are dealing with sets of *virtual registers*, but the complements can easily be eliminated and replaced by a combination of membership tests and set difference.

Finally, after the data flow iteration is done, we insert RISC *save* and *restore* instructions (see figure 5.2 on page 172) according to the final *Save* and *Restore* attributes. We try to place a save instruction as early as possible in a basic block, with the exception that the register must be defined before we save it. Restore instructions are always placed at the beginning of basic blocks. Note that we use the special RISC *save* and *restore* instructions rather than ordinary *store* and *load* instructions. The reason for this is to make the implementation simpler. First, we do not need to bother about allocating *stack slots* at this point (this is all done after the register allocation is finished, when all *spills* are also known). Second, using special instructions makes it very easy to identify saves and restores later in the allocator, e.g., in the *spill costs* estimation phase (there is no need to save and restore a live range that is already spilled to memory).

The return address register

As noted above, the proposed method has to be modified slightly to also handle saves and restores of the special return address register. This is not very surprising, since the return address really behaves much more like a callee-saves register than a caller-saves register (the "current" return address is really the one put there by the previous procedure, and we are not allowed to destroy it).

Normally a save of the return address would be inserted at procedure entry and restores inserted at all procedure exits. After that, the saves and restores can possibly be moved "inwards" into the procedure to avoid saving if no call is made. However, it turns out that we can achieve exactly the same effect using our above algorithm for caller-save registers. To do this we need to modify the equations in figure 6.6 and 6.7 slightly (we did not include this because it would make the equations even more complicated, although not very much). The *Clob* attribute must include the return address registers, for all call blocks (not only recursive calls). We also need to propagate demand for the return address from all return points and add that the return address register is defined at procedure entry. This is done in the equations for Ant_{out} and May_{in}, they should both give the return address register rather than an empty set in the case where there is no successor (predecessor).

In figure 6.5 it was obvious that the return address save should be moved into the call branch, but it is not always that simple. In fact, there are flow graphs

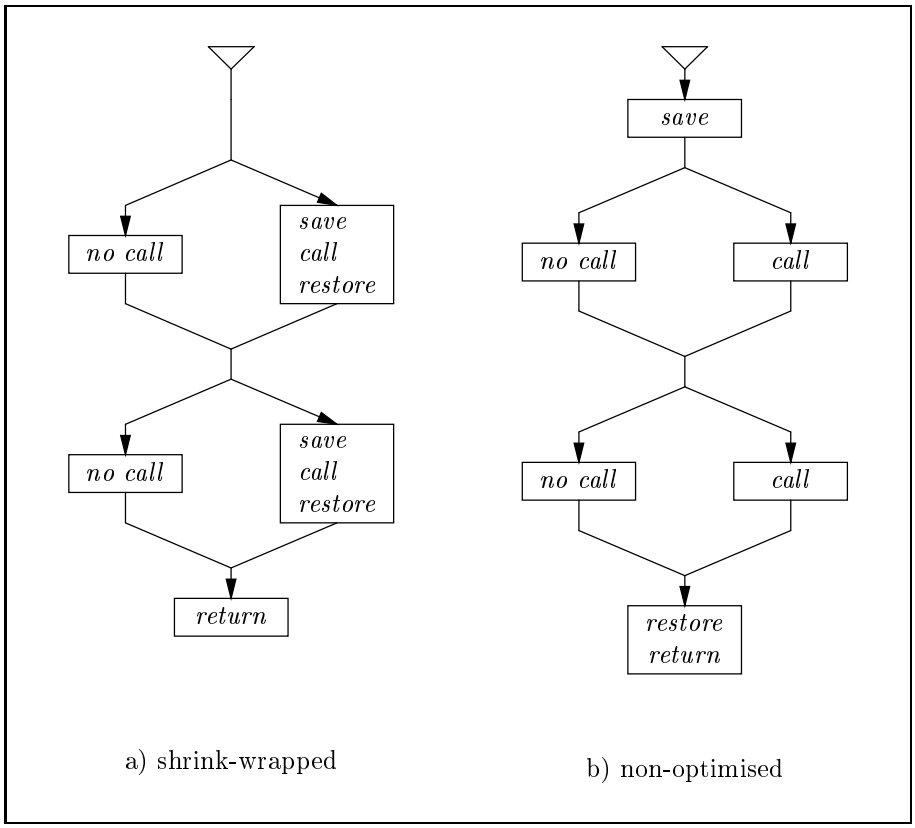

Figure 6.8: A "difficult" flow graph, with and without shrink-wrapping.

where it is not possible to determine the benefits of the optimisation without some kind of dynamic execution profiles. This is exactly the same problem observed by Chow in [Cho88]. In figure 6.8 we show two versions of the same flow graph, with and without our shrink-wrapping applied. The non-optimised code uses the standard way of saving the return address on procedure entry and restoring it on exit. There are four different ways to pass through this flow graph. In one case the optimised version wins (no function call), in one case it loses (two function calls), and in two cases there is no difference (a single function call).

6.2.4 The build phase

The *build* phase is the first phase of the important *build-colour cycle* (see figure 6.2 on page 200). The purpose of the *build* phase is to build an *interference*

graph (see the introduction, on page 11) for the entire program. This is done in two steps:

1. Use an iterative *data flow analysis* to find, for each point in the program, the set of *live variables* (live ranges) at that point. In our case this means doing an *interprocedural* (and program-wide) live-variable analysis.

2. Use the liveness information to build an interference graph. This can be done in a single pass over all intraprocedural flow graphs (each basic block is traversed backwards).

An interprocedural live variables analysis

To compute the liveness information we need to solve a slight variation of the standard global data flow equations for the *Live* attribute [ASU86, section 10.6]. These equations can be seen early in figure 6.6 on page 207. In principle, live variables analysis is a backwards data flow analysis, which means that a *use* of a register will flow backwards (upwards) in the intraprocedural flow graph, until the flow is *killed* by a definition of the register. When extending this to an interprocedural setting we must also take the flow along interprocedural edges in the ICFG (*call* and *return edges*) into account (see figure 6.3 on page 201). Unfortunately this is not as easy to handle as the intraprocedural flow.

The calling context problem

A general problem that appears when doing any kind of interprocedural data flow analysis is that different *call sites* may interfere erroneously with each other through a common callee. Sometimes this problem is called the *calling context problem* [LR91]. In our setting it manifests itself in the following way. Consider the flow graphs in figure 6.9 on the next page. Two procedures (*foo* and *baz*, of which only a part of the flow graphs are shown) both call a third procedure (*bar*). In our ICFG this will be seen as call edges going from the call sites in *foo* and *baz* to the entry of *bar*, and by return edges going from *bar* back to *foo* and *baz*.

First, recall that liveness is a *backwards analysis*, i.e., live variables will flow along the flow graph edges in the reverse direction. Now, consider the *Live* flow at point A in the *foo* procedure, i.e., directly after the call site. The flow at that point consists of registers that are live over the call to *bar*. For all these registers we need to create conflicts with all the "locals" of *bar* (remember that the whole point of this exercise is to colour these registers differently, to avoid save and restore overhead). This in turn means that the live flow at point A must be propagated along the return edge into the body of *bar*. For the *baz* procedure we have the same situation, i.e., the registers live at point B must flow along the return edge into the body of *bar*.

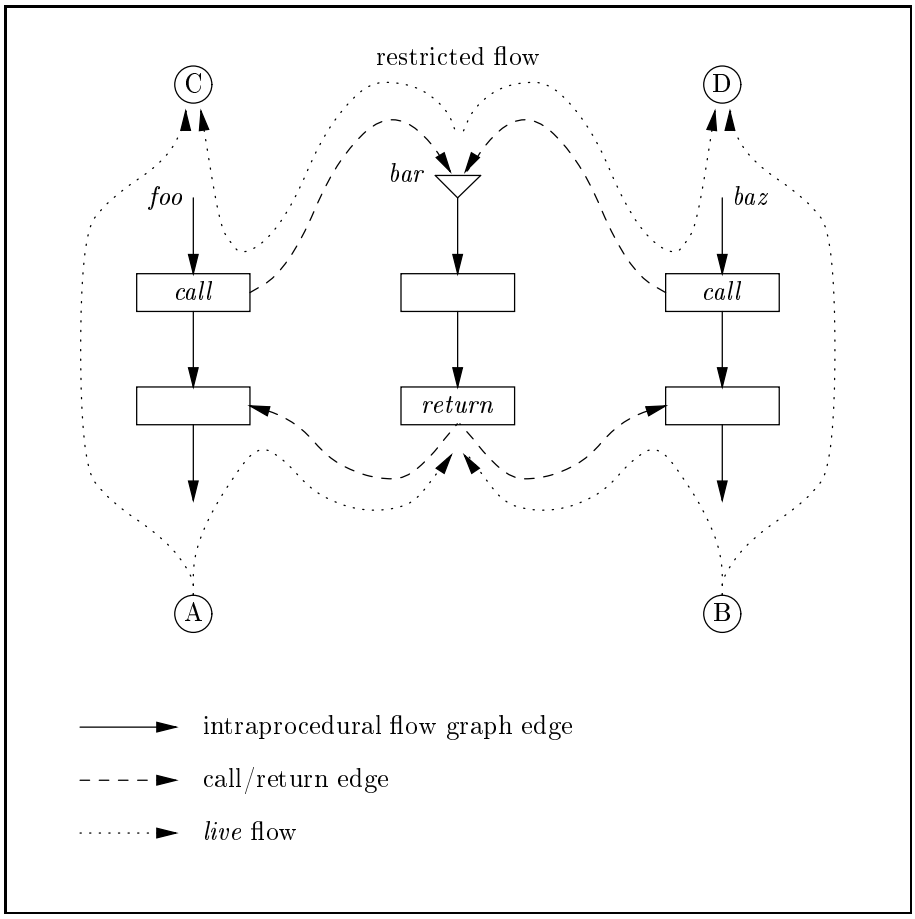

Figure 6.9: An illustration of the calling context problem.

Now to the calling context problem. Normally the flow that enters *bar* at its return point will flow through the entire body of the *bar* procedure, since the registers flowing from points A and B have no definitions in *bar*, and hence are never killed (with the exception of the return registers of *bar*). So, if we did not take special precautions the flow would continue through the call edges and enter both the *foo* and the *baz* procedures (at points C and D, respectively). But this would be unnecessarily imprecise! The local variables of *baz* are clearly not in conflict with the local variables of *foo*, so there is no point in letting live registers from *baz* flow into the body of *foo*, and vice versa.[8] This is exactly the *calling context problem*. Fortunately it is quite easy to solve in the case of interprocedural live variables, and we will do so by *restricting the flow* along *call edges*.

The flow that we want to appear at point C in *foo* is the intraprocedural flow in *foo* (i.e., the flow from point A), except for the return registers of *bar* which must be considered killed by the call. Also, we want the argument registers of *bar* to reach point C through the call edge (because they may be used inside *bar*), but nothing else! I.e., the only thing that we need to do is to restrict the (backwards) flow along call edges to the argument registers of the called function (the target of the call edge). The intraprocedural flow (from A and B) can flow inside each procedure in the normal way (along the intraprocedural flow graph edge between the call block and its successor, compare with figure 5.5 on page 184). As a result, we achieve the desired flow into *bar* (for the wanted conflicts), but avoid the unnecessary interference between *foo* and *baz*.

As mentioned above, the return registers of *bar* (which probably are live both at point A and point B, if the result of the call is used) should not be allowed to flow past the call block in either of *foo* or *baz*. However, this will be taken care of by the normal *Live* equations simply by defining the *Kill* set for the call instruction to be the set of return registers for the called procedure.

Interprocedural Live data flow equations

With all the above taken into account we arrive at the *Live* equations shown in figure 6.10 on the next page. Compared to the *Live* equations in figure 6.6 we have added an additional attribute, called *EdgeFlow*, to handle the restricted flow along call edges. In the figure, *Succ* now denotes the full interprocedural successor relation (including call and return edges), $argRegs(j)$ gives the argument registers for the procedure with entry block j, and $retRegs(f)$ gives the return registers for the procedure f.

The data flow equations are then solved by iteration. We use a *reverse depth first ordering* [ASU86, section 10.9] of the basic blocks in the ICFG to speed up the convergence. This makes a huge difference in the number of required iterations, since the ICFG becomes quite a large graph (the entire program).

[8] Of course, this assumes that *foo* and *baz* do not directly call each other, but in that case the ICFG would be different.

$$Gen(i) = \begin{cases} \emptyset, & \text{if } i \in call\_blocks \\ \text{"used before defined in block } i\text{"}, & \text{otherwise} \end{cases}$$

$$Kill(i) = \begin{cases} retRegs(f), & \text{if } i \in call\_blocks \text{ and } i \text{ calls } f \\ \text{"defined in block } i\text{"}, & \text{otherwise} \end{cases}$$

$$Live_{out}(i) = \bigcup_{j \in Succ(i)} EdgeFlow(\langle i, j \rangle)$$

$$Live_{in}(i) = Gen(i) \cup (Live_{out}(i) \setminus Kill(i))$$

$$EdgeFlow(\langle i, j \rangle) = \begin{cases} Live_{in}(i) \cap argRegs(j), & \text{if } \langle i, j \rangle \text{ is a call edge.} \\ Live_{in}(j), & otherwise \end{cases}$$

Figure 6.10: Data flow equations for interprocedural live variables.

Building the conflict graph

Once we have the *Live* data flow information it is quite easy to build the interference graph, and we do it in the standard way. Each basic block is traversed once, and backwards, keeping the *Live* set upto-date, and each instruction examined. For each instruction we add conflicts between the live ranges in *Live* at that point and the live ranges *defined* by the instruction.

When examining the instructions we are careful not to introduce conflicts for *copy instructions*, as this would stand in the way for *coalesce* (see section 6.2.5). This observation was made by Chaitin [CAC+81].

As an extra bonus we can detect instructions where none of the registers that are defined by the instruction are in the current *Live* set. This means that the instruction is effectively *dead* and can be removed.[9] As a result, we get an interprocedural *dead code elimination* during the register allocation without extra cost.

6.2.5 The coalesce phase

The purpose of the *coalesce* phase (also called *copy propagation* or *subsumption*) is to remove "unnecessary" register to register copies in the program, where unnecessary is defined to mean that the two registers (live ranges) do not *interfere*

[9] Unless it has a some side-effect of course, like writing to memory.

(in the conflict graph). If two live ranges do not interfere they can be combined into a single live range and the copy deleted. The *coalesce* phase is in some descriptions of Chaitin-style graph colouring not emphasised very much, but in our setting we consider it to be very important! The reason is that the *coalesce* phase is they key to achieving *tailor-made* calling conventions for each procedure, i.e., the use of arbitrary registers to pass function arguments and results for each function, rather than using a fixed register convention.

Coalesce has traditionally been used (in global allocators) for several purposes:

1. To remove extraneous copies inserted by a code generator or by earlier optimisations. Not having to worry about inserting unnecessary copies can make some optimisations much simpler.

2. As a means to achieve *targeting*, i.e., to compute the arguments of upcoming function calls directly into the correct registers, thus avoiding extra register "shuffling" at the call site.

3. To handle instructions with specific register demands.

Our main use of *coalesce* will be as a combination of the two first items above. However, the really interesting thing happens when coalescing is combined with the following two conditions:

- allow any virtual register to be used as argument (and result) registers for function calls, rather than to use a convention with fixed machine registers,

- do *interprocedural* graph colouring.

Interprocedural coalescing

As soon as these two conditions are added, coalescing as a technique to achieve *targeting* will be much more powerful, because it becomes independent of *procedure boundaries*. The targeting can not only occur inside a single procedure, but over several procedures at once. E.g., a value that is computed into a register at a certain point in the code can be held undisturbed in the register over a large piece of code, covering several procedures (if the register is passed as an argument between procedures), and then finally be used in some other procedure (still in the original register). The coalesce phase, together with the rest of the colouring, will select the registers to be used to pass function arguments and return values for each procedure, such that all callers of a certain function achieves as good targeting as possible.

It can not be emphasised enough how important we consider this kind of coalescing to be in the context of the GRIN back-end, since lazy functional

languages are so call-intensive.[10] In fact, we have deemed this so important as to even give the technique its own name: *interprocedural coalescing* [Boq95a].

The result of our coalescing will be live ranges that span over several procedures, i.e., live ranges that are defined and/or used in more than one procedure (recall that virtual registers used to pass procedure arguments and results are treated just like ordinary live ranges). We call this phenomenon *interprocedural live ranges*. They will slightly complicate the rest of the allocator in a way that does not arise in global (per-procedure) register allocators, and we will add a *live range splitting* phase to handle this problem, see section 6.2.10 on page 228). Note that the only way in which a live range can be "transferred" between procedures is when it is used as an argument or a result register in procedure calls. The consequence of this is that the *definitions* and *uses* of a certain live range will always be *connected* in the ICFG (interprocedural control flow graph). An example of a very simple interprocedural live range is shown in figure 6.11 on page 223, and a more complex one in figure 6.13 on page 231.

Performing coalesce

Although we claim that interprocedural coalescing can be much more effective than intraprocedural coalescing, technically there is not any major differences in how they are performed. We will perform coalescing similarly to Chaitin and Briggs: examine each copy instruction in the program in some *order* (see below), and if the two live ranges do not interfere, and if some additional *restrictions* are also met (see below), we will combine the two live ranges into one and delete the copy instruction. The interference graph is updated with the combined live range as the *union* of the two live ranges. We use a register substitution that is applied to each instruction to change already coalesced live ranges.[11]

Repeating coalesce

When the complete program have been examined, and if any coalesces were found, it is necessary to completely rebuild the interference graph (using the *build* phase), and then try another round of coalesce. The reason for this is that when coalescing two live ranges it may happen that a conflict with a third live range effectively disappears (since the copy instruction that caused the conflict is removed), but there is no way to get rid of that conflict in the interference graph during the coalesce phase. The only chance to discover new coalesce possibilities is to rebuild the interference graph from scratch.

[10] The same is probably true also for strict functional languages, as well as other call-intensive languages.

[11] Actually, when this is done we sometimes find that a copy instruction is already unnecessary, even before we have tried to coalesce the live ranges, because the substitution has turned the left and right hand sides of the copy into the same live range. This indicates that a single coalesce can make several copy instructions unnecessary.

Strictly speaking, it is not *necessary* to repeat the build phase (it is somewhat costly), but we risk losing opportunities for coalesce if we do not repeat it.

Coalesce order

The order in which coalesces are made can be significant. Not so much because of interference between live ranges, but because of other restrictions that we may put on when coalesces are allowed (see below). This means that copy instructions should be examined in some kind of "priority order" for best results. For imperative languages, typically the *inner loops* should be examined for copies early, and copies coalesced if possible.

We have not yet tried to include any priorities in our implementation, but one situation where it would probably be useful is when we have had a *tailcall* that was changed by the RISC code generator into an ordinary call followed by a number of register copies (return transfers, see figure 5.5 on page 184). For our tailcall optimisation (see section 7.4 on page 250) to succeed in re-introducing a tailcall it is necessary that all the return transfer copies have been coalesced.

Coalesce restrictions

With unrestricted coalescing, i.e., with interference as the only means to decide if a coalesce should be made or not, it can happen that a live range which is the result of a coalesce becomes so constrained (many neighbours in the interference graph) that it can not be coloured, and must be spilled.[12] Of course, this can never be good, a register to register copy will always be cheaper than a spill to memory, so it is better not to make that coalesce in the first place.

So, to avoid unnecessary spills, coalesce should be restricted in some way. Briggs propose a technique called *conservative coalescing* [Bri92]. It is based on the observation that a *necessary* (but not *sufficient*!) condition for a node to be *spilled* (during the *simplify* phase) is that the node has at least N neighbours[13] of *significant degree*. A node is defined to be of significant degree if it has at least N neighbours. Given the traditional Chaitin-style simplification heuristic (see the introduction, on page 12) it is easy to see that this is indeed a necessary condition for a spill to occur. By contradiction: if a node did not have at least N neighbours of significant degree, then there must be at least one neighbour with fewer than N neighbours. But such a node is *trivially coloured*, and can be removed from the graph. After repeating this, the node we started with must eventually end up with fewer than N neighbours itself (because the initial assumption was that it had fewer than N neighbours of significant degree, and

[12] I.e., it is possible that a graph that before coalescing is colourable, after unrestricted coalescing is no longer colourable using the same number of colours.

[13] Whenever we use the capital letter N in this chapter it will mean the number of colours available to the register allocator

all other neighbours have been removed), so the node itself is trivially coloured and will not be spilled.

Briggs does not always use conservative coalescing however, sometimes it is combined with the use of unrestricted coalescing (how exactly depends on the particular version of his allocator, see [Bri92]). George and Appel discuss the issue of restrictions on coalesce extensively [GA96a, GA96b], and also note the important fact that restricting coalesce becomes even more important in the presence of *pre-coloured* nodes in the interference graph (typically used to support parameter passing to external or previously compiled procedures). In such a situation unrestricted coalescing can in the worst case lead to *uncolourable graphs* (which can not be helped by introducing spills)! E.g., the combined live range after a coalesce can in the worst case get N pre-coloured neighbours, all with a different colour. In this case the node must be spilled, but that will not help because there is not even a temporary register available to reload it into at the points where it is used.

George and Appel claim that Briggs' unrestricted coalescing (which they call "reckless coalescing") often lead to uncolourable graphs of the above kind in contexts with many pre-coloured nodes. They also claim that Briggs' conservative coalescing if used all the time is *too restrictive*, in that it leaves too many copies behind. They instead propose a method, called *iterated coalescing* to improve this situation by interleaving a variant of the *simplify* phase inbetween uses of conservative coalescing. This way they can guarantee not to introduce uncolourable graphs while still removing many more copies than with ordinary conservative coalescing.

In the current GRIN back-end the default setting is to allow only conservative coalescing during the first run of the *build-colour cycle*, and then allow unrestricted coalescing on subsequent iterations. The reason that we do not exclusively use conservative coalescing is that we also found, as George and Appel, that it often was unnecessarily restrictive. This is confirmed by our measurements (see section 9.6 on page 288). It would be very interesting to try iterated coalescing, but we have not yet had the time to do so.

Coalesce and splitted live ranges

In addition to the above restriction on coalesce, we need one more restriction to avoid creating graphs that are more difficult to colour. Live ranges that are the result of *live range splitting* (see section 6.2.10 on page 228) can not be allowed to coalesce without restrictions. This is not really surprising, since *coalesce* and *split* does exactly the opposite of each other. If we first let the live range splitting introduce copies (in order to reduce the complexity of the interference graph and make it easier to colour), and then immediately coalesce all the copies, we would not make any progress.

On the other hand, the way our splitting works it will often introduce a large number of copies (many splits), since there is no way for the split phase

to know where in the code a single split would be most useful.[14] As a result of this, many of the copies introduced by a split can probably be coalesced again without any problems at all, so it is not a good idea to completely disallow coalesce on splitted live ranges.

What we need to balance the coalescing and the live range splitting is really a kind of "middle way" between the two. Our solution to this is to always use *conservative coalescing* (see above) for live ranges that have been part of a previous split. Since conservative coalescing guarantees not to introduce a potential spill, we will not end up in a situation where a previously splitted live range is first coalesced and then splitted again. Note that this does not preclude that a "sub-part" of a splitted live range may be spilled (inside a single procedure). The relationship between coalescing and live range splitting is further discussed in the description of the *simplify* phase (see section 6.2.7 on page 222).

6.2.6 The spill costs phase

Our *spill costs* phase is done in more or less the standard way, with two exceptions that may be a bit non-standard (both due to the fact that we use interprocedural graph colouring):

- *procedure depths*: the procedure call graph is used to calculate *weights*, to estimate the *dynamic* execution behaviour of the program (compare with inner loops),
- *split costs*: for some live ranges the cost we calculate is really a *split cost* rather than a *spill cost*, due to our combination of *live range splitting* and *spilling*.

Spill costs should be an estimate, for each live range, of the relative cost in execution time (measured in instruction cycles or some other sufficiently fine grained measure), of keeping the live range in memory rather than in a register. For a RISC machine, a rough approximation is to simply count the number of instructions where a live range occurs. In principle, the spill costs that we accumulate when examining an instruction will be a *load* for each live range *used* by the instruction, and a *store* for each live range *defined* by the instruction (to write back the result).

Procedure depths

To give realistic spill cost estimates it is also important to try to mimic the dynamic execution behaviour of the program, i.e., to estimate how many times

[14] The problem of finding a single split with good effects is really an instance of a problem that very often turns up in graph colouring algorithms, that the interference graph abstraction does not contain enough information about the "structure" of the code, and not in any way relates conflicts in the graph to instructions in the code.

a certain instruction will be executed. For imperative languages, a standard method that works well in practice is to assume that all loops are run a small constant number of iterations (typically 10). A use of a variable that occurs inside "k" nested loops (each of which do "n" iterations) will have its cost *weighted* by "n^k" compared to a use outside the loop. As a result, a variable that is used inside a deeply nested loop will be less likely to be spilled than a variable outside the loop (due to its higher spill cost), which is exactly the intended behaviour.

In our allocator we make a similar assumption, we assume that every recursive function call is made "n" times. By looking at the strongly connected components of the program call graph it is possible to estimate the call nesting using the *depth* of each procedure in the SCC call graph. The spill cost *weight* for a certain use of a live range is then calculated as "n^d" where "d" is the depth in the SCC graph of the procedure where the use occurs.

The result of this is larger spill cost estimates for live ranges used in procedures near the bottom of the call graph. It is fully in line with the observation made by Steenkiste and Hennessy in [SH89] that "programs spend most of their time in the bottom of the call graph" (for Lisp programs, but our programs behave in the same way).

The exact value of "n" is not particularly important, what is important is only that some "regions" of the procedure call graph are more weighted than other regions. The current default in the GRIN back-end is to use the number 5. In some sense it may be natural to use a slightly lower number than the constant used for traditional loop nesting (often 10), because not all functions in the call graph use recursion (of course this would be easy to take into account when calculating the weights, but that is currently not done).

Split costs

Due to the presence of *interprocedural live ranges* (see section 6.2.5 on page 215), some live ranges can not be *spilled* in the normal (intraprocedural) way, and must instead first be *split* (see the discussion in the *simplify* phase, in section 6.2.7 on page 222). This means that the spill costs estimation should really estimate a *split cost* rather than a spill cost for all live ranges that are used to pass procedure arguments and results. An interprocedural live range that is split will be so at all procedure boundaries that it "crosses". We mimic this cost by introducing a cost of "1" for each point where an interprocedural live range crosses a procedure boundary, i.e., all live ranges used as argument and result registers will get a unit cost for each function entry, each call site, and each return point that they appear in. The costs are also weighted in the normal way using the procedure depths. In addition to this, all uses and defines of interprocedural live ranges in other code "inside" procedures should be *ignored* (because it is not a cost of the split)! Instead, all intraprocedural uses of a splitted live range will show up as normal spill costs during the next iteration of the build-colour cycle (one cost for each of the newly introduced intraprocedural

live ranges that were the result of the split).

The algorithm

With the above in mind it is easy to formulate an algorithm to do the spill costs estimation. Basically, the algorithm goes through the code and accumulates costs for live ranges and finally sums up all the costs. When doing this a special *infinite* spill cost is used for live ranges that we do not want to be spilled (for example live ranges that have already been spilled). We normally use a unit cost ("1") except for when something needs to be loaded from memory. In that case we use the double cost ("2"), to indicate that load instructions are a bit more costly than other instructions.

1. Calculate the procedure depth weights using the call graph SCCs (call the resulting mapping "$weight(f)$").
2. For each function f, add a cost of $weight(f)$ to each argument and each return register of f.
3. Examine each basic block, and each instruction (let f below mean the currently examined function). Inside each basic block we keep track of a set *loaded* that are live ranges that have already been used inside that basic block. Accumulate costs according to the following rules:
   (a) *save* and *restore* instructions: no costs (they are not needed if a live range is spilled),
   (b) *spill* and *reload* instructions: add an *infinite* cost for all live ranges involved (the tiny live ranges that are the result of previous spills should not be spilled again),
   (c) *call* instructions: add a cost of $weight(f)$ to each argument and each return register of the called function,
   (d) *return instructions*: no cost (handled by 2 above),
   (e) other instructions: find live ranges *used* and *defined* by the instruction, not counting interprocedural live ranges. Then add costs according to:
      - $2 * weight(f)$ for all live ranges in *used* that are not in *loaded*,
      - $weight(f)$ for all live ranges in *defined*.

      After this, add all *used* and *defined* live ranges to the *loaded* set.[15]
4. Summarise (for every live range) the costs accumulated in 2 and 3, making sure that *infinite* costs are handled properly.

[15] A variable should not be allowed to be *loaded* for a very long time, see the discussion in section 6.2.9 on page 226.

Note that both the two special cases with *save/restore* and *spill/reload* instructions becomes very easy to identify due to the use of special RISC instructions (compare this to if ordinary *store* and *load* instructions had been used instead).

6.2.7 The simplify phase

The overall style of our register allocator is a variant of Briggs' *optimistic colouring* [BCKT89] (see figure 6.2 on page 200), and our *simplify* phase is more or less identical to the one used by Briggs. We do the following:

1. Repeatedly remove "trivially coloured" nodes (with fewer than N neighbours, see the introduction on page 11). Save all removed nodes on a special *stack*.

2. When no more nodes can be removed in 1, choose a *spill candidate* (in our case it can also be a *split candidate*) according to some *spill choice heuristic* (which in turn uses the previously calculated *spill cost estimates*). Remove the candidate node from the graph, push it on the stack and continue.

3. Go back to 1, and repeat until the graph is empty. The stack is returned as the final simplify result (it will be passed on to the *select* phase).

The result of the simplify phase is the special stack, containing both spill (and split) candidates and nodes that are trivially coloured. This stack is passed on to the *select* phase (see section 6.2.8 on page 225). The only major difference between ours and Briggs' simplify phase is that instead of always choosing a *spill candidate* when simplify blocks, we will choose a "spill or split" candidate. However, this "choice" between spill or split is not really a choice, it is fully determined by the type of live range chosen as a candidate. If it is an interprocedural live range it will be *split*, otherwise it will be *spilled*.

Interprocedural live ranges

As mentioned above we will not *spill* a live range that spans several procedures, at least not in the normal sense. to understand why, consider the example in figure 6.11 on the facing page.

Figure 6.11 shows (part of) three intraprocedural flow graphs illustrating an interprocedural live range. The live range "v_1" is used to pass an argument to the *bar* function, i.e., "v_1" will be defined in the two callers, *foo* and *baz*, and it ranges into the body of *bar* where it is used. Now suppose that we need to spill, and that "v_1" seems like the best choice according to our *spill choice heuristic* (see below). The question then is where we can spill "v_1"? The normal place to spill is to the activation record of the current procedure, but in figure 6.11 there is no single "current" procedure during the lifetime of "v_1". Moreover, there is no (easy) way for *baz* to tell if its caller is *foo* or *baz*. One possible solution

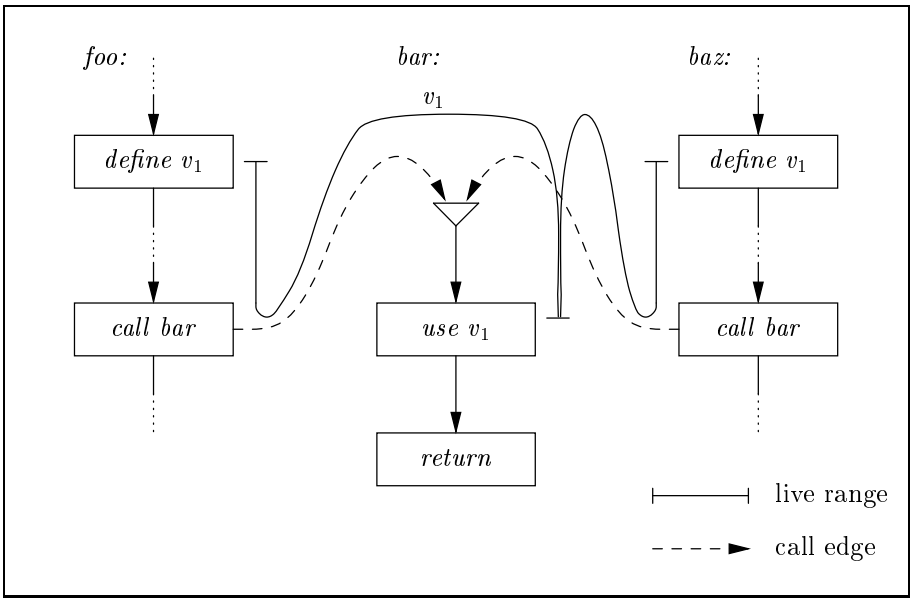

Figure 6.11: An example of an interprocedural live range.

would be to force both callers (*foo* and *baz*) to spill "v_1" to exactly the same "place", and then let *bar* reference the spilled live range in the stack frame of the calling function (the parent). However, this would very quickly lead to a difficult program-wide optimisation problem, since all callers and callees must agree on the locations of spilled variables. Although this may be possible, we have chosen an alternative and much simpler method, that fits well into our normal technique of spilling only to the stack frame of the current procedure.

Restricted splitting

Our solution to this problem is to completely avoid the problem of spilling across procedure boundaries, by saying that interprocedural live ranges are never spilled. Instead they will be *split*, i.e., divided into a number of shorter live ranges with copy instructions inbetween. They key insight to understand this is to note 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 interprocedural *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 will *split* it on both sides of all procedure boundaries that it crosses, i.e., we

insert copy instructions and undo some of the work done by *coalesce*. In this way, the splitting (for details, see section 6.2.10 on page 228) transforms a long interprocedural live range into a number of short live ranges. These new live ranges can be of two kinds:

1. completely intraprocedural, one for each procedure where the original live range appeared,
2. for call sites where the original live range was used as an argument register we create a new live range that is used only at that call (it is only used to "transfer" the value between the two procedures). Similarly for function result registers.

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. As an example, consider the live range in figure 6.11 on the preceding page. Figure 6.12 on page 229 shows how it would be split.

In some sense this combination of splitting and spilling 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 of standard Chaitin-style colouring and it becomes even more apparent in our interprocedural setting where live ranges can become very long. Splitting at procedure boundaries is an effective way to eliminate this problem. We call the technique *restricted splitting* to contrast it with Briggs' *aggressive live range splitting* (see the discussion in section 6.4.2 on page 239).

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 (where it is little used and the register pressure is high). This is achieved by a combination of the use of normal virtual registers to pass function arguments and results (a non-fixed calling convention), the interprocedural coalescing, the interprocedural splitting and the normal intraprocedural spilling. The coalesce phase will often create a number of very long live ranges, which passes procedure boundaries as arguments or return registers. If they are hard to colour, they will be split. After that, some (intraprocedural) parts of the split live ranges can be spilled in the normal way, during the next iteration of the build-colour cycle.

An example of how a complex interprocedural live range is split is shown in the section describing the *split* phase, see figure 6.13 on page 231.

Spill choice heuristics

The purpose of a *spill choice heuristic* is to choose a *spill candidate* at the point when simplify blocks, i.e., when all remaining nodes in the interference

graph have at least N neighbours. Normally the heuristic used is some kind of *combination* of *cost* and *benefit*. I.e., we should prefer a node which is cheap to spill (has a low spill cost estimate), but which is also beneficial to spill (reduces the complexity of the interference graph when it is removed). The latter is normally measured as the *degree* of the node (its number of neighbours).

The heuristic that we use currently is very simple, we choose the node "n" which has the lowest value of:

$$\frac{spill\_cost(n)}{degree(n)}$$

The intent is that if we remove a node with a high degree then the interference graph will become much simpler.

Note that a candidate is chosen solely due to this heuristic. It is not until after a choice has been made that we can see if it becomes a split or a spill candidate (depending on if the live range is interprocedural or not). We never pre-commit to do a spill or a split in a certain situation.

The way the spill costs are estimated for intraprocedural and interprocedural live ranges (see section 6.2.6 on page 219) and the actual spill/split choice heuristic used by simplify will all interact in a complex, but very important way. In some sense, this is what decides if the allocator should prefer to spill or to split. We have not yet studied this issue in detail, but from our experiences sofar it seems that the methods we currently use work quite well. Often, splits will be preferred in the beginning of allocation, since the very long interprocedural live ranges created by coalesce can get a very high degree, and be hard to colour.[16] When the longest live ranges have been eliminated, the allocator will prefer normal spilling during the rest of the allocation. The very long live ranges is prevented from reappearing by the restrictions that we put on coalesce for live ranges that are the results of previous splits.

6.2.8 The select phase

The purpose of the *select* phase is to assign colours to the nodes on the node *stack* returned by the *simplify* phase described in the previous section (recall that this stack contains both "trivial nodes" as well as spill or split candidates). The colouring is done in reverse to the order in which simplify removed nodes from the graph, i.e., the node removed last by simplify is coloured first. Each node is given a colour that is free for the node, i.e., not assigned to any of its (coloured) neighbours. Since we are using *optimistic graph colouring* [BCKT89], the *select* phase is not guaranteed to succeed. For nodes that were found to be "trivially coloured" by simplify, the select phase will always find an unused colour (since such nodes always have fewer than N neighbours). For spill or

[16] This assumes that unrestricted coalescing is used, see the discussion in section 6.2.5 on page 217.

split candidates, it is possible that a colour can not be found (they have always at least N neighbours). However, in practice it is often the case that select is able to colour also spill or split candidates (that is the whole point of optimistic colouring). This can happen if some of the neighbours of a node receives the same colour (by "accident"), which means that having N or more neighbours is not a sufficient condition for a spill (or split) to occur.

If select fails to find a colour the node will be marked as a *definite spill* (or *split* if it is an interprocedural live range). The select phase continues until all nodes on the simplify stack have been either coloured or marked as definite spills/splits. If all nodes could be coloured, the graph colouring has succeeded and we exit the build-colour cycle (see figure 6.2 on page 200), otherwise we say that select *failed*.

Select failure

In normal optimistic colouring, a select failure would mean that we had found definite spills, and we would simply continue to the *spill* phase, to insert spill code, and then start a new iteration of the build-colour cycle. In our case it is a bit more complicated. If select fails it can be for one of three reasons:

1. either we found spills, but no splits,
2. or, we found splits, but no spills,
3. or, we found both spills and splits.

The first two are easy, with only spills we insert spill code (the *spill* phase, see section 6.2.9), and with only splits we insert split code (the *split* phase, see section 6.2.10 on page 228). However, with both spills and splits we must make a choice, either we honour only the spills, only the splits, or both.

Our choice is to honour only the splits, and to "throw away" all spills. The motivation for this is as follows. If a live range must be split, it is probably a "very long" live range, maybe ranging over several procedures. If we split such a live range, at all procedure boundaries that it crosses, it is likely to affect the entire interference graph, in a non-trivial way. Possibly, the spills that had been chosen together with the splits may no longer be necessary. Rather than risk producing unnecessarily pessimistic code we will throw away all the chosen spills. If they really need to be spilled, even after the splitting, that will be done by the general machinery during the next iteration of the build-colour cycle.

6.2.9 The spill phase

Our *spill* phase is completely standard, it traverses the program and transforms spilled live ranges from being kept in registers to being kept in memory. This is done by creating *home locations* in the stack frame for all spilled live ranges

(recall that only intraprocedural live ranges can be spilled). Basically, for each *occurrence* in the code of a spilled live range we do the following:

- For each instruction where a spilled live range is *used* we insert a load from the corresponding home location immediately *before* the instruction. We always load the value into a fresh virtual register, which is then used in the instruction instead of the original live range.
- For each instruction where a spilled live range is *defined* we insert a store to the home location immediately *after* the instruction. Also in this case we create a fresh virtual register and use that instead of the original live range.

Effectively, this will replace a spilled live range with a number of different, and very short, live ranges. The original live range will completely disappear from the code. Since all the new live ranges are independent of each other, they can each be given a different colour and will hopefully constrain the interference graph less than the original live range.

An optimisation

We use a very simple local optimisation to the above scheme, that is only done inside each *basic block*. A live range that has more than one use inside a basic block will not be reloaded twice if the uses are sufficiently *close*. It is possible to have more or less complicated definitions of *close* in this case, but we use Chaitin's original definition, i.e., two uses of a live range are close if no other live range goes dead (has its *last use*) inbetween. The intent of this is as an approximation that "nothing interesting" happens to the register pressure between two uses that are close.

Spill instructions

Similarly to when inserting RISC *save* and *restore* instructions around potentially recursive function calls (see the *save locals* phase, section 6.2.3 on page 202), we use special RISC instructions for spills, called *spill* and *reload*. The intention is to make the implementation simpler by not having to worry about concrete stack slots inside the allocator (the build-colour cycle).

Each *spill* or *reload* instruction takes as argument a list of *register pairs*. An example of a spill instruction that spills a single register is:

$$\mathit{spill} \quad [(v_5, v_2)]$$

This spills the register "v_5", but to the place in the stack frame that is the home location of "v_2". The important point is that for each spilled live range, all the new (very short) live ranges introduced by the spill code insertion can spill to

the same home location. In the above example "v_2" would be the original live range and "v_5" one of the short live ranges introduced by the spill code.

It can also be mentioned that live ranges that are spilled need not be saved and restored around potentially recursive function calls, so any *save* or *restore* instructions for spilled live ranges found when inserting spill code can simply be deleted.

6.2.10 The split phase

As discussed earlier, the intent of the *split* phase is to implement our *restricted splitting* (see the discussion in section 6.2.7 on page 223). It will traverse the program and introduce copy instructions at function boundaries (both *entry* and *exit* points) and around all call sites (both before and after the call), for live ranges that should be split. An example of this can be seen in figure 6.12 on the next page, which shows the result after splitting the live range "v_1" in figure 6.11 on page 223. We have introduced four new live ranges, one to be used as a new argument register ("v_2"), and three new live ranges to be used inside a single procedure. The original live range, "v_1", is no longer used.

Note that in principle it is possible that there are several "independent" occurrences of a live range inside the same procedure. If we return to figure 6.11 on page 223, and instead assume that the two calls to *bar* were done from the same function, but from two different call sites in that function. After splitting, in figure 6.12, this would result in the two live ranges "v_3" and "v_5" occurring in the same procedure. But these two are now completely independent, and need not be coloured to the same registers!

Unfortunately, it is not always trivial to detect this situation, i.e., whether the intraprocedural live ranges that appear after splitting are really independent or not. E.g., as yet another variation of the above example it is possible to imagine a single definition of a live range that is routed to two different call sites (in the previous example we had two separate definitions that were each routed to a call site). But in this case, the split live ranges must really be the same (because there is only one instruction defining the live range).

In general, for best results we would need to have some kind of *renumber* phase (to use Chaitin's terminology), to identify intraprocedural *def-use chains* after splitting. Note that we previously claimed that our allocator would not need a renumber phase due to the static single assignment property of the GRIN code (see section 6.2.1 on page 199), but obviously that can no longer be true if we require the splitting to be done in an optimal way. However, we have not yet implemented a renumbering phase, so we can not use it here. Instead we use a simple, but safe, method to insert splits inside each procedure.

Our current solution works as follows: for each live range to be split, and for each procedure, we use a single virtual register to represent all the split live ranges inside that procedure. This can sometimes be non-optimal, two live ranges that are really independent but occur in the same procedure will now be

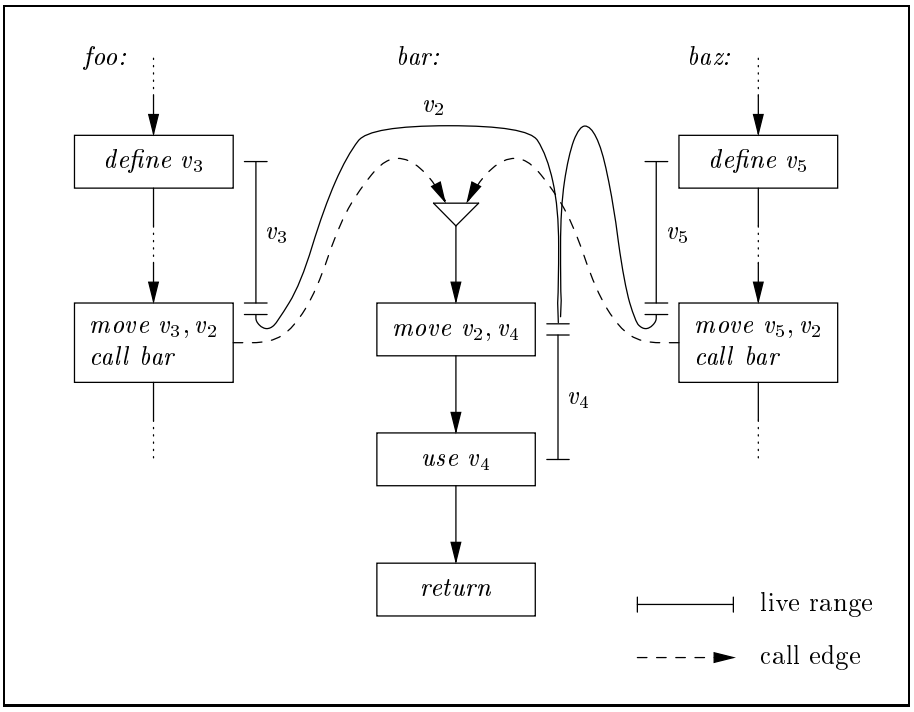

Figure 6.12: An example of splitting.

forced to the same colour, since the same virtual register is used to represent the two.

Note that a similar problem arises on the "interprocedural level" (the above discussion concerned only virtual registers used inside procedures, not the actual registers used to pass arguments and results). This happens when a live range span more than two procedures, i.e., the same virtual register is used to pass arguments to more than one function (a value that is an "incoming argument" in one function is passed on unchanged directly to a new function call). If we during splitting were to reuse the original live range for argument passing to all functions where it was originally used, we would run into the same problem as above, that independent live ranges were forced to the same colour because they were represented by the same virtual register. However, this problem is much simpler to avoid by always creating new argument and result registers for each function (that used the live range we are splitting).

Summary

To summarise the discussion above, our splitting algorithm does the following. Each live range to be split is replaced by a set of new virtual registers:

- one new register for each occurrence as an argument register,
- one new register for each occurrence as a result register,
- one new register for each procedure that the live range occurs in.

To "connect" all these new registers we insert copies, between the new argument/result registers and the new intraprocedural registers. This is done in exactly the same way as during the original RISC code generation for function calls (see section 5.3.2 on page 183, and especially figure 5.5 on page 184).

Figure 6.13 on the next page tries to illustrate this. The figure shows two functions (as boxes), *foo* and *bar*, before and after splitting. In the left part of the figure, before splitting, a single live range is used as an argument register for both functions. This single live range is shown in the figure with its "end points" marked (these are either last uses or definitions of the live range). The *foo* function calls *bar* from two different call sites, and *bar* calls *foo* once. Note how one of the calls inside *foo* simply passes on the incoming argument of *foo* directly on to *bar* (often called an *invariant* argument). A similar thing happens in *bar*. Live ranges like this may seem very artificial, but are in fact quite common in our setting, due to the use of *interprocedural coalescing* (see section 6.2.5 on page 214).

After splitting this live range we would get the result shown in the right part of the figure. After the split we get four different virtual registers, one inside each procedure, and one as an argument register for each of the procedures. Note that inside the *foo* procedure, we are forced to use the same virtual register for two live ranges that really are independent (this is the approximation discussed above).

Note also that the two functions use different argument registers after splitting, they used the same before splitting.

Split and coalesce

Some of the inserted copies in figure 6.13 will probably be coalesced during the next iteration of the build-colour cycle, but not all. This is guaranteed by the fact that coalesce always uses *conservative coalescing* for live ranges that are the results of live range splitting, so it will never create a combined live range that risk getting split again. See also the discussion in section 6.2.5 on page 218.

6.2.11 The post process phase

The *post process* phase is very simple. Its purpose is to do some final adjustments to the code, after the actual colouring (the build-colour cycle) is done. These

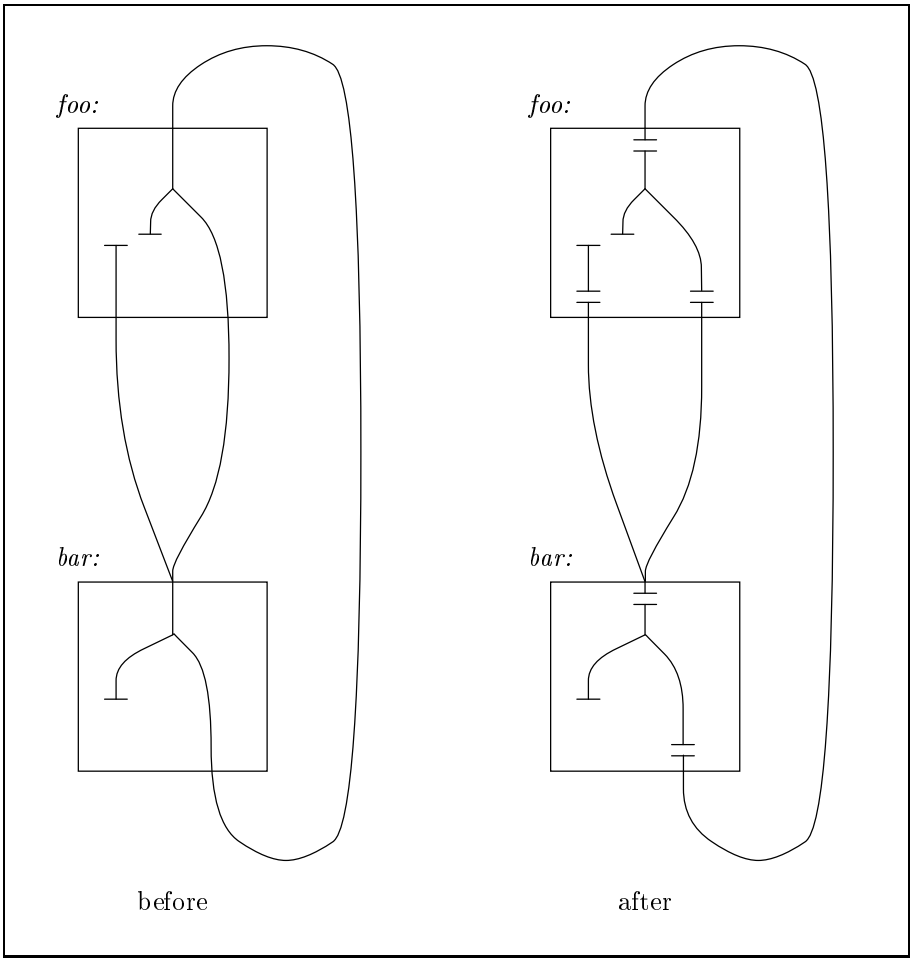

Figure 6.13: Splitting a complicated live range.

are:

- substitute real machine registers for all virtual registers in the code,
- assign stack slots (home locations) and eliminate all the special RISC stack instructions.

The result of a successful run of the *select* phase (see section 6.2.8 on page 225) is a complete mapping of virtual registers into real machine registers (the colours). We use this as a substitution and simply apply it to all instructions in the program. The register mapping is complete, i.e., all virtual registers appearing in the program have been assigned a colour, so after applying it all virtual registers will be eliminated.

To assign stack slots we need to filter out all instructions that use stack memory, for each procedure. But this is really simple, since the only such instructions are the special RISC stack instructions: *save* and *restore* (to save and restore locals around potentially recursive function calls, see the *save locals* phase, section 6.2.3 on page 202), and *spill* and *reload* (used for normal spills during the colouring, see section 6.2.9 on page 226).

Since all *spill* and *reload* instructions include the original live range that was spilled, it is trivial to determine how many stack slots are needed and then assign one to each spilled/saved local.

In addition to assigning stack slots, we also insert instructions to allocate and deallocate a stack frame, at procedure entry and exit, respectively. These instructions are initially inserted in a naïve way, but will later be optimised using a *shrink-wrapping* optimisation similar to what we used to save and restore locals around function calls (see section 6.2.3 on page 202). This optimisation of stack frame allocation is described in section 7.5 on page 254.

After the post process phase, we have code that is "runnable", for the first time during the compilation process. Or rather, the RISC code that we have at this point can be trivially "pretty printed" as SPARC assembler, which can then be run.

6.3 The extended algorithm

The *extended* register allocation algorithm is a proposal for a more practically useful register allocation algorithm than our *base* algorithm.

6.3.1 Motivation

The extended algorithm attacks the following two problems with the base algorithm:

- *Scalability*: since the base algorithm builds an interference graph for the entire program, the graph becomes very big for large programs. Graph colouring as a technique may be non-practical for very large problem sizes.
- *Flexibility*: the base algorithm can not handle what we call "unknown" procedure calls (or really, calls where either the *caller* or the *callee* is unknown at compile time). It requires all function calls in the program to be resolved (which is done using the program-wide control flow analysis).

The first problem has, to our knowledge, not been studied very carefully (possibly because conventional wisdom see the whole idea of program-wide graph colouring as futile). The problem is mentioned in [Ste91], but after a short discussion of problem size the whole idea of program-wide colouring is deemed computationally impractical.

From the status of our current implementation it is hard to draw any definite conclusions. It is true that our implementation of the base algorithm can not handle very large programs, but we believe that to be due to an overall inefficient implementation, rather than the intractability of graph colouring. The current implementation has never been meant as anything else than an experimental testbed, so it is not surprising that it is not very efficient. E.g., it is written in pure Haskell, using arrays without update-in-place (or any other side-effects for that matter), something which is really needed for efficient implementation of graph colouring (at least using the traditional algorithms).

In theory it is hard to give a complexity measure for graph colouring register allocation, since a complete allocator necessarily consists of many different parts and use complex data structures that interact in non-obvious ways. A worst case complexity would also not be very meaningful, e.g., the number of iterations through the build-colour cycle would probably in worst case be the same as the number of nodes in the graph, but in practice it is much better (according to earlier work two or three iterations are almost always enough). With a careful implementation it should be possible to do the actual colouring, i.e., one run of the *simplify* phase, in near linear time (in the number of nodes in the graph). To this should be added the extra cost if spills have to be chosen. Experimental results reported by Briggs in [Bri92] indicate that his entire register allocator in practice has a complexity of $O(n \log n)$, where n is the number of nodes in the

graph. This does not sound too bad, but it is still possible that it will not work for really large programs.

The second problem above, called *flexibility*, is more of the practical kind, when using the algorithm in a real compiler. Even without allowing traditional *separate compilation* it is in practice more or less necessary to be able to handle some level of "uncertainty" in the caller/callee situation. In the base algorithm, no uncertainty at all is allowed when we build the ICFG (see figure 6.3 on page 201). Of course, the whole ICFG is an *approximation* of what function calls that can take place during a run of the program, but it is not that sense of uncertainty that we mean here. Instead, we mean the fact that the current ICFG must contain only *call edges* with a single known target procedure, there is no way to encode a completely unknown call (an *indirect* call).

As an example consider the control flow analysis (see chapter 3), done on the GRIN level. The analysis result is used to specialise and inline calls to *eval* (see figure 4.2 on page 85). For situations where an inlined *eval* would result in a case expression with a very large number of alternatives, it might be profitable to keep only a small number of the alternatives, and catch all the rest with a *default alternative* (recall that all the right hand sides are very similar). But doing so would result in lost flow information, and from the point of view of the interprocedural register allocation would give a situation with both unknown callers and callees. We might also want to approximate inside the analysis itself, and as a result keep some of the unknown control flow (calls to a general *eval* procedure).

To handle situations like the above, we really need to be able to do a function call using some standard calling convention (and to a target that is unknown at compile time). Compare this to the RISC "*ecall*" instruction which is used to do *external* calls, "out of" the GRIN program. For full inter-operability with code written in other languages we must also be able to "call in" to the GRIN program, something which currently is not possible.

6.3.2 The algorithm

We will now show how the above problems can be solved using a combination of the base algorithm and techniques from *bottom-up* interprocedural register allocators (see the introduction, on page 14). To summarise:

1. We keep the overall program-wide compilation environment of the back-end, but we allow some amount of "uncertainty", i.e., allow approximations done on the GRIN level resulting in unknown function calls. The starting point of the allocation is the procedure *call graph*, after control flow analysis and *eval* inlining.

2. The procedures in the call graph are *partitioned* into *clusters* of connected procedures. The size of a cluster can be chosen freely, as long as all the clusters taken together cover the call graph and no procedure belongs to

more than one cluster. In general however, larger clusters are expected to give better allocation results, but also result in longer compilation times.

3. The base algorithm, with some very small modifications, is applied to each cluster, in a bottom-up traversal of the clusterised call graph. Standard bottom-up techniques are used to avoid clobbering registers used by descendant clusters. Completely unknown calls (out of the cluster) are handled using a standard calling convention.

Identifying clusters

The point of the extended algorithm is to combine the advantages of traditional bottom-up colouring with the needs of lazy functional programs. As we have advocated earlier this means doing graph colouring on several procedures at once (the clusters), to catch all the "loops" of the program. To get best results it is advised to make sure that all SCCs (strongly connected components) of the call graph are kept intact, i.e., that all procedures in an SCC are put in the same cluster. The reason for this is that the SCCs are where the potential loops can be found. As an example, recall the call graph for the *queens* program, after *eval* inlining (see figure 3.9 on page 74). In figure 6.14 on the following page we show a possible partitioning of the same call graph into clusters.

A simple algorithm for identifying clusters is to start with any of the *leaf* procedures, and then add procedures upwards in the call graph. Whenever a procedure is added, all it descendants that are not yet assigned to a cluster should also be added (and possibly all procedures in the same SCC). This is repeated until some predefined size (code size or number of procedures) is reached for the current cluster. Then we start with a new procedure (either a leaf procedure or an immediate ancestor to some already finished cluster) and do the same exercise again, until all procedures are divided into clusters. Note that it is not a catastrophe if we are forced to break an SCC while doing this (see below).

Cluster colouring

Once the clusters are fixed we apply the base algorithm to each cluster, in a bottom-up ordering of the clusters. When doing so we make sure that no live data is kept in registers used by descendant clusters when doing a call to a known descendant procedure. This is trivially achieved by simply marking such registers as *clobbered* by call instructions to descendants. All that machinery is already present in the base algorithm (used for external calls). Note that this does not necessarily mean that we completely *avoid* using registers used by descendants, it is only over calls to descendants that we must not use them.

When colouring a cluster we also have a choice, either we give the cluster all the registers it needs, or we put a limit on the number of registers that each cluster is allowed to use. This is really the standard problem of bottom-up

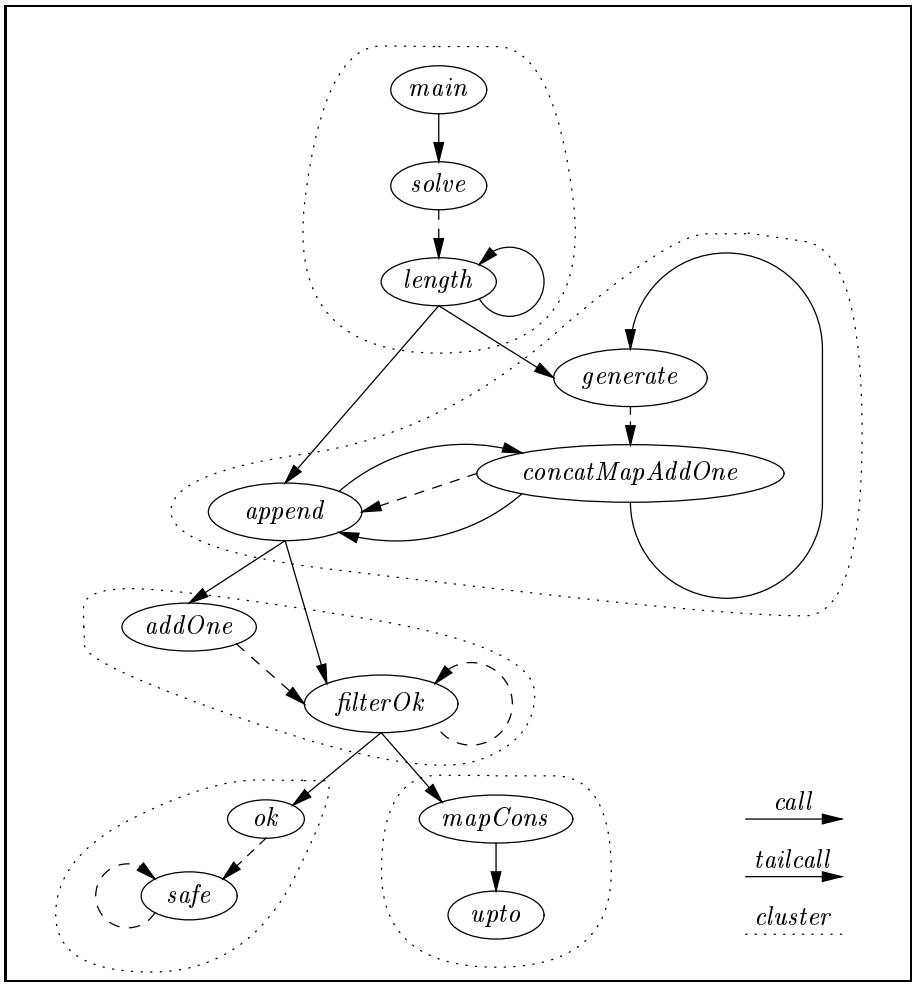

Figure 6.14: An example cluster covering for the *queens* program.

allocators, how to handle the situation when the algorithm runs out of registers in the higher regions of the call graph (see discussion in [Cho88] and [SH89]). We are still a bit uncertain how this is best handled, experimentation is needed to find a good compromise.

Function calls inside a cluster are handled completely by the base algorithm. This means that our interprocedural coalescing still works, and also that the save locals phase will detect SCCs inside a cluster and insert save and restore instructions around such calls. Function calls outside of the cluster, but to known procedures in descendant clusters (already coloured) are also simple, and no registers need to be saved or restored around such calls. If the clusters were chosen never to split an SCC we can always find a bottom-up traversal so that a procedure in an ancestor cluster is never called. But even if we split an SCC (because it was very big) we can still handle calls to ancestor clusters by saving all registers currently in use similarly to in other bottom-up allocators, and use a standard calling convention. This also means that we get a natural way to handle totally unknown calls, e.g., to external procedures or because of approximations on the GRIN level. This is basically the same technique as Chow's *open* and *closed* procedures [Cho88], but ours become more complex since we have both procedures and clusters.

Discussion

Naturally, how well the cluster partitioning works depends on the structure of the call graph. As an example, consider again the *queens* program in figure 6.14 on the preceding page. Before the control flow analysis and *eval* inlining the same program had the call graph shown in figure 3.8 on page 73, which is much more "connected". When faced with a call graph with a large amount of connectivity, it will of course be much more difficult to make the partition into clusters without breaking SCCs. Hopefully though, call graphs after the eval inlining will not be that bad even for very large programs (see also section 9.4 on page 284 in the benchmarks chapter, which discusses the related issue of the number of tags in unfolded calls to *eval*). If needed, there are several ways to reduce the "connectivity" of the call graph, e.g., function specialisation and cloning (see section 10.2.1 on page 296).

It is in fact possible that the extended algorithm can produce even better results than the base algorithm in some situations. To see why, consider a program with a very big procedure at the bottom of the call graph that is very seldomly called, e.g., some kind of "error" procedure, or the garbage collector if that had been visible in the call graph. This is a standard problem for bottom-up allocators, because such a procedure may require lots of registers to colour, and will thus affect all its ancestors negatively. The same will of course be true for our base algorithm. However, in a bottom-up algorithm it is quite easy to get rid of the problem (if we can only identify problematic procedures). The solution is simply to use the standard calling convention when calling the

problematic procedure, so that it consumes no registers for its ancestors. This method is easy to incorporate into our extended algorithm, but would be more problematic to use in the base algorithm.

To help in situations like the above, the use of *profiling* information in identifying clusters would also be a big help.

6.4 Related work

This section will discuss related work in the area of register allocation. The presentation here follows the same structure as the register allocation overview in the introduction (see section 1.3 on page 10) where we defined the three categories *local*, *global* and *interprocedural* register allocation. We will first discuss register allocation in general terms, and then register allocation for functional languages.

6.4.1 Local register allocation

As discussed in the introduction, local register allocation is normally made at the same time as code generation, using some very simple technique to select registers for the generated code. A common example is the method outlined in the introduction, called *information retention* or *register tracking* [ASU86, FH92].

Some advanced code generation techniques could also qualify as a kind of local register allocation, methods that try to minimise the number of registers needed to compute the value of an expression [SU70, AJ76, AGT89, WW90].

6.4.2 Global register allocation

The basis of global register allocation was also discussed in the introduction. In particular, we introduced *graph colouring*, the technique that we are mainly interested in, and described Chaitin's original graph colouring algorithm.

Chaitin-style colouring

Briggs *et al.* have made several extensions to Chaitin's original algorithm. A way of delaying spill decisions, called *optimistic colouring*, is introduced in [BCKT89, BCT94]. It will never produce worse colourings than Chaitin's original heuristics, but sometimes better (see also the description of our *select* phase, in section 6.2.8). Furthermore, they introduce *rematerialisation* [BCT92], a technique to reduce the spill cost for values that are cheap to recompute and therefore should not be spilled in the normal sense. A way to allocate registers for multi-register values, called *colouring register pairs* is described in [Bri92]. This technique handles the architectural constraints that often arise when handling multi-register

values, e.g., that two registers must be consecutive and evenly aligned. A similar technique was independently developed by Nickerson [Nic90].

George and Appel attack the problem that Briggs' method to restrict the coalesce phase, called *conservative coalescing*, is often too restrictive. Their compiler inserts a lot of copy instructions for reasons similar to our use of the coalesce phase, so effective coalescing becomes really important. Their proposed solution, called *Iterated Register Coalescing* [GA96a, GA96b], is to interleave the normal colouring (the simplify phase) with the coalescing, and thereby allowing more coalesces to be made without producing uncolourable graphs. See also the discussion in section 6.2.5 on page 217.

Bernstein *et al.* [BGM+89] experiment with different spill choice metrics. They find that a "best-of-three" measure works best in practice, i.e., produces the fastest code. This is quite natural since when dealing with NP-complete problems there will always be situations where one single heuristic will give far from optimal results.

Priority based colouring

An alternative graph colouring method, called *priority based colouring*, have been presented by Chow and Hennessy [CH84, CH90]. Here, instead of using spilling as the main method to reduce the complexity of the interference graph, they use *live range splitting*. By inserting register-to-register copy instructions, i.e., *splits*, the chromatic number of the graph can be reduced. However, to do this they have to sacrifice many of the desirable features of Chaitin's allocator. In order to make live range splitting not too costly, the interference graph is built with greater coarseness (it becomes less precise). Whereas in Chaitin's original framework, interferences are collected for each instruction, Chow and Hennessy looks only at the basic block level. This means that two live ranges that occur in the same basic block will always be in conflict, making *coalescing* impossible. However, priority based colouring can sometimes give better results than Chaitin-style spilling, and it can also be cheaper (in compile time) depending on how costly split heuristics that is used. Another difference is that Chaitin originally assumes that all variables are kept in registers, and later spills some to memory. Chow, instead, initially assumes all variables to be in memory (*home locations*), and then allocates the most beneficial ones to registers. A more in-depth comparison of the two methods is given in [Bri92]. Chow implemented priority based colouring in the context of the MIPS R2000 compiler system [CHKW86]. Larus and Hilfinger successfully adopted priority based colouring in the SPUR Lisp compiler [LH86].

Aggressive vs. restricted splitting

A method to combine the live range splitting used in priority based colouring and a Chaitin-style colouring framework have been proposed by Briggs [Bri92].

It is not possible to simply add splitting to a Chaitin-style framework, mainly because the cost of keeping the interference graph upto-date as live ranges are split is considered too high. The reason for this is that the interference graph normally contains no information about "where from" different interferences were collected, so when splitting a live range, we have to completely rebuild the interference graph to get it updated. Briggs handles this by aggressively splitting live ranges before colouring is started, in contrast to the traditional approach, to use splitting as a way of reducing the complexity of the interference graph when the colouring algorithm is blocked. The splitting is done at certain strategic points, e.g., 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, e.g., the coalesce phase 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 *biased colouring* that tries to give related live ranges (originating from the same split) the same colour, so that the copy instruction can be deleted.

As an alternative to aggressive live range splitting, in our interprocedural framework, we introduced a *restricted* form of splitting (see section 6.2.7 on page 223). The advantage of our solution compared to Briggs' is exactly that it is *restricted*, it can only happen to interprocedural live ranges. Because of that it is quite cheap to simply add it to the allocator in much the same way as spilling is handled (see figure 6.2 on page 200). When the *simplify* phase finds that one or more interprocedural live ranges would be the most beneficial to "spill" (according to the calculated spill costs), it will simply mark it as a *split candidate*. If the *select* phase later finds that a split candidate can not be coloured it will continue to the split code insertion. All this is completely analogous to the way spills are handled. Experiments suggest that simplify often finds several live ranges to split in "one go", even further making the cost of splitting feasible.

Other methods

Gupta, Soffa and Steele [GSS89] device an efficient algorithm for graph colouring where the complete interference graph need not be built. *Clique separators* are used to identify subparts of the interference graph that can be independently built and coloured.

Callahan and Koblenz [CK91] concentrate on the problem that when using the interference graph abstraction, most information about the program flow structure and local reference patterns is lost. They try to remedy this by making the allocator sensitive to program flow structure. The main idea is to represent the program's loop and conditional structure as a hierarchy of nested *tiles*. Tiles can be basic blocks, conditionals or loops and they are coloured in a bottom-up fashion. Spill instructions are inserted in less frequently executed portions of

the program.

Another method to try to take more of the program structure into account, using *hierarchical cyclic interval graphs*, is described by Hendren *et al.* [HGAM92, HGAM93]. The main motivation is to be successful when allocating over loops. Cyclic interval graphs is an alternative representation compared to the standard interference graph. Their algorithm will, whenever possible, favour live range splits (called register floats) over register spills.

Methods to make register allocation possible for *subscripted variables*, like array elements, are presented by Callahan *et al.* in [CCK90] and by Duesterwald *et al.* in [DGS92]. Array elements are usually not considered as candidates for allocation at all, since allocating anything else than simple variables requires more powerful data flow methods.

A method that is not really graph colouring but uses related methods is Proebsting and Fisher's *probabilistic register allocation* [PF92]. Probabilities are used to decide what values to allocate to registers, in a way that is similar to the usage counts method, the most beneficial ones are allocated to registers.

Procedure calls

The introduction discussed the handling of procedure calls in a global register allocator, using standard linkage conventions (see page 13). The standard question of which one to use, *caller-saves* or *callee-saves*, is examined in some detail by Davidson and Whalley in [DW91]. They find that most of the time, the best results is achieved with a split caller/callee-saves convention, where the register set is partitioned into two disjoint sets. The register allocator decides, for each local variable, from which set it should be given a register. A short-lived temporary is probably best kept in a caller-saves register, whereas a long-lived local variable that is live across several procedure calls, is better put in a callee-saves register. A split linkage convention was used in Chow's implementation of priority based colouring.

6.4.3 Interprocedural register allocation

In the introduction (on page 14) we characterised previous attempts at interprocedural register allocation as doing either *per-procedure* allocation in a *bottom-up* traversal of the program call-graph, or as doing real program-wide register allocation, for all variables in the program at once.

Per-procedure bottom-up register allocation

A bottom-up algorithm was developed by Steenkiste and Hennessy [SH89, Ste91] and implemented in the context of the PSL (Portable Standard Lisp) compiler [GBJ82]. The main motivation for the bottom-up algorithm was the observation that many Lisp programs "spend most of their time in the bottom of

the call graph". If the allocator ran out of registers in the higher regions of the call graph, it reverted to the standard PSL linkage convention. Measurements showed that they were able to reduce more than 70% of the stack references that remained after the standard register allocation, resulting in a 10% speedup.

Recursive calls were handled by taking the strongly connected components (SCCs) of the procedure call graph. For calls inside the same component, i.e., possibly recursive calls, all local variables that were live across the call had to be saved and restored around the call site. Indirect calls were handled by saving all registers in use before doing a procedure call, i.e., saving all live registers for all *ancestors* in the call graph. Registers used to pass parameters were chosen according to the PSL compiler's standard (fixed) convention.

Mulder *et al.* implemented the same algorithm for Pascal [FMM87] with almost identical results, 73% eliminated stack references using 16 registers. When going to 64 available registers, 87% of the stack references were eliminated.

Independently, Chow developed a similar algorithm [Cho88], but with call-intensive C programs in mind. This was done in the context of the MIPS R2000 compiler system [CHKW86]. Chow divides all procedures into *open* and *closed procedures* as a unified way of handling many of the problems mentioned earlier; recursion, indirect procedure calls and separate compilation. *Closed* procedures are "well behaved" and normal bottom-up register allocation can be applied. For calls between closed procedures, he is able to use special (not fixed) parameter registers to even further reduce the procedure call overhead. This was deemed too complicated in the PSL compiler which used a fixed linkage convention. Chow calls a procedure *open* if some of its callers have been compiled before itself (recursive procedures) or if some of its callers are unknown (indirect calls or calls from other modules). For open procedures a standard split caller/callee-saves convention is used.

Chow measures speedups ranging from 1% to 14% with a mean of 1.3% when compared to the intraprocedural allocator. Chow's results may seem disappointing when compared to those obtained by Steenkiste and Mulder, but this is probably because the MIPS intraprocedural register allocator was already very well tuned. Furthermore, it used a split caller/callee-saves convention that reduced some of the procedure call overhead. An important factor was also that Chow measured on C programs, which may have been less call intensive than Steenkiste's Lisp programs.

None of these bottom-up allocators applied the interprocedural register allocation to global variables since that does not really fit well into their per-procedure compilation. It requires program-wide information to succeed well.

Program-wide register allocation

Wall describes an algorithm for register allocation on all the variables in the program at once [Wal86]. The real register allocation is done at *link-time*, and hence the allocation at compile time is very limited, mostly consisting of anno-

tations describing how the code should be changed if a variable gets allocated to a register. Estimated *usage frequencies* are used to decide what variables should be allocated to registers. Similarly to Steenkiste and Chow, Wall uses the observation that procedures not active at the same time can use the same registers for their local variables. Since Wall has program-wide information available he is able to allocate heavily used global variables to registers.

The problems with indirect procedure calls and recursion are handled in a similar way as in Steenkiste's implementation, but instead of saving registers incrementally for each (possibly) recursive call, the registers for the complete SCC are saved on the *back edges* of the call graph (see figure 6.4 on page 203). For indirect calls, parameters are passed on the stack and loaded into registers by a procedure preamble. A direct procedure call would skip the preamble.

Wall compares his program-wide algorithm to Steenkiste's bottom-up approach and concludes that with the use of profiling information, his approach performs slightly better, but without it, the bottom-up algorithm is slightly better [Wal88].

Santhanam and Odnert [SO90] attacks the problem that a global variable in Wall's approach either will be allocated to a register for the complete program or not at all. Their solution is to identify regions of the procedure call graph, called *webs*, where a global is heavily used (possibly a call intensive region) and allocate the global to a register inside that region. In the rest of the program, where the global is less frequently used, it will be kept in memory, freeing up the register for other variables. Also, they will try to move spill code out of call intensive regions, trying to take into account that the observation used by the bottom-up allocators, that most programs "spend most of their time in the bottom of the call graph", is not always true. The concept of webs is in spirit similar to our use of clusters, in the extended register allocation algorithm (see section 6.3 on page 233).

Other methods

An approach that does not really fit into the two classes of interprocedural register allocation that we defined above (per-procedure vs. program-wide), is the work by George *et al.* in the context of the ML-RISC back-end [GGR94, Geo96] and in recent versions of the SML of New Jersey compiler [AM87]. In short, Chaitin-style graph colouring is applied to more than one procedure at a time [GA96b] (normally for calls to local functions inside a *module*), and standard calling conventions are used otherwise. The resulting method seems very similar to our base algorithm, e.g., using coalescing across module boundaries to achieve tailor-made calling conventions, but their technique is used in a much more "practical" way (colouring a couple of procedures rather than the entire program).

6.4.4 Lazy functional languages

Not much work have been done on register allocation specialised to lazy functional languages. An investigation of the methods used is further complicated by the fact that most descriptions of implementations of lazy functional languages concentrate on describing mostly "high level" details such as the *abstract machine* used. Of course, design decisions at this level can be vital to register usage in the underlying implementation, but the actual mapping of the abstract machine onto the hardware is often described in very little detail. This makes it hard to classify the register allocation algorithms used in different implementations.

Native code

The well known Chalmers Lazy-ML compiler by Augustsson and Johnsson [Aug84, AJ89c] uses a code generator based on attribute grammars [Joh81] and the resulting register allocation is a kind of local *register tracking* algorithm. Smetsers *et al.* [SNvGP91], compiling the language Clean, go one step further in using an ad-hoc global (intraprocedural) algorithm, e.g., in the m68k port [Gro90]. 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 [AJ89b] once used a global graph colouring algorithm, resulting in a 25% speedup compared to the simple code generation algorithm.

Generate C

A popular approach recently has been to compile lazy functional languages into some form of the C language, and thereby take advantage of the great deal of effort put into writing register allocators and other optimisations for C compilers.

The FAST compiler [HGW91b, HGW91a], 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). Many 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 [LH92]. Here, the Functional C is further compiled, and an explicit stack added to support the garbage collector. The result, Koala code, is very similar to "standard" abstract machine code, like G-machine code, and the C compiler is used as a portable assembler. Things like passing arguments in registers have to be explicitly introduced in the Koala code (using global variables that are explicitly mapped to registers).

The Glasgow Haskell compiler, by Peyton Jones *et al.*, implementing the Spineless Tagless G-machine [PJS89], 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 [PJ92].

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. E.g., doing a direct (non-returning) jump is in general not possible in C, but vital in an implementation of the STG machine. To do this they use the possibility of most C compilers to write direct *inline assembly code*. Unfortunately, doing such things can often inhibit the C compiler from doing many of the optimisations that it would have been able to do on a "clean" C program. Another problem is the control of register usage, where special treatment is necessary to put for example the heap pointer in a register and to pass function arguments in registers.

Perhaps the most serious problem, however, is that the global register allocation that is normally 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 (see the discussion in the introduction, on page 15).

6.4.5 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, e.g., Steenkiste have used bottom-up allocation for Lisp, and George interprocedural graph colouring for SML (both described above).

Register allocation in general is much more well examined for strict functional languages than for lazy ones, probably much due to the large Lisp/Scheme and ML communities, but maybe also because implementations of strict languages in some sense are more similar to implementations of traditional imperative languages. Many traditional optimisation methods tend to apply better to strict languages than to lazy ones. Just to name a few compilers, that have contributed in this area; the SPUR Lisp compiler described above, the Orbit Scheme compiler by Kranz *et al.* [KKR+86] and all the work on the SML of New Jersey compiler, by Appel and MacQueen *et al.* [AM87, AJ89a, AS92] and recently by George, Guillame and Reppy in [GGR94] and by George in [Geo96].

Much of this work is related to the use of CPS [Ste78, AJ89a], which we discussed in section 2.6.2 on page 61. It was noted already by Steele [Ste78] that the conversion to CPS can make some optimisations more precise. An example of this is the register allocation technique used in the *Orbit* compiler, called *trace scheduling* [KKR+86], which due to the use of CPS achieves a simple variant of register allocation across procedure boundaries. The idea is to use a heuristic in the code generator that at *forks* decide the most probable branch, follow that and remember the register state at *join* points. When doing the code generation for the other branches, a *join* means that the registers are synchronised. As a result it achieved what we call a tailor-made calling convention for local and

known functions.

It is interesting to note that CPS conversion is a caller-saves convention since values that are needed after a function call is stored in a closure before the call and later restored by the continuation. In [AS92] Appel and Shao show how a callee-saves convention can be added to CPS, essentially by introducing additional arguments for function calls where callee-saves registers are preferred. George and Appel later note that coalescing (copy propagation) becomes crucial for the technique to work well as the number of callee-saves registers is increased [GA96b].

An interprocedural bottom-up register allocator similar to Steenkiste's is developed for SML in [KHO96]. Considerable speedups are reported, much due to the use of individual linkage conventions for functions.

A technique with similar purpose to our shrink-wrapping, but with an overall goal of being very efficient (linear), was presented in [BWD95].

A very novel approach to register allocation has recently been proposed by Agat [Aga98], where the register allocation problem is attacked using techniques from type and effect systems.

Chapter 7

RISC optimisation

This chapter will describe optimisations done on RISC code, with the exception of the register allocation and issues related to garbage collection (these are both described separately, see chapters 6 and 8, respectively). The common factor of all RISC optimisations described here is that they are all more or less standard optimisations, done in many other optimising compilers. However, we still believe that it is interesting to see "the entire picture", so we will describe them briefly here, to make the presentation of the GRIN back-end complete.

7.1 Overview

Some of our RISC optimisations are specialised to code generated from a functional language (like the optimisations of tailcalls and heap allocation), whereas others are more general (like instruction scheduling).

Figure 7.1 on the following page shows how the RISC optimisations fit into the RISC part of the back-end. Compare this to figure 1.3 on page 17 that shows the organisation of the entire compiler.

The RISC optimisations are divided into two parts, done before and after the register allocation. The first part, the phase labelled "RISC optimisations part I" in figure 7.1, currently consists of a single optimisation:

- *dead basic block elimination*: delete basic blocks that can never be reached (see section 7.2 on page 249).

The second part, the phase labelled "RISC optimisations part II", consists of:

- *peephole optimisation*: match some very simple patterns (see section 7.3 on page 250),
- *tailcall optimisation*: re-introduce and optimise tailcalls (see section 7.4 on page 250),

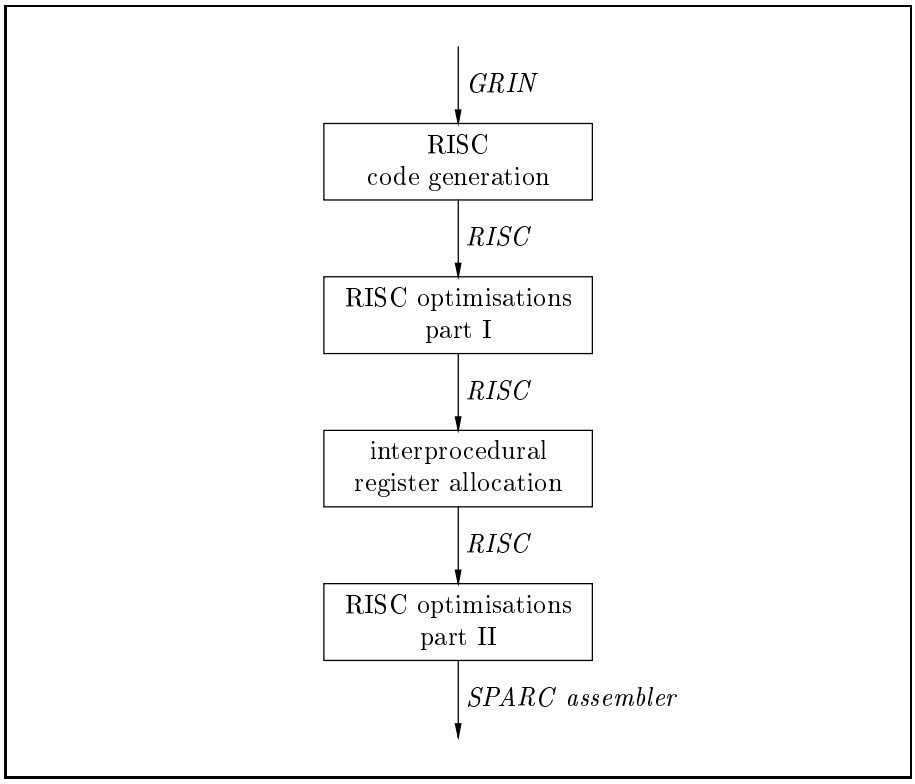

Figure 7.1: Overview of the RISC part of the back-end.

- *stack frame optimisation*: optimise allocation and deallocation of stack frames (see section 7.5 on page 254),
- *heap optimisation*: optimise heap allocation (see section 7.6 on page 255),
- *GC code*: insert code to do "heap full?" tests before heap allocation and to call the garbage collector (this is entirely described in chapter 8, see section 8.3.3 on page 269),
- *basic block scheduling*: determine the order (in the code) of basic blocks (see section 7.7 on page 255),
- *instruction scheduling*: schedule ordinary instructions, optimise branches and fill delay slots (see section 7.8 on page 256),
- *GC descriptors*: calculate the final root pointer information to be used by the garbage collector (this is entirely described in chapter 8, see section 8.3.4 on page 270).

Maybe the most notable property of the optimisations is that they are so few! Many standard low level optimisations (e.g., constant propagation and common sub-expression elimination) are missing. The reason for this is that many traditional optimisations are subsumed by GRIN program transformations (recall that the final GRIN code is quite "low level", it has good control of many low level details, see section 4.4.5 on page 161). So, there is less need to rerun the same optimisations on the RISC level. Of course, this is not entirely true, there would probably still be room for some improvement in the RISC code after the code generation, but the possible gain would be comparatively small.

Another thing that can be seen in figure 7.1 is that the register allocation is done almost immediately after the RISC code generation, and the bulk of RISC optimisations done *after* the register allocation. This may look a bit strange when compared to other optimising compilers, where register allocation is normally done very late, and with most optimisations done *before* the register allocation. However, this is mostly an illusion, if we take all GRIN optimisations into account we still have the same property for the GRIN back-end, that the register allocation is done late, after most other optimisations.

Finally, after all RISC optimisations the RISC code is "pretty printed" as SPARC assembler. This is trivial due to the close correspondence between our RISC code and the SPARC architecture (see section 5.1 on page 165).

In what follows we will give a brief overview of each of the RISC optimisations.

7.2 Dead basic block elimination

During the RISC code generation we do an optimisation of sequences of GRIN case expressions, and try to short-circuit unnecessary tests (see section 5.3.2 on page 189). Sometimes this will result in some unreachable basic blocks in the flow graph. For an example, see figure 5.8 on page 192 and imagine that all the alternatives of the first case expression would be short-circuited (all had a "known return value"). In that case the block labelled "case2 test" would be dead since it could never be reached. This situation is not detected by the RISC code generator, and we use a standalone RISC optimisation immediately after the code generation to take care of it.

Technically the optimisation is very simple. All that needs to be done is to examine each intraprocedural flow graph and delete blocks with no predecessor (except the procedure entry block). We use a simple work-list algorithm to avoid the problem of ordering the basic blocks.

7.3 Peephole optimisation

This is a standard peephole optimisation that tries to match a few very simple patterns. The only interesting pattern is probably register to register copy instructions where the source and destination registers are the same:

$$move \quad r_1, r_1$$

Such a strange instruction can only be due to a "failed coalesce" where the two virtual registers involved just happened to receive the same colour by the *select* phase of the register allocator. A technique such as *biased colouring* [Bri92] could make such patterns appear more often, but we have currently no mechanism in the register allocator to prefer a certain colour based on copies in the code.

7.4 Tailcall optimisation

As mentioned earlier the current RISC code generator treats tailcalls as ordinary returning calls (see section 5.3.2 on page 183). The reason for this is to avoid restricting the available *return registers* that are assigned to procedures by the register allocator. To see why, consider a procedure A that potentially could tailcall another procedure B. Because of the potential tailcall this would force A and B to use the same return registers. This in turn could make the register interference graph harder to colour. Our idea is instead to generate ordinary returning calls for all GRIN tailcalls in the original RISC code, and then "re-introduce" tailcalls in the RISC code when the register allocation is done (the motivation behind this is further discussed below).

An example of a tailcall in the GRIN code that is changed into an ordinary call in the initial RISC code can be seen in figure 7.2 on the next page. The figure shows the RISC code before the tailcall optimisation for the *sum* function of the program in appendix A (see page 314 for the final GRIN code for this function). To make the code in figure 7.2 easier to read we have already incorporated some of the changes done by the *basic blocks scheduling* and the *instruction scheduling* that really are done at a later time (we do not show unnecessary basic blocks and branches to immediate successors). We also show the argument and return registers of the procedure as a comment. The important thing to note in the figure however, is that the recursive call is an ordinary returning call followed by a "*return*", not a tailcall. Note also that the *shrink-wrapping* optimisation (see section 6.2.3 on page 202) have moved the save and restore of the return address into the recursive part of the function.

The tailcall optimisation is done in two steps:

1. Re-introduce tailcalls, basically replacing a "*call*" instruction with a branch and adjusting the return address and stack pointer to match.

```
sum :                                   ! [r1, r2, r3] ↦ [r1]
        add     %sp, -8, %sp
        cmp     r2, r3
        bgt     L2
        nop
L1 :    store   %o7, [%sp + 4]
        add     r2, 1, r4
        add     r1, r2, r1
        move    r4, r2
        call    sum
        nop
        load    [%sp + 4], %o7
        add     %sp, 8, %sp
        return
        nop
L2 :    add     %sp, 8, %sp
        return
        nop
```

Figure 7.2: The *sum* function, before tailcall re-introduction.

2. Optimise the tailcalls further, removing unnecessary handling of the return address.

Together these two steps can transform a tail recursive function into a "real loop" (this is the first time during the compilation process that we can get cycles in the intraprocedural flow graphs).

7.4.1 Re-introduce tailcalls

First we examine the entry block of all procedures and split each block into two parts, i.e., we introduce a new label. This is done to create an "alternate entry point" for each procedure which avoids adjusting the stack pointer and possibly also avoids saving the return address (in some cases the return address save has been shrink-wrapped and do not appear in the entry block, e.g., in figure 7.2). According to the instructions present in the entry block we categorise each procedure into one of the following three classes:

1. leaf procedures (always expect the return address in its register),
2. non-leaf procedures with the return address save shrink-wrapped,
3. non-leaf procedures with the return address save still in the entry block.

```
sum :                                   ! [r1, r2, r3] ↦ [r1]
        add     %sp, −8, %sp
L3 :    cmp     r2, r3
        bgt     L2
        nop
L1 :    store   %o7, [%sp + 4]
        add     r2, 1, r4
        add     r1, r2, r1
        move    r4, r2
        load    [%sp + 4], %o7
        ba      L3
        nop
L2 :    add     %sp, 8, %sp
        return
        nop
```

Figure 7.3: The *sum* function, after tailcall re-introduction.

Next, we examine all call blocks for each procedure, and identify calls with no "useful" instructions between the call and the return (in the successor block). The only instructions that we allow are adjustments of the stack pointer and a restore of the return address register. For call sites that match this we transform into a real tailcall, and jump to the alternate entry point of the callee that we created above. The function arguments are already setup, but depending on the current stack height and the type of the callee (the three classes above) we will generate slightly different code (before the actual tailcall):

- if the stack height of the caller and the callee differs, adjust that,
- if the callee is of class 1 or class 2, restore the return address register,
- if the callee is of class 3 there is no need to restore the return address since all procedures store it in the same location in the stack frame (the callee can fetch it where the caller once put it).

Note that the above is independent of if the caller and the callee are the same function (since the entire program is available in the GRIN back-end this is never a problem).

The sum function in figure 7.2 will, after the tailcall re-introduction, look like in figure 7.3. Note how we have introduced a new label "*L3*" and how the "*call sum*" and the instructions following it have been changed into a restore of the return address and a branch.

Discussion

This method, to first remove all tailcalls and then re-introduce some of them, is not an ideal solution. As discussed above we do this to avoid unnecessary register interferences due to procedures being forced to use the same return registers. For the tailcall re-introduction to be able to put back all tailcalls it requires that the *coalesce* phase of the register allocator has removed all return transfer copies introduced by the code generator (see figure 5.5 on page 184). Unfortunately this can not be guaranteed in general, which means that some tailcalls can be lost and we have no control over which that is. The situation could be made a bit better by using *priorities* in the coalesce phase to increase the probability that return transfer copies are eliminated, but this can never be guaranteed to succeed.

The best solution would probably be to use the strongly connected components of the procedure call graph to identify potential loops in the program. Then we would only need to make sure that we did not break tailcalls when both the caller and the callee are in the same component. Other tailcalls are not a catastrophe if we lose, so we can let the coalescing work as usual. It is also possible that it does not really matter much in practice if we force some procedures to use the same return registers, in which case the whole re-introduction process is unnecessary. More experiments are needed to decide what is best.

7.4.2 Optimise tailcalls

The second part of the handling of tailcalls in RISC is to optimise the code produced above a bit further. In particular we can eliminate some unnecessary saves and restores of the return address register. We do this by calculating a few global data flow attributes (by iteration, recall that the intraprocedural flow graphs can now have cycles). The main attributes used are variations of the normal *Live* and *AV* (available) attributes, which in this context is taken to mean:

- $Live(b)$: in basic block b, a need for the return address in its memory location in the stack frame may occur.
- $AV(b)$: in basic block b, the return address is definitely available in its register.

After that we try to delete as many as possible of the return address saves and restores that are done first in tailcalled functions, and immediately before tailcalls. In short:

- If we find a save of the return address in basic block b, and not $Live(b)$, then delete the save.
- If we find a restore of the return address in basic block b, and already $AV(b)$, then delete the restore.

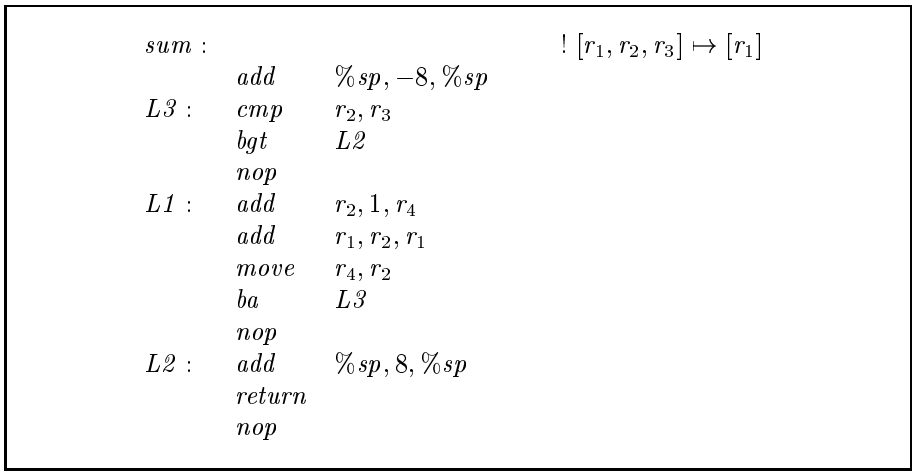

Figure 7.4: The *sum* function, after tailcall optimisation.

After this the code for our running example will look like in figure 7.4. Note that the return address is now kept in its register during the whole loop (the return address of the original caller). The stack frame allocation and deallocation have now also become unnecessary and will be removed by the stack frame optimisation below.

7.5 Stack frame optimisation

The next RISC optimisation is to optimise the allocation and deallocation of stack frames, i.e., adjustments of the stack pointer. At an earlier stage, the *post process* phase of the register allocator (see section 6.2.11 on page 230) inserted stack adjustments in procedure entry and exit blocks, just to get "runnable code". We can now optimise this:

- Not all execution paths through a procedure may need stack space. In that case, try to push the allocation in the entry block into the procedure, for as long as not all successor blocks need stack space. Delete the exit block deallocations for all paths where it has not been allocated.
- The tailcall optimisations may have eliminated the need for a stack frame completely. In that case, delete it all.

This is in fact very similar to the shrink-wrapping used to place saves and restores of the return address (see section 6.2.3 on page 202). But since the need for stack frames is not totally visible until after the register allocation stack frame allocation can not take part in the normal shrink-wrapping (at

```
sum :                                   ! [r1, r2, r3] ↦ [r1]
L3 :    cmp     r2, r3
        bgt     L2
        nop
L1 :    add     r2, 1, r4
        add     r1, r2, r1
        move    r4, r2
        ba      L3
        nop
L2 :    return
        nop
```

Figure 7.5: The *sum* function, after stack frame optimisation.

least not in the current implementation). Instead we use a similar but much simpler technique with a few global data flow attributes, and then use that to move or delete stack adjustments.

The result for our running example is shown in figure 7.5, where the stack frame has been completely eliminated. The function has now turned into a "tight loop".

7.6 Heap optimisation

Our heap optimisation is a completely standard optimisation in a compiler for a lazy functional language. Its sole purpose is to avoid repeated heap pointer adjustments inside a single basic block. It is always enough to adjust the heap pointer only once in a basic block. Note also that it is no problem to do this after the register allocation, since the optimisation needs no new registers (it will only adjust offsets in *store* instructions).

7.7 Basic block scheduling

The RISC basic block scheduling is also a completely standard optimisation. We try to minimise branches by ordering the basic blocks and enable "fall-through" as often as possible. Basically we do a depth first ordering of each intraprocedural flow graph.

As mentioned earlier the basic block scheduling had already been applied to the sum function shown earlier. Without that the basic block ordering would be arbitrary and we would have to insert explicit branches to all successors.

7.8 Instruction scheduling

To get decent code, our RISC optimiser includes a simple instruction scheduler. It has two parts: local instruction scheduling and branch optimisation.

7.8.1 Local instruction scheduling

The local instruction scheduling assumes a delayed-load architecture (like the SPARC) and uses standard techniques to order the instructions in each basic block. More specifically we have implemented something like [GM86]. We build a dependency graph for the instructions in each basic block, and select instructions to avoid interlocks. A simple set of heuristics is used to select an instruction when several candidates are available. We also optimise the control flow instructions at the end of the basic block, enable fall-through to immediate successors (according to the basic block scheduling order), and try to fill delay slots with useful instructions.

7.8.2 Branch optimisation

The next part of the scheduling is mostly a standard "branch chaining" optimisation, that eliminates different kinds of branch-to-branch sequences. We also do a final attempt to fill delay slots that the local instruction scheduling could not fill. This can almost always be done by changing the target of branches and/or using annulled branch instructions.

Chapter 8

Garbage collection support

This chapter will describe the methods used by the GRIN back-end to support automatic memory management, via *garbage collection*. Doing so in the presence of aggressive optimisations, such as interprocedural register allocation, turns out to be non-trivial. We will first present what the problem is, in the form of a number of requirements that various optimisations put on the garbage collection method. After that we will explain our solution in two steps: *runtime* support, and *compile time* support. We conclude with a discussion of related work.

8.1 Background

Most functional programs allocate memory at a horrendously high rate, and often in many small-sized chunks. A large part of that memory is also very short-lived, necessitating some kind of *reuse* of dead memory. Since that normally means moving heap objects around, it will exclude *conservative garbage collectors* [Wil92, Boe93]. Exact garbage collection requires quite close cooperation-operation between the compiler and the garbage collector, since the garbage collector must be able to find *all live data* (an approximation will not do), interpret it and possibly move it. The actual garbage collection usually includes interpreting some kind of stack, to find all *root pointers* (pointers to live heap objects).

There are many things that a compiler can do to make things "harder" for the garbage collector, e.g., use a complicated layout for heap objects (needing complex operations to determine what an object is), or do optimisations that make it harder to find where all root pointers are. All this "compiler information" must be communicated to the garbage collector in some way.

In general, the problem gets more complicated the more advanced optimisations the compiler performs. As an example, how should a garbage collector know if the contents of a processor register is an integer or a pointer? It would

be a catastrophe if it tried to follow an integer as it if were a pointer. One way to handle this is to use *tag bits* (to distinguish pointers from non-pointers), or to pass runtime *type parameters*. Unfortunately both these techniques have a runtime cost, i.e., a cost in the *mutator* (the normal execution, outside of the garbage collector), which is something that we want to avoid as far as possible. Optimally, we want a garbage collection method that doesn't slow down the mutator at all.

8.1.1 Overview

The key question that is discussed in this chapter is:

> *how does the garbage collector find all root pointers?*

We will present a method that solves this problem without incurring any extra mutator overhead. The other task of the garbage collector, interpreting data stored in the heap, is handled in a standard way and will only briefly be discussed (see section 8.2.4 on page 266). To summarise the root-finding machinery:

- *Compile time support*: The compiler must be able to determine, for each value, and for each potential GC point, if the value is a pointer or not. This problem is easy in the beginning of the compilation process, but gets more and more complicated as the program is transformed and optimised. This is particularly true after the translation to RISC code. Our solution includes calculating various kinds of liveness and availability data flow information at some strategic points during the compilation.

- *Runtime support*: The information gathered by the compiler needs to be communicated to the garbage collector in some way. We do this by providing a procedure *gc_find_roots*(), that a garbage collector can use to find all *root pointers*, i.e., pointers to live heap objects. Descriptor tables are associated with each potential GC point (including all call sites), describing the layout of the stack, register contents, etc. At each GC, the *gc_find_roots*() procedure traverses the runtime stack and interprets each stack frame, looking for pointers. The method imposes no mutator overhead at all.

Below we will describe these two issues in the reverse order, i.e., first a runtime solution, and then the compiler support needed to implement it (this has also appeared in [Boq98]).

8.1.2 The problem – how to find all roots

However, before describing the runtime machinery, we will briefly describe some of the "features" of the GRIN back-end, i.e., some optimisations it does and other

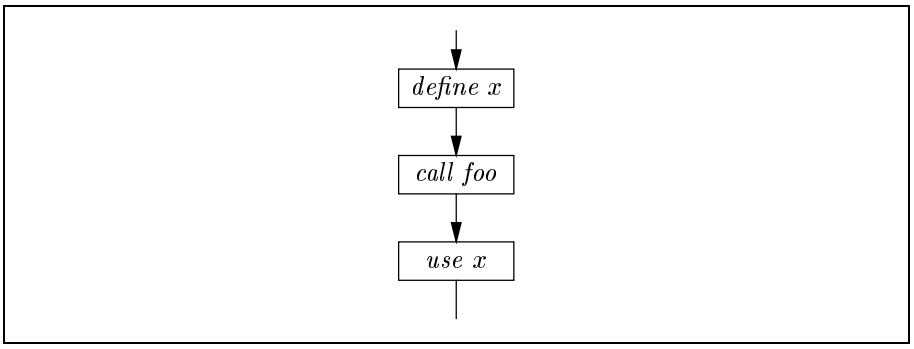

Figure 8.1: A variable that is live over a call.

assumptions made in the code that it produces, to show some of the difficulties for the garbage collector.

A typical Haskell compiler will keep all root pointers on a special *pointer stack*, which means that the process of finding all roots becomes trivial, just scan the pointer stack. As far as root pointers is concerned, the only thing the implementation needs to guarantee is that at potential GC points all pointers to live heap objects (and nothing more) are stored on the pointer stack (possibly modulo some global objects like CAFs). In the GRIN back-end it is much more complicated. We do not use a special pointer stack. Instead we have a base assumption that all pointers (and other variables) are allocated to registers. When this is not possible, they are *spilled* to the normal system stack, necessitating more refined methods to allow the garbage collector to find all root pointers.

Feature 1: interprocedural register allocation

The main complication is the *interprocedural register allocation* (see chapter 6). As a result of interprocedural register allocation, values can be kept in registers across procedure calls. Consider the small example code in figure 8.1.

In the figure, the goal is to try to keep "x" in a register over the call to *foo*. This generally requires some kind of interprocedural register allocation (in this case we can get the same effect just by using a *callee-saves* register convention). The consequence of this optimisation is rather large however, when it comes to the issue of garbage collection. It means that at entry to the garbage collector, the *register state* is unknown, i.e., the *owners* of all registers are unknown. The value in each register can "belong" to any active procedure invocation on the call stack (i.e. the stack that are used to support procedure calls, in our case the standard system stack). Before we know the owner of a register there is no way we can tell if the contents is a pointer or not.

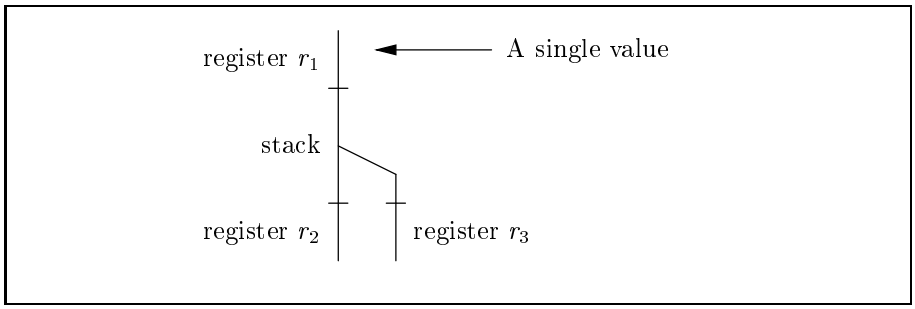

Figure 8.2: An example of a spilled live range.

Feature 2: values change locations

A single *value* (locals/temporaries/arguments etc.) may reside in different registers or in a stack slot at different code points, as a result of how the register allocator performs register spilling. An example of this is shown in figure 8.2.

The figure illustrates what was originally a single value (live range), but then became spilled to memory. After spilling, the value is kept on stack at some points, and held in (different) registers at other points. The original value is now "connected" using *spill* and *reload* instructions (see section 6.2.9 on page 226). Note that we need to keep track of such things as "the original value" since that is where the information about pointers/non-pointers originates from.

The consequence of this feature is that the garbage collector must not only be able to identify what procedure all stack frames (procedure activation records on the call stack) belong to, but also the exact "active" *call site* for each procedure.

Feature 3: interprocedural live ranges

Yet another complication is that the *interprocedural coalescing* in the register allocator (see section 6.2.5 on page 214) can create very long live ranges, that span over several procedures. The interprocedural live ranges are "communicated" between procedures when used as argument or result registers in function calls. An example of such a live range is shown in the left part of figure 6.13 on page 231.

Especially interesting is the case where a live range just "passes through" a procedure, i.e., it is used as an incoming argument for a certain procedure, but then is never actually used inside the procedure, instead it is only passed as an outgoing argument to a new function call. Such a live range will not occur anywhere in the code of that procedure, but still it is part of the register state inside the procedure!

Feature 4: multiple register values

The normal return convention for GRIN procedures specifies that they return a complete *node* in registers. The return registers are not fixed, and are determined, for each procedure, during the normal register allocation. E.g., a function that returns a list (of something) will be given 3 registers to return a value in. The first register will be used to hold a tag (representing either a *nil* or a *cons*). Depending on the tag, the two remaining registers will be either unused or contain the head and the tail of the list.

This means that at any point in the code, there might be "complete nodes" in registers and these nodes must of course be garbage collected exactly as nodes in the heap, i.e., we must check them for root pointers. However, it is technically much harder to "tell" the garbage collector that a node in registers is a root than to just tell it that a pointer to a node in the heap is a root. For the example above, a list node in registers, the garbage collector must examine the first register before it can determine if the other two are roots or not. If we call the return registers r_1, r_2 and r_3, we can summarise the previous by saying that the *types* of r_2 and r_3 *depend* on the *value* of r_1.

In this example it would have worked to set unused registers to some special value (like zero), and then test for this special value before a (potential) root pointer is followed. However, this is not enough in general. E.g., we can easily create a datatype where the first argument is either a pointer or an (unboxed) integer. Also, having to explicitly set registers to zero in the code would impose a mutator cost, something that we really want to avoid.

A further complication is that a node can be *partly in registers*, if some registers (that was part of a "complete node" in registers) have been spilled to stack.

Feature 5: stack frame allocation

Different *code paths* inside a procedure can have different stack frame layout. Typically, recursive functions will often not need to allocate a stack frame at all for the base case(s), since these are normally very simple. This means that at certain points inside a procedure there might not be a stack frame at all.

Another complication is that different values can spill to the same stack location, so it is not possible to say that a certain stack slot will always be a pointer. The consequence of this is that we will also have to associate information about stack frame layout and total height with every call site in the code (in fact, with all potential GC points).

Feature 6: the return address

The last feature is really a special case of "Feature 2", but an important one: the return address. It is extra important since the return address will be the only "link" to the previous call site (i.e., to the *caller* of the current procedure).

As with other values, the return address can be stored either in a register or in the stack frame. A convention we use, however, is that when in a register it is always in the special RISC return address register ("$\%o_7$"), and when stored on stack, it is always stored in the top-most slot in the stack frame.

8.2 Runtime machinery

The above features all add up to rather "complicated" code seen from a GC perspective, but we will now describe a runtime machinery that can handle all of the problems described, and with no mutator overhead at all. Additionally, the cost of finding the roots when a garbage collection actually occurs, will be small compared to the overall cost of doing a garbage collection.

In essence, we provide a "*gc_find_roots*()" procedure that can be used by any kind of garbage collector to find all root pointers. This procedure will traverse the runtime call stack, and use GC descriptor tables to interpret stack frames, searching for root pointers. Each potential GC point (including all call sites) will have a descriptor table associated with it.

8.2.1 Modified call sites

The first part of our runtime machinery is to find a way in which information about the current register and stack state (in the form of a descriptor table) can be associated with all *potential GC points* in the code. To understand our notion of potential GC point, let us consider the state exactly at the point of entry into the garbage collector. At that point, a garbage collection must have been triggered by some *failed memory allocation* (because the heap was full). The state of the procedure call stack (the activation records on the standard system stack) at this point is:

- the top-most procedure activation is always the one who caused the GC call, i.e., the one in which a heap allocation failed,
- all the other procedures on the call stack are "waiting" for a call that has not yet returned.

Note that the common property here is that all active procedure invocations will be waiting at a procedure call. The "last" procedure is waiting at the call into the garbage collector (which was reached due to a failed allocation), and all the other procedures are waiting at ordinary call sites (to GRIN procedures). This means that the points in the program where a GC can occur, or the *potential GC points*, are exactly all the call sites of the program.[1]

[1] Except for external calls (the *ecall* RISC instruction), since external functions in the current design are not allowed to allocate heap memory.

So, if we can just find a way to associate information (a descriptor table) with each call site that should be sufficient for our needs. Fortunately, this is easy, and we will use a well known technique to insert an extra pointer word after each call instruction and then modify all return addresses correspondingly.

Procedure calls in RISC code have a single *delay slot* (see section 5.1.3 on page 172). So, a normal call site looks like:

call foo
⟨*one delay slot instruction*⟩
⟨*rest of code*⟩

When a procedure call occurs, the address of the return point ("*rest of code*" above) is automatically put in the RISC return address register ("$\%o_7$"). Since each RISC instruction is 4 bytes (this is all copied directly from the SPARC, see chapter 5), the return address for a call will always be the address of the *call* instruction plus 8. In fact, the RISC *return* instruction is just a short-hand for:

jmpl $\%o_7 + 8, \%g_0$

The use of the special *zero* register ("$\%g_0$") signals that the jump should be made without a new return address being saved.

Since we need to associate a descriptor table with each call site, we will modify the call, and insert an extra word:

call foo
⟨*one delay slot instruction*⟩
⟨*pointer to a descriptor table*⟩
⟨*rest of code*⟩

All returns from functions is now modified to skip over the descriptor table pointer when returning:

jmpl $\%o_7 + 12, \%g_0$

Note that this method has no extra cost during the normal execution, it just gives us a way to find the descriptor tables when a garbage collection actually occurs.[2] Note also how both ordinary function calls and allocation failures can be handled using this single technique, as described above.

For architectures with a "fixed" return instruction, we can instead use some other method to associate return addresses and descriptor tables. E.g., we can build a global (hash) table indexed by return addresses, that contains pointers to all descriptor tables. Also this method has the property that it avoids all mutator cost.

[2] Actually, the technique may incur a cost on architectures that use special hardware to *predict* function calls and returns.

8.2.2 GC descriptor tables

The second part of our runtime machinery for GC support is the actual GC descriptor tables. Each descriptor table will contain the information needed for the garbage collector to find all root pointers that "belongs" to the same procedure as the descriptor table itself does. Each procedure typically has more than one descriptor table, one for each call site (which includes all allocation points due to calls into the garbage collector). Each table describes the root pointer state of the procedure at exactly that call site.

A descriptor table has four parts:

- A bit mask describing live registers.
- A bit mask describing live stack slots.
- An integer describing stack frame height.
- One or more bytes describing "complete nodes".

The two bit masks are a single word (32 bits) each and every bit describes one register (stack slot). If a bit is set, it means that the corresponding register (or stack slot) contains a *live root pointer*. The 32 bits can easily be extended for a certain descriptor table (which most notably may be needed for the stack slots descriptor) by defining the highest bit to mean "if set then continue one more word".

The third part simply gives the total height of the stack frame (in words). This is needed to "pop off" the current stack frame and go to the previous one.

The last part of a descriptor table is a list (possibly of length zero) containing information about "complete nodes" that are stored in registers (and/or in stack slots). For each node, one byte specifies where the tag can be found (values between 1 and 31 denotes a register, and 32 and above denotes a stack slot offset by 32). When the actual tag is found we can use the normal method used by the garbage collector to find information about how many arguments a certain tag has and if they are pointers (this is the normal kind of information that any garbage collector needs, and it is stored in tables or bit encoded in the tags themselves). Assuming a tag has N pointer arguments, the next N bytes will then give the locations for the tag arguments, one per byte, encoded in the same way as the tag. A zero byte marks the end of the sequence, i.e., after examining the N pointers the garbage collector will skip over bytes until it finds a zero. Note that this zero byte is necessary (and sufficient) to signal the "end" of a node in registers, since different tags have different numbers of pointer arguments but for each node we know at compile time its maximal size. Finally, yet another zero byte signals the end of the "complete nodes" part, and hence the end of the whole descriptor table.

Note that the total size of a descriptor table is arbitrary, but this is not a problem. The only thing that is put inside the actual program code is a pointer to the table.

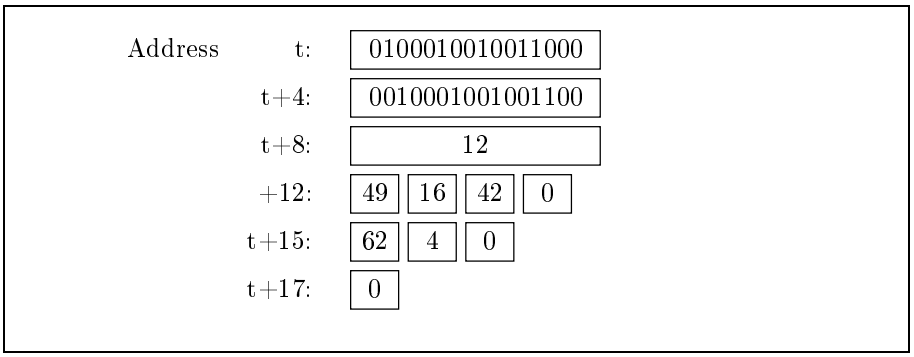

Figure 8.3: An example GC descriptor table.

Figure 8.3 shows an example of a GC descriptor table.

In the figure, the two first words are the register and stack bit masks, respectively. The 12 is the total stack frame height (in words). The numbers 49 and 62 both represent tags, i.e., stack slots 17 ($49 - 32$) and 30 ($62 - 32$) both contains a tag that must be examined. That will give a number of pointer arguments which are root pointers and must be followed by the garbage collector. In the case of 49, the maximal number of pointer arguments for the tags that can appear at that point is two, and in the case of 62 a single root pointer can appear at most (note that the number of pointer arguments is always dependent on the actual tag that is found when examining the first register/stack slot). Of the node arguments, 16 and 4 represent registers and 42 a stack slot.

8.2.3 Runtime stack traversal

Once we have created all descriptor tables, it is relatively simple to implement the "*gc_find_roots*()" procedure. Just iterate the following:

1. Use the current return address to find the descriptor table describing the previous stack frame ("$desc\_table = *(\%o_7 + 8)$").
2. Interpret the register bit mask ("$reg\_mask = desc\_table[0]$").
3. Interpret the stack slot bit mask ("$stack\_mask = desc\_table[1]$").
4. Interpret the "complete nodes" byte descriptors
5. Fetch the next (or really, the previous) return address from the top of the current stack frame, using the stack frame height stored in the descriptor table.
6. Use the stack frame height to switch to the previous stack frame.

7. Goto 1, and repeat for all stack frames on the call stack.

Before entering the garbage collector the contents of all registers are saved to memory (this is done by a small stub written in assembler, because the garbage collector itself is written in C). This ensures that all registers at the GC point becomes available to the garbage collector and that the registers are "indexable", i.e., can be indexed by the small integers corresponding to the bits stored in the descriptor table.

When a GC occurs, the "first" return address (in the first item in the list above) will point to the procedure that triggered the garbage collection, i.e., to the GC call that followed a failed allocation. All other return addresses in the chain will refer to ordinary call sites.

For each found root pointer a *callback function* is called, to let the ordinary garbage collector act on the root (and for example move the node it points to).

Liveness

An interesting observation is that since our compiler includes *live variable analysis*, the bit masks describing registers and stack slots at call sites can be made very "exact", much better than to just include "all variables in scope". Unless a pointer variable is live at the call site, i.e., it may be used after the call, it will not be included in the bit mask for that call, and the heap node it points to will be considered garbage if a GC occurs during that call (unless someone else is holding on to it of course). The interesting thing is that this is a case where our automatic method is stronger than explicit programmer controlled memory management. A programmer could not "*free*()" memory until after the call has returned! E.g., an outgoing argument that is not live after the call will not be included in the register bit mask, and hence will not be retained at a GC unless it is still live in the callee.

This property is similar to the "safe-for-space-complexity" definition of reachability by Shao and Appel [SA94]. They call it a "crucial property that any industrial-strength compilers (for functional languages) must satisfy". The idea is that once a value will not be used anymore, it must be considered dead, i.e. not contributing to the liveness of the data structure it points to, independently of scoping rules. According to Shao and Appel this is a property that many compilers for functional languages fail to satisfy.

8.2.4 Garbage collection

Sofar we have only described the process of finding root pointers. Obviously, in addition to that some real garbage collection algorithm must also be used. However, this issue is not very interesting in the GRIN back-end, and we will not describe it at all. The reason is that the optimisations done by the GRIN back-end affects only pointers to heap nodes held in registers or in the stack

frame, and not the heap nodes themselves (apart from those being held as "complete nodes" in registers, but that has already been taken care of). So, the optimisations affect only the process of finding root pointers, and not the rest of the garbage collection. This, combined with the very simple (and uniform) representation of heap nodes (see section 5.2.3 on page 178), means that most standard garbage collectors will probably be quite simple to implement in the GRIN back-end. Currently we use a Cheney-style garbage collector [Che70].

8.3 Compile time support

To be able to calculate the bit masks in the GC descriptor tables used by the runtime garbage collector the compiler must maintain a lot of information (about pointers and non-pointers) throughout the compilation process. This is relatively easy at the GRIN level but much more complicated at the RISC level. Ideally, we would want have a situation where individual optimisations would not have to worry about pointer information at all. Instead we would analyse the final RISC code and extract enough pointer information to be able to build the GC descriptors (safely). Unfortunately that does not work, the RISC code contains too little information to do that.

The complications for the runtime support that we listed above (all the "features") give rise to similar complications when it comes to the compile time side of the GC support, and would make it even more difficult (and expensive) if we were to maintain correct information throughout all optimisations. Note that even if we made our RISC code typed (using pointer/non-pointer types) that in itself would not be enough. E.g., consider entire node values that are kept in several registers, or partly on stack (see "Feature 4" above, on page 261). To give pointer information types to such registers we would need some kind of *dependent types*, since the type of the node argument variables depend on the value of the node tag variable.

Even if it is not possible to recreate all pointer information from the final RISC code, we still believe that to be a very nice basic idea, and we will use a variation of it. Our approach is to collect pointer information at a few strategic points in the compilation process, which taken together provides enough information. As a result, most optimisations can be implemented independently of the GC support (in particular the register allocator). During the compilation we include GC support at four points:

1. GRIN support: we examine the final GRIN code and remember pointer information for all variables (including dependencies between variables, for entire node values).

2. RISC code generation: we use the information collected in 1 to annotate RISC instructions with pointer information.

3. Inserting GC code: about half way into the RISC optimisations (see figure 7.1 on page 248) we insert code to check if a GC is needed before each heap allocation.

4. Calculating GC descriptors: we analyse the final RISC code and extract and recreate the pointer information needed to build the GC descriptor tables.

8.3.1 GRIN support

We collect as much information as possible from the final GRIN code by doing two traversals over the GRIN expression tree for each procedure, one backwards and one forwards:

1. Backwards: annotate expressions with free variables and *use* information.

2. Forwards: use the annotations from 1 and combine it with *def* information.

3. Take the union of all information found in 1 and 2.

There are several ways to actually extract information from GRIN expressions. One example is to use function *boxity* when we find a function application (recall that in the final GRIN code only known global functions can be called, and we always know the boxity of these). An application can look like:

$$foo\ x\ y$$

Here we can simply look up the boxity of the "*foo*" function and add pointer information for "x" and "y" accordingly. Note that this information only applies to exactly this point in the code. The variable "x" and "y" can be dependent on a case expression and have other boxities at other points in the code.

A similar example is a GRIN store operation:

$$\mathsf{store}\ (\mathsf{CCons}\ a\ b)\ ;\ \lambda p \rightarrow$$

Here we know the boxity of the CCons tag so we know that "a" and "b" must be pointers (at this point). We also know that "p" is a pointer to a node in the heap (always, since variables in GRIN are never rebound).

Case expressions and dependent variables are a bit more tricky. Consider the following code:

$$\begin{array}{l} foo\ a\ b\ ;\ \lambda\,(t'\ x\ y) \rightarrow \\ \langle m_0 \rangle \\ \mathsf{case}\ t'\ \mathsf{of} \\ \quad \mathsf{CNil} \quad \rightarrow \langle m_1 \rangle \\ \quad \mathsf{CCons} \rightarrow \langle m_2 \rangle \end{array}$$

At the first binding, of the node pattern "$(t'\ x\ y)$", we will record that "x" and "y" are sofar unknown but *depend* on the value of the tag variable "t'". Later, when we find the case expression examining "t'" we know the boxity of the CNil and CCons tags, so we can deduce the boxities for "x" and "y" in each of the alternatives ("$\langle m_1 \rangle$" and "$\langle m_2 \rangle$").

After collecting as much information as possible from the GRIN code we summarise the information and associate with each variable in the GRIN program an element of the following datatype:

$$
\begin{array}{lcl}
\mathit{data}\ \mathit{PointerInfo} & = & \mathit{Pointer} \\
& | & \mathit{NonPointer} \\
& | & \mathit{DependsOn}\ \mathit{Var} \\
& | & \mathit{TagVar}\ [(\mathit{Tag}, [\mathit{Var}])]
\end{array}
$$

The first two are easy, a variable can be either a known pointer or a known non-pointer (at all program points where it is in scope). Otherwise a variable can be either a tag variable ("t'" in the example above) or variables that depend on the value of a tag variable ("x" and "y" in the example above). For a tag variable we record, for each actual tag that its value can be, which other variables that will become pointers in that alternative. So, for the case expression above we would get the bindings:

$$
\begin{array}{lcl}
x & \mapsto & \mathit{DependsOn}\ t' \\
y & \mapsto & \mathit{DependsOn}\ t' \\
t' & \mapsto & \mathit{TagVar}\ [\,(\mathsf{CNil}, []),\ (\mathsf{CCons}, [x, y])\,]
\end{array}
$$

8.3.2 RISC code generation

During the RISC code generation we use the information collected above to keep track of the "current pointer situation". All instructions where an operand is a pointer into the heap (a GC root) are annotated to show that. In the initial RISC code these annotations will be "complete", i.e., all known pointers will be annotated (except for undecided variables). But after some optimisations this will no longer be true, since most optimisations (and in particular the register allocator) will insert new instructions without including new pointer information.

8.3.3 Inserting GC code

This phase is run as part of the RISC optimisations, after the register allocation (see figure 7.1 on page 248), although this is not very critical, it could have been placed earlier. The phase inserts RISC code to test the heap pointer before heap allocations, to see if a garbage collection needs to be performed first. If that is the case the garbage collector is called. Such a test is needed only once in each basic block that does heap allocation.

As many other implementations of lazy functional languages we use a dedicated machine registers (called $\%hp_{gc}$) to make the GC test very cheap. The $\%hp_{gc}$ register always holds a value that is a bit below the real heap top (we call the difference *hclaim*). For blocks that allocate less memory than hclaim a simple comparison of the heap pointer and $\%hp_{gc}$ serves as a test if GC is required. For blocks that allocate more than hclaim a slightly more expensive test is needed, but in that case the cost of the GC test is probably negligible compared to the cost of all the memory writes (unless hclaim is very small).

8.3.4 Calculating GC descriptors

When all RISC optimisations are done we analyse the final RISC code to find the definite pointer information to use for the GC descriptor tables. The examination of the RISC code is in spirit very similar to the examination of the final GRIN code described above. However, the RISC analysis is much more involved since it is not always easy to "understand" what a RISC instruction does (recall that the pointer annotations are no longer complete). E.g., one of the examples we used above was a GRIN store operation, in which case we would always know that the result of the store would be a pointer to a node in the heap. On the RISC level, on the other hand, a RISC *store* instruction can write any kind of data, so only a small subset of all *store* instructions will result in a pointer to a node in the heap.

But even if it is harder to get pointer information out of the RISC code, it is doable, and we use an overall strategy that is very similar to the GRIN analysis described above. We do both backward and forward traversals of the code. In this process we also take advantage of those RISC instructions that are annotated with pointer information. Taken together, this provides enough information.

Technically we perform the examination in several steps. We calculate global data flow attributes both backwards and forwards (*live* pointers and *available* pointers). Finally the liveness information is used to calculate the GC descriptor bit masks.

8.4 Related work

We have presented a method for communicating GC related information between the compiler back end and the runtime garbage collector, essentially by showing how a "*gc_find_roots*()" procedure is implemented using GC descriptor tables. The method works in the presence of the quite advanced optimisations done by the GRIN back-end, and in particular, it has no mutator overhead.

Although the details in our method are quite specific to the GRIN back-end, the overall methods used are not very new. Unfortunately low level details like this are not very often published (at least not on its own).

The idea of associating information with a particular call site by modifying the call code and then use the return address to refer back to the call site is a well known trick amongst compiler writers.

The method to traverse a runtime call stack and interpret stack frames using some kind of frame descriptors has also been widely used before. For example it is used by many C++ compilers to support exceptions (which may need to unwind the stack). In the functional community, similar methods have been implemented by for example Xavier Leroy in his various Caml compilers [Ler97a] and by Karl-Filip Faxén in his Fleet compiler [Fax97].

The overall problem that optimisations make it more complicated for the garbage collector than if we use simple and uniform representation has been studied extensively in the ML community. In particular, it has recently been very popular to use *type information* to enable non-uniform data representations (e.g. unboxing). Examples of this are Xavier Leroy's Gallium and Objective Caml compilers [Ler92b, Ler97b], Shao and Appel's Type-Based compiler [SA95] and the TIL compiler [TMC+96]. Unfortunately many of these approaches carry a runtime (mutator) overhead, such as passing extra *type parameters*, or by inserting *coercions*.

It can also be noted that even such a common technique as using *tag bits* to distinguish between pointers and non-pointers, e.g. used in the SML of New Jersey compiler [AM87], imposes some mutator overhead. Of course, tag bits have many advantages too. In the GRIN back-end we have not yet prioritised the issue of an efficient mapping of nodes and tags to memory. The current model is very simple (see section 5.2.3 on page 178) but it would be interesting to examine some more "advanced" layout (possibly using tag bits that could be used by the garbage collector).

Part IV

Conclusions

Chapter 9

Experimental results

This chapter describes some experiments that we have made with the GRIN back-end, which implements the optimisation methods described in this thesis. We compare the code produced by the GRIN back-end with that produced by some other compilers, and show that our code executes several times faster. We also examine how various GRIN transformations affect the performance of the code, and make some experiments with our register allocator.

9.1 Benchmark programs

We use a set of 16 small to medium-sized Haskell programs as a basis for our measurements. The programs are described in figure 9.1 on the next page. The first 6 programs in the figure are written by us to test our compiler. The other 10 programs are taken from the nofib benchmark suite put together by Partain [Par93] (which also includes programs from Hartel [HL93]). In figure 9.1 we also give the sizes of the programs as lines of Haskell source code (and also specify how many of these are from the Haskell Prelude). The line number counts are made with all large comments removed, but the programs may still contain some single line comments, and often a blank line between function definitions.

To compile Haskell we use the GRIN back-end together with the hbcc front-end, as described earlier (see figure 1.3 on page 17). All measurements are done on SPARC machines, since that is the only target of the current GRIN back-end.

Program	Lines (Prelude)	Short description
nfib	7 (0)	Compute "*nfib* 35".
tsumupto	10 (0)	Sum a lazy list of 10 million numbers.
sieve	33 (11)	Find prime numbers using the sieve of Eratosthenes algorithm.
queens	70 (37)	Solve the queens problem, size 12.
words	181 (91)	Character manipulation, permute and compare strings.
puzzle	214 (46)	Search solutions to a rectangle puzzle.
tak	10 (0)	Tak function, compute "*tak* 24 16 8" (nofib).
exp3_8	33 (0)	Compute 3^8 using natural numbers (nofib).
awards	150 (53)	Compute scores and assign awards to competitors (nofib).
sorting	208 (30)	Sort using eight different sorting algorithms (nofib).
cichelli	224 (59)	Calculate a perfect hashing function (nofib).
event	225 (50)	Event driven simulation of an electronic circuit (nofib/Hartel).
clausify	300 (69)	Reduce propositions to clausal forms (nofib).
ida	328 (57)	Solve the n-puzzle using iterative deepening (nofib/Hartel).
typecheck	507 (51)	Polymorphic type checking (nofib/Hartel).
boyer2	716 (3)	Gabriel suite "boyer" benchmark (nofib).

Figure 9.1: The benchmark programs.

The nofib benchmark programs have been slightly modified to better suit our experiments. The important changes are:

- Each program has been put in a single Haskell source file, to give all compilers (ours, ghc and hbc) the same chance to optimise the entire program at once.
- All uses of Haskell overloading (via type classes) have been eliminated.

We eliminate overloading to avoid differences in the handling of overloading among the different compiler front-ends, since that is typically something which can have a huge impact on execution time (we want to benchmark the compiler back-ends, not the front-ends). Note also that both the above changes should benefit all the tested compilers. The absence of overloading means that we use only "normal functions" from the Haskell Prelude, no type classes.

All the benchmark programs, as well as the raw results from all measurements presented in this chapter, can be downloaded from the WWW.[1]

To measure performance we use both timings and dynamic instruction counts. All instruction counts are made using the *Shade* and *Spixtools* instruction level profiling tools [CK94].

9.2 Execution speed

We have compared the execution speed of the code produced by the GRIN back-end with that of code produced by the two major Haskell compilers, the Glasgow ghc compiler [PJHH+93] and the Chalmers hbc compiler [AJ89c]. We have compiled the same programs with the three compilers (modulo the Prelude functions, which each compiler provides), and use compiler flags for full optimisation according to the hbc and ghc documentation. We execute the resulting binaries with the same heap size on a Sun UltraSPARC machine.

The results are shown in figure 9.2 on the next page (instruction counts) and in figure 9.3 on page 279 (timings).

As with all benchmark results, ours should be taken with a large grain of salt. Ideally, we would compare only the compiler back-ends, but in practice that is not possible. So, when interpreting the results the reader should take in mind that front-end issues may cater for some of the differences. With regards to front-end optimisations, ghc is probably the strongest among the three, with hbc being the weakest, and hbcc somewhere in the middle. On the other hand, our manual elimination of overloading in the benchmark programs will hopefully make the front-end a bit less important.

The instruction counts and timings give very consistent results, and show that the GRIN back-end produces code that is several times faster than the other compilers. It is easy to calculate an average of the speedup factors (to be

[1] URL: `http://www.cs.chalmers.se/~boquist/phd/benchmarks.tar.gz`.

	Instruction counts			Speedup	
	GRIN	ghc[a]	hbc[b]	$\frac{\text{ghc}}{\text{GRIN}}$	$\frac{\text{hbc}}{\text{GRIN}}$
nfib	343,401,150	724,103,969	1,464,935,563	2.1	4.3
tsumupto	70,003,567	919,528,731	1,287,756,238	13.1	18.4
sieve	445,471,996	2,739,402,544	1,346,790,492	6.1	3.0
queens	1,351,158,429	7,837,488,116	6,270,156,262	5.8	4.6
words	215,496,559	1,276,432,886	782,473,011	5.9	3.6
puzzle	1,692,269,593	4,198,491,460	4,253,475,269	2.5	2.5
tak	36,155,800	58,649,124	150,474,057	1.6	4.2
exp3_8	267,097,385	708,485,203	575,164,670	2.7	2.2
awards	195,862	614,995	1,048,362	3.1	5.4
sorting	289,375,168	1,308,814,805	1,785,522,597	4.5	6.2
cichelli	16,353,000	63,840,403	47,462,579	3.9	2.9
event	73,519,679	248,087,123	173,689,714	3.4	2.4
clausify	111,195,338	174,846,366	259,989,602	1.6	2.3
ida	26,579,902	112,109,335	1,599,434,352	4.2	60.2
typecheck	161,367,525	331,543,306	475,869,269	2.1	2.9
boyer2	13,412,809	37,399,209	36,382,598	2.8	2.7

[a] Using ghc version 4.01 with flags "`-O2 -fvia-C -O2-for-C -mv8 -static`".
[b] Using hbc version 0.9999.5a with flags "`-O -msparc8 -static`".

Figure 9.2: Overall results – instruction counts.

	Running time (seconds) [a]			Speedup	
	GRIN	ghc	hbc	$\frac{\text{ghc}}{\text{GRIN}}$	$\frac{\text{hbc}}{\text{GRIN}}$
nfib	3.50	5.89	14.04	1.7	4.0
tsumupto	0.21	7.95	12.86	37.9	61.2
sieve	9.13	30.52	20.38	3.3	2.2
queens	11.56	88.08	63.13	7.6	5.5
words	2.28	14.02	7.33	6.1	3.2
puzzle	16.11	43.61	43.02	2.7	2.7
tak	0.28	0.47	1.30	1.7	4.6
exp3_8	2.48	7.12	6.57	2.9	2.6
awards	0.01	0.01	0.01	-	-
sorting	2.81	13.22	20.16	4.7	7.2
cichelli	0.16	0.66	0.48	4.1	3.0
event	0.85	2.48	1.89	2.9	2.2
clausify	1.25	1.84	2.74	1.5	2.2
ida	0.30	1.16	17.02	3.9	56.7
typecheck	1.99	3.99	5.23	2.0	2.6
boyer2	0.16	0.42	0.39	2.6	2.4

[a] All times are user times, averaged over 5 runs on an otherwise idle UltraSPARC-I cpu (140MHz). We measured also on a faster machine, an UltraSPARC-II (300MHz) with more or less identical results (the speedup factors).

Figure 9.3: Overall results – timings.

fair we remove the largest and smallest factors, and also ignore hbc's result on the ida benchmark). If we do this, the GRIN back-end is 3.6 times faster than ghc (on both instruction counts and timings), and 3.6 times faster than hbc on instruction counts (3.5 on timings).

Given the experimental state of the current GRIN implementation, and the fact that we have not "tuned" our algorithms on a particularly large set of programs, these results must be considered very satisfactory.

The close agreement between the timings and the instruction counts may actually be somewhat surprising to some. Normally you would expect the speedup factors in the timings to be smaller than in the instruction counts, due to the instruction level parallelism available in modern hardware. However, we believe that this to a large extent can be explained by differences in the basic evaluation model for the different implementations. In the GRIN back-end we have put much effort into using a model that we believe works well on modern hardware, e.g., to use conditional branches and direct function calls rather than loads and indirect calls. The typical example of this is our inlining of the *eval* procedure. With a small number of tags in the resulting case expression our back-end will compile an *eval* into a single or a few conditional tests and branches (possibly followed by calls to known functions, or by inlined function bodies). Combined with the fact that we use registers a lot (for example to pass all function arguments and results), this leads to very efficient code. Evaluation in ghc and hbc is done via one or more dependent memory loads followed by an indirect call.

The cost of using memory is clearly seen if we compare the speedup factors in the instruction counts and the timings for the tsumupto benchmark. This program is compiled by our back-end to code that does not use memory at all (see appendix A), leading to much higher speedups in the timings than in the instruction counts.

9.3 Program size and compile time

Given the program-wide optimisation done by the GRIN back-end it is relevant to see how program size affects compile times. The "program size" for some points during the compilation process are shown in figure 9.4 on the next page. This should be related to the time spent in different compiler phases shown in figure 9.5 on page 282. Figure 9.5 also shows the total compile time for our benchmark programs. The result for the typecheck benchmark should be disregarded, the long register allocation time is due to a known bug in our implementation (that could be fixed relatively easy, but we have not yet had time to do that).

Below we will analyse each major phase of our back-end regarding its compile time.

	Haskell	λ^a	GRIN^b	GRIN^c	GRIN^d	RISC^e	RISC^f
nfib	7	20	33	27	24	94	64
tsumupto	10	27	43	45	21	86	44
sieve	33	60	81	114	104	436	301
queens	70	131	168	226	220	884	621
words	181	689	1,225	1,968	1,661	5,964	4,154
puzzle	214	594	798	1,301	1,463	6,153	4,454
tak	10	20	34	28	27	127	92
exp3_8	33	49	72	108	162	634	490
awards	150	433	654	1,053	1,190	4,706	3,704
sorting	208	859	1,062	1,945	2,259	9,295	6,805
cichelli	224	639	1,049	1,631	1,771	7,014	5,173
event	225	665	940	1,730	2,723	11,060	9,129
clausify	300	810	1,191	2,066	2,345	9,429	6,829
ida	328	973	1,242	2,239	2,914	12,204	9,097
typecheck	507	1,610	2,551	4,164	5,339	22,557	20,804
boyer2	716	894	2,642	15,882	9,234	29,899	34,184

[a] Functional intermediate code (see figure 2.10 on page 49)
[b] Initial GRIN code.
[c] After *eval* inlining.
[d] Final GRIN code.
[e] Initial RISC code.
[f] Final RISC code.

Figure 9.4: Program sizes at various compiler phases (line number counts).

	Control flow analysis	GRIN transform	Register allocation	RISC opt.	Total [a]
nfib	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
tsumupto	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
sieve	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0:01
queens	0 (0%)	0 (0%)	0:01 (0%)	0 (0%)	0:03
words	0:01 (2%)	0:07 (15%)	0:20 (42%)	0:16 (33%)	0:48
puzzle	0 (0%)	0:05 (6%)	1:03 (70%)	0:17 (19%)	1:30
tak	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
exp3_8	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0:01
awards	0 (0%)	0:05 (16%)	0:14 (45%)	0:10 (32%)	0:31
sorting	0 (0%)	0:08 (6%)	1:21 (60%)	0:41 (30%)	2:16
cichelli	0:01 (1%)	0:07 (9%)	0:49 (60%)	0:21 (26%)	1:22
event	0 (0%)	0:08 (4%)	2:45 (74%)	0:44 (20%)	3:43
clausify	0:03 (2%)	0:07 (4%)	1:44 (67%)	0:37 (24%)	2:36
ida	0 (0%)	0:11 (5%)	2:07 (63%)	0:55 (27%)	3:23
typecheck	0:02 (0%)	0:25 (3%)	13:03 (79%)	2:42 (16%)	16:32[b]
boyer2	0:28[c](8%)	1:07 (20%)	2:49 (51%)	0:55 (17%)	5:31

[a] All times are user times taken on a Pentium II 450MHz running NetBSD.

[b] This result is totally bogus. It is due to a bug in our implementation resulting in an unnecessarily large number of iterations of the build-colour cycle in the register allocator.

[c] This anomaly is due to a very large "constant" data structure in the program. Constant data could easily be treated efficiently in the analysis, but is currently not.

Figure 9.5: Compile times (min:sec).

Control flow analysis

From the compile times we can see that the time spent in the GRIN control flow analysis (see chapter 3) is nearly not measurable (in seconds), except for one anomaly, the boyer2 program. However, the reason for this is a huge completely constant data structure that participates in the iteration. This is of course unnecessary since the data structure and all memory it will occupy is already completely determined. Situations like this could easily be worked-around in the implementation of the analysis, but that is currently not done.

GRIN transformations

The time taken for GRIN transformations is also a small part of the total time, and we do not expect that to change for larger programs. Most GRIN transformations are global (done per procedure), and the ones that use interprocedural information are never more expensive than time linear in the size of the program. The GRIN transformations for the boyer2 benchmark suffers from the same problem as in the analysis above (a huge constant data structure that unnecessarily takes part of the optimisation).

Register allocation

As the program size is increased the time spent in register allocation becomes a larger and larger part of the total time. From our experiments it is clear that the register allocation currently is the limiting factor on how large programs we can compile (although that is not really shown in figure 9.5). This is not surprising, since the current implementation uses program-wide graph colouring (the *base* algorithm described in chapter 6). However, we do not believe that any definite conclusions about the register allocation algorithm should be drawn from these measurements, for the following reasons:

- the current implementation is extremely inefficient, written in Haskell using arrays without update-in-place (which is really crucial to implement graph colouring efficiently),
- the main efficiency problem with the current implementation is space rather than time, i.e., the peak amount of memory it needs. In an efficient implementation of graph colouring, memory usage should not be the most important factor,
- the maximal graph size that we can handle is not particularly large (a couple of thousand nodes, see figure 9.8 on page 289). This is not very large even when compared to global register allocators (which sometimes are faced with very big procedures).

With an efficient implementation, Chaitin-style graph colouring in practice has a complexity near $O(n \log n)$, where n is the number of nodes in the conflict

graph (see [Bri92]). For really large programs even this may become a problem, which is why we proposed our *extended* register allocation algorithm (see section 6.3 on page 233). It offers a simple tradeoff between allocation quality and compile time. So, although we have not yet proved it, we believe that the register allocation can be fixed to not be a problem when it comes to handling real programs.

RISC optimisations

The time spent in RISC optimisation may seem almost as large as register allocation time, but this would almost certainly change for larger programs. All RISC optimisations are global and only a few use some interprocedural (linear time) information. So, the RISC optimisations should not be a problem for larger programs.

Total compile time

To summarise the discussion about compile time, we believe that with two changes to the current implementation, our back-end ought to scale to large programs:

1. implement the *extended* register allocation algorithm, and
2. implement the actual graph colouring efficiently.

9.4 The control flow analysis

We have done a simple experiment to illustrate the result of the control flow analysis, which is used to perform the GRIN *eval* inlining transformation (see section 4.2.1 on page 84). Figure 9.6 on the next page shows the maximal and average sizes of the case expressions that are the results of inlined calls to *eval*. The figure also shows the number of iterations that the analysis required (see section 3.2 on page 67). We can see that the average size of *eval* case expressions is very low. In fact, most case expressions become very small and there are only a few large ones (but not extremely large). The largest can hopefully be reduced in size by more aggressive use of function specialisation and cloning (see the discussion in section 10.2.1 on page 296).

	evals	*eval* size[a]		Analysis iterations
		max	avr.[b]	
nfib	1	1	1.0	3
tsumupto	3	1	1.0	4
sieve	9	2	1.6	4
queens	15	3	1.7	5
words	115	7	1.9	30
puzzle	76	14	2.6	16
tak	1	1	1.0	3
exp3_8	6	5	3.4	5
awards	64	12	2.5	17
sorting	145	6	2.4	9
cichelli	95	11	2.4	28
event	95	18	4.3	10
clausify	116	7	2.7	45
ida	163	14	2.6	9
typecheck	217	40[c]	4.0	16
boyer2	133	14	2.9	114[d]

[a] The number of tags in case expressions resulting from unfolded calls to *eval*.
[b] Arithmetic mean.
[c] The next largest in this case is 21.
[d] As discussed earlier this number could easily be reduced.

Figure 9.6: Result of *eval* inlining (size of *eval* case expressions).

9.5 GRIN program transformations

We have tried to illustrate the relative importance of the different GRIN transformations in figure 9.7 on the facing page. Each column denotes the performance with a single transformation turned off. Each number is given relative to a run with all optimisations turned on. E.g., "1.03" means a 3% slowdown.

Not all GRIN optimisations are shown in figure 9.7, and there are two reasons for this. First, no measurable difference (less than 1%) were observed for the following transformations: *evaluated case elimination*, *case copy propagation*, *unboxing of return values* (which is not strange since the hbcc front-end already achieves much unboxing), *update elimination* and *dead parameter elimination*. Second, it was not possible to turn off some GRIN optimisations (the GRIN implementation is not as robust as it should be), and we were not able to measure code without those transformations. These were: *copy propagation*, *constant propagation* and *dead code elimination*.

The most important conclusion from figure 9.7 is probably that no single GRIN transformation is extremely important. This is a good sign, and fits well with experiences from other compilers: to get good performance in practice a large number of different optimisations must be implemented.

The transformation with the largest slowdowns in figure 9.7 is the *whnf update elimination* (see section 4.3.9 on page 133). The conclusion from this is that *updates* are important, and when done unnecessarily can attribute a large extra cost.

Some numbers also indicate a small speedup (a number less than 1). This is probably due to better register allocation (many GRIN transformation increase the register pressure). Unfortunately there is not much we can do about this. The whole optimisation process done by the back-end is so complex, and contain NP-complete parts, that there is no chance that we can ever guarantee that every single optimisation is always a win.

The transformation of the tsumupto benchmark is shown in its entirety in appendix A, and its "success" depends heavily on a few GRIN transformations working together: the *late inlining*, the *case hoisting* and the *arity raising*. From figure 9.7 we can see that as soon as one of those transformations is omitted, the program runs much slower. In fact, the tsumupto in figure 9.7 is not exactly the one that is in appendix A, it has already been unboxed by the front-end (the example in appendix A is completely boxed in the beginning). With a fully boxed version, the differences would have been even greater.

	Relative instruction counts							
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*
nfib	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
tsumupto	1.00	1.00	1.00	3.71	2.00	3.00	2.29	1.00
sieve	1.00	1.00	1.00	1.00	1.03	1.00	1.00	1.00
queens	1.02	1.00	1.00	1.01	1.01	1.00	1.20	0.97
words	1.00	1.00	1.00	1.03	1.00	1.00	1.00	1.00
puzzle	1.01	1.02	1.14	1.02	1.01	1.00	1.00	1.02
tak	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
exp3_8	1.00	1.00	1.00	1.00	1.03	1.00	1.00	1.09
awards	1.00	1.01	1.03	1.03	1.01	1.00	1.00	1.01
sorting	1.00	1.00	1.00	1.03	1.00	1.00	1.01	1.04
cichelli	1.03	1.00	1.22	1.06	1.02	1.00	1.00	1.04
event	1.00	1.00	1.04	1.06	1.02	1.00	1.00	0.99
clausify	1.01	1.00	1.00	1.01	0.99	1.00	0.98	1.02
ida	1.00	1.00	1.08	1.00	1.01	1.00	1.00	1.03
typecheck	-	1.01	1.03	1.00	1.01	1.00	1.00	1.00
boyer2	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.02

[a] Skip GRIN *sparse case optimisation* (see section 4.3.6 on page 127).
[b] Skip GRIN *trivial case elimination* (see section 4.3.5 on page 126).
[c] Skip GRIN *whnf update elimination* (see section 4.3.9 on page 133).
[d] Skip GRIN *late inlining* (see section 4.3.10 on page 135).
[e] Skip GRIN *common sub-expression elimination* (see section 4.3.14 on page 148).
[f] Skip GRIN *case hoisting* (see section 4.3.11 on page 137).
[g] Skip GRIN *arity raising* (see section 4.3.13 on page 144).
[h] Skip GRIN *right hoist fetch operations* (see section 4.2.8 on page 107).

Figure 9.7: Performance with a single GRIN transformation disabled.

9.6 Register allocation

We have also made some experiments with the register allocator (the *base* algorithm described in chapter 6).

Graph size

The sizes of the interference graphs for the different programs are shown in figure 9.8 on the facing page. For each program we show the size of the initial graph and the size of the graph after the first round of the *coalesce* phase (conservative coalescing, see section 6.2.5 on page 214). The number of nodes in the initial graph corresponds exactly to the number of virtual registers used by the program, but is probably not a very good measure of the "real graph size", since it includes a lot of "unnecessary" live ranges created by the RISC code generator. A better measure of the size of the graph is the size after the first round of coalesce, when most unnecessary live ranges (and copies) have been removed.

In figure 9.8 we also show the number of iterations through the *build-colour cycle* (see figure 6.2 on page 200) that is required to colour the graph. In most cases a very small number of iterations is enough. The exception is the result for the typecheck benchmark, but as mentioned earlier this is due to a bug in our implementation.

Coalesce restrictions

As a complement to the discussion in section 6.2.5 on page 214, we have also measured the effect of different coalesce restrictions. The result is shown in figure 9.9 on page 290. We have compiled with three different restrictions. In the first case we always use *conservative coalescing* (see section 6.2.5 on page 217). We compare this to using conservative coalescing during the first iteration of the build-colour cycle (when most coalesces are made) and then no restriction at all after that. Finally we also include a case with no restriction on coalesce at all. For each case we show the number of coalesces made during the first iteration of the build-colour cycle and the number of coalesces made after that. We also give the total number of build-colour iterations. For the latter two cases we also show the speedup of the resulting code, as instruction counts relative to the base case (always conservative coalescing). As mentioned before, the number of iterations for typecheck should be disregarded, it is due to a known bug in our implementation. We have also removed the smallest programs from the table, since they showed no differences for the three cases.

The third case, completely unrestricted coalesce, were not able to handle all the programs (either the register allocator could not colour the graph, or it produced incorrect code which made the resulting program crash). These are marked with a "-" in the table.

	Procs[a]	Initial graph		After 1st coalesce		Iterations
		Nodes	Edges	Nodes	Edges	
nfib	2	15	9	6	6	1
tsumupto	2	18	28	8	11	1
sieve	5	92	314	39	137	1
queens	10	176	576	57	214	1
words	46	940	10,626	438	5,921	2
puzzle	49	1,130	30,858	935	28,980	2
tak	2	21	46	9	18	1
exp3_8	6	95	269	28	69	1
awards	32	760	7,568	333	4,016	2
sorting	66	1,688	24,921	809	17,254	4
cichelli	52	1,252	18,117	728	13,350	4
event	52	1,818	46,980	1,623	45,559	2
clausify	52	1,407	40,183	1,069	37,388	3
ida	73	2,200	35,802	1,677	31,630	2
typecheck	126	3,886	124,048	3,121	117,079	14 [b]
boyer2	51	1,382	35,956	1,055	31,849	2

[a] Number of procedures in the RISC code.
[b] This high number is due to a bug in our implementation (see previous discussion).

Figure 9.8: Register allocation graph size.

	1: conservative[a]		2: conservative 1st[b]			3: no restriction[c]		
	#coal.[d]	#[e]	#coal.	#	tot[f]	#coal.	#	tot
words	502+120	2	502+133	2	0.99	628+16	4	-
puzzle	200+493	2	200+637	2	0.88	796+157	5	-
awards	429+5	2	429+94	2	0.86	522+3	2	0.85
sorting	886+66	2	886+328	4	0.94	1,222+56	5	0.93
cichelli	526+66	2	526+346	4	0.88	844+87	4	0.95
event	195+757	2	195+1,170	2	0.90	1,283+66	3	0.89
clausify	338+604	2	338+791	3	0.96	1,060+221	5	-
ida	523+218	2	523+1,021	2	0.83	1,547+9	3	0.83
typecheck	768+1,569	2	768+1,969	14[g]	0.91	-	-	-
boyer2	328+529	2	328+684	2	0.94	952+322	5	-

[a] Always use conservative coalescing (see section 6.2.5 on page 214).
[b] Restrict coalesce only first iteration through the build-colour cycle.
[c] No restriction on coalesce.
[d] Number of coalesces made (1st build-colour iteration + the rest).
[e] Number of iterations through the build-colour cycle.
[f] Instruction count compared to first column. E.g., 0.94 means a 6% speedup.
[g] This strange result is due to a bug in our implementation (see previous discussion).

Figure 9.9: Different coalesce restrictions.

From figure 9.9 a number of observations can be made. If we compare the number of coalesces made in the first build-colour round, the first two cases ("1" and "2") are always equal, which is to be expected since they are both conservative coalescing, whereas the third case makes many more coalesces during the first round. If we compare the number of coalesces after the first round, we can see that as soon as the conservative restriction is dropped the number of coalesces increase ("2" has larger counts than "1").

We can also see that the performance of the code produced by "2" is always better than that of "1". Case "3" is sometimes better and sometimes worse than "2", for programs where it works at all. It is also possible that for larger programs, even our case "2" could run into similar problems as "3", resulting in uncolourable graphs.

In any case, the measurements clearly show that conservative coalescing in itself is far too restrictive, and leaves too many copies in the code. Significant speedups can be obtained by weakening the coalesce restrictions further. Unfor-

tunately it is also easy to do too many coalesces, and end up with graphs that are harder (or impossible) to colour. This is exactly the experiences reported in [GA96a], where also a more powerful coalesce method is presented (*iterated coalescing*, see section 6.2.5 on page 217).

Spilling

To see how our allocator performs under higher register pressure we have also examined the code produced with gradually fewer machine registers (colours) available. These results are shown in figure 9.10 on the following page, for three of the benchmark programs. Since the main effect of increased spilling is that values are written and read from stack memory rather than being kept in registers, we report the number of stack loads and stores as the number of registers decreases. Each number is given relative to the base case, of 20 registers. The total number of instructions are not included for space reasons, except as the total slowdown when going from 20 to 7 registers.

Of course it should be possible to go even below 7 registers, and still produce runnable code, but our implementation is not very good at handling such situations. As an example, it currently must pass all function arguments in registers, so it needs at least that many registers to work. This would be easy to fix though.

Shrink-wrapping

The final measurement on the register allocator concerns the shrink-wrapping algorithm used by the *save locals* phase (see section 6.2.3 on page 202). In particular, we examine if the shrink-wrapping should be applied to saves and restores of the return address register. As discussed earlier (page 209) this is not always obvious. It may or may not be a win to include the return address register in the shrink-wrapping depending on the dynamic execution characteristics of the program. Although the differences are not large, the measurements in figure 9.11 on page 293 confirm this, we get both speedups and slowdowns when turning off the return address shrink-wrapping (the default is to include it).

Unfortunately the implementation can not turn off shrink-wrapping for all registers, which would also be an interesting measurement.

	sorting		cichelli		boyer2	
#r	Sp load	Sp store	Sp load	Sp store	Sp load	Sp store
20	12,978,250	11,800,815	840,382	1,314,335	1,151,617	1,077,542
19	+14,402	+1	+22,790	+4,859	+122,814	+117,977
18	+1,042,804	+498,393	+763,580	+493,196	+122,814	+117,977
17	+1,057,206	+505,594	+755,982	+485,598	+199,088	+233,539
16	+5,402,802	+4,846,389	+809,753	+520,830	+201,024	+236,050
15	+13,424,504	+9,687,987	+794,649	+505,746	+465,368	+369,551
14	+27,403,302	+14,519,982	+285,177	+251,416	+770,377	+547,860
13	+27,410,303	+14,524,382	+802,295	+513,382	+468,360	+371,641
12	+27,415,905	+14,526,383	+810,485	+520,972	+786,280	+588,950
11	+27,417,507	+14,527,984	+842,935	+553,284	+804,612	+595,551
10	+27,421,909	+14,529,585	+1,128,538	+728,965	+788,790	+590,205
9	+27,425,049	+14,532,724	+1,317,809	+917,702	+805,867	+597,487
8	+28,699,715	+16,994,239	+1,317,809	+917,702	+978,215	+687,281
7	+33,444,411	+19,314,078	-	-	+993,050	+693,028
Total icount:		+18%		+9%		+11%

Figure 9.10: Forced spilling – decreasing available registers.

	Shrink-wrap $\%o_7$	
	Yes	No[a]
nfib	1.00	1.09
tsumupto	1.00	1.00
sieve	1.00	1.00
queens	1.00	0.99
words	1.00	0.99
puzzle	1.00	1.00
tak	1.00	1.07
exp3_8	1.00	1.03
awards	1.00	0.98
sorting	1.00	1.02
cichelli	1.00	0.99
event	1.00	1.01
clausify	1.00	1.01
ida	1.00	1.00
typecheck	1.00	0.99
boyer2	1.00	1.01

[a]The numbers denote instructions counts relative to the "Yes" column.

Figure 9.11: Shrink-wrapping – with and without the return address register.

Chapter 10

Conclusions

This final chapter will first summarise very shortly the contributions made in this thesis, and then discuss some issues that need further research.

10.1 Results

We have described a complete and in many ways novel back-end for lazy functional languages. The back-end uses a large number of optimisations (of which some are interprocedural) to produce greatly optimised code. We have introduced the GRIN language, an intermediate language well suited for analysis and optimisation. The GRIN language offers a convenient compromise between a high level function intermediate code and a low level imperative code. It provides the low level control over data structures, sequencing and evaluation that is needed for many low level optimisations, but is still "functional enough" to be a good basis for program transformation.

We have shown how to use a program-wide control flow analysis combined with a GRIN program transformation to eliminate much of the unknown control flow (indirect function calls) that is normally present in lazy functional programs due to the evaluation of suspended computations. A large number of GRIN source-to-source program transformations are used to simplify and optimise the program.

On the lower level, called RISC, we have developed an interprocedural register allocation algorithm to reduce the function call and return overhead, something which is vital in an implementation of a functional language. In addition, a number of RISC optimisations are used to produce code of high quality.

A more detailed summary of the main contributions in this thesis can be found in section 1.5 on page 21 ("Summary of research contributions").

With the experimental results presented in chapter 9 we have shown that our approach is viable, and in particular that the code produced by our back-

end is of very good quality. Compared to two other Haskell compilers, the Glasgow ghc compiler and the Chalmers hbc compiler, the code we produce executes several times faster (see figure 9.3 on page 279). We have also argued that the only remaining issue preventing our back-end from being used on really large programs is the current register allocation algorithm, and we have given a detailed proposal on how this problem should be solved (the *extended* register allocation algorithm, see section 6.3 on page 233).

So to summarise, we believe that the GRIN back-end and the optimisation methods presented in this thesis are both practically useful, and produce very good results (fast code).

10.2 Discussion and further work

Having said the above, the current implementation is far from being practically useful. There are a number of "minor problems" that needs to be fixed first. This section will discuss some of these. There are also a few issues where it is uncertain what will happen for really large programs, and we discuss ways to handle also these potential problems, in case they should arise.

10.2.1 The GRIN intermediate code

There are a few potential problems, of varying degree, within the GRIN language itself and the way we use GRIN to implement graph reduction. One example of this is the way we currently eliminate all calls to *eval*, using the *eval* inlining transformation (see section 4.2.1 on page 84). It is still uncertain how this approach scales to really big programs, and it could potentially lead to problems with both compile time and runtime efficiency. The main issue regarding compile time is the program-wide control flow analysis that we use (see chapter 3). Fortunately, our experiments sofar indicate that the analysis is very fast in practice (see figure 9.5 on page 282), but measurements on larger programs are needed to be absolutely sure that it scales. The main issue when it comes to runtime efficiency, i.e., the execution speed of the generated code, is dependent on the result that the control flow analysis delivers. For each call to *eval* the analysis will give a set of possible GRIN tags that can appear at that *eval* point. When calls to *eval* are unfolded this set of tags will result in a GRIN case expression. It would be unfortunate if the number of tags in these case expressions exploded as the program size went up (since large case expressions can be expensive to implement, see below). Fortunately, from our experiments sofar it seems like that is not the case. In fact, the number of tags in case expressions resulting from unfolded calls to *eval* seems almost independent of program size (see figure 9.6 on page 285). The exception to this is functions from the Haskell Prelude, which may have very many *uses* in a program, possibly leading to a large number of possible closures for some calls to *eval*. However, even if "tag

explosion" should turn out to be a problem, there are several ways in which it can be attacked. One way is more aggressive use of function specialisation (both of types and of higher order functions), and cloning. This can hopefully solve the Prelude problem to a large extent (specialisation and cloning will cut down the number of tags for each *eval* point). Another way is to not unfold all calls to *eval*, and instead implement a built-in GRIN operation for evaluation and use that instead. Related to this is also the ability to have *default alternatives* in GRIN case expressions (which we need to add for several reasons). That would make it possible to handle a very large number of possible tags for a certain *eval*, and only keep the "most probable" as explicit alternatives (for example using profiling information).

An issue that is closely related to the handling of *eval* is how *updates* are performed. Currently we require the node value written by a GRIN `update` operation to be known at compile time. But this is not an ideal solution, in some situations it forces us to insert a scrutinising case expression to be able to perform the update (see section 4.2.3 on page 95). However, this problem could be solved in a similar way to *eval* above, by implementing a more general `update` operation that does not need to know an exact tag at compile time.

The implementation of case expressions is very important in the GRIN approach, since due to the *eval* inlining our code will have a lot more (visible) case expressions in it than those present in the original source. The GRIN case expressions can potentially also become very large. Moreover, we require all GRIN tags to be uniquely numbered, which means that the interval of tag numbers appearing in a certain case expression can be very *sparse* (compare this to a traditional implementation, e.g., the G-machine, where case expressions use very small tag numbers, only unique within each datatype). Fortunately, there are a lot of previous work in this area (implementing sparse case expressions, see for example [Sal81, Spu94]). Since we have control over the tag number assignment in GRIN it should also be possible to optimise that with regard to the case expressions that actually appear in the code (see the discussion in section 4.4.3 on page 159).

A very interesting experiment would be to try to isolate the effects of our basic *evaluation model*, i.e., the way we force the evaluation of closures using the *eval* procedure. Effectively, after the control flow analysis and the unfolding of all calls to *eval*, evaluation is done via (hopefully) small case expressions and calls to known functions (or inlined function bodies). This should be compared to the traditional way of implementing evaluation using *indirect* calls (jumps to a compile time unknown function). Variations of this latter technique are used in both the G-machine [Joh84], where a *tag* is a pointer to a dispatch table, and in the STG-machine [PJ92], where a tag is a pointer to the evaluation code itself. It is our belief that our method is better suited for modern computer architectures, where memory loads and indirect calls can be quite expensive operations. Unfortunately, it is very difficult to perform such an experiment in practice, since the evaluation design decision affects so many other things in

the implementation. One possible way to do an experiment could be to use the GRIN back-end as a base, and then skip the unfolding of calls to *eval* (we could still keep the control flow analysis, for the benefit of the register allocator). This would mean that all calls to *eval* would be kept in the code as calls to a built-in runtime system procedure. This could then be implemented via some dispatch table technique, like in the G-machine.

Currently, GRIN is an untyped language. Since typed languages have many advantages, it would be interesting to find a suitable type system for GRIN. However, it is not obvious how such a system would look, or if a type system would be useful at all. One complication is the way we view GRIN *values*, and especially values representing suspended computations (so-called F-nodes). In fact, the way we currently write the *eval* and *apply* procedures (see figure 2.8 on page 45) pretty much mandates the use of some kind of *union type* containing all node values (both values representing weak head normal forms and closures). With this, the type system may be too weak to really say anything useful. On the other hand, a lot of work have been done lately on using explicitly typed intermediate languages (TILs), often based on some variant of higher order lambda calculus [PJ96, SA95, TMC+96, PJLST98]. Although these languages mostly are on a "higher level" when compared to GRIN, it might be possible to use ideas from that work. Another idea might be to use ideas from the work in [MR85, AW93], which use *subtypes* to infer types based on the usage of constructors (and thereby eliminate the need for datatype declarations).

10.2.2 Interprocedural register allocation

The current register allocator works really well, except for being a bit inefficient. But as discussed previously, this should be relatively easy to fix:

1. implement the *extended* register allocation algorithm (see section 6.3 on page 233), and
2. implement the graph colouring efficiently (the current implementation is extremely inefficient).

With that fixed, the register allocation will hopefully work even for really large programs, and produce almost as good code (if not better) as the current algorithm.

10.2.3 Program-wide optimisation

It is our strong belief that in order to get really good performance out of a language like Haskell, aggressive interprocedural optimisation is required. And we advocate that this optimisation must be done with access to the entire program.

For a lazy language like Haskell, current compilers typically compile one *module* at a time. At first sight, this might appear as a good opportunity to

optimise several procedures at once, and that is also true when it comes to many typical "front-end optimisations" like unboxing and inlining. However, such a method does not apply very well to low level optimisations, like those presented in this thesis, where the actual *dynamic control flow* is important. 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 (due to an *eval*). This means that some kind of control flow analysis must be used, to resolve the unknown control flow. Without such an analysis, many of the low level optimisations, like our register allocation, will not be able to produce any large benefits compared to normal *global* (per-procedure) optimisation.

The control flow analysis that we currently use require the entire program, and it is still uncertain how it will behave if we were to analyse only parts of the program.

Note that program-wide optimisation in itself does not have to be intractable, it is just a question of the complexity of those algorithms that are used on the entire program. E.g., many of our GRIN and RISC optimisations are mostly global (operating on one procedure at a time), but uses information collected from the entire program (in linear time).

There are also more "front-end like" reasons for doing program-wide optimisation. One of the most important is probably to eliminate the use of Haskell-style *overloading* (via type classes). Overloading is normally implemented by passing extra function arguments at runtime (called dictionaries), containing function pointers that are called instead of "real functions". This is horribly inefficient. But with access to the entire program, all overloading can be eliminated.

Also note that the kind of optimisations that we are discussing here should not be used in the everyday program development, when short compile times are essential. They should only be used when program development is nearly completed, and a lot of time can be spent on optimising the code for best possible performance.

Appendix A

GRIN transformation example

This appendix shows a complete GRIN program transformation example, demonstrating many but not all of the transformations described in chapter 4. In particular, the example shows how an inefficient lazy computation can be transformed into an efficient strict (and unboxed) computation, with effects similar to the results of *deforestation* [Wad88] and even *listlessness* [Wad84].

Unfortunately it is not possible to use a large example program and still show all transformation steps, so we will use a very small program and show all the transformation details instead. The initial functional program is shown below, written in a Haskell-like notation:

Begin program

```
main = sum 0 (upto 1 1000)
upto m n = if m > n then
             []
           else
             m : upto (m + 1) n
sum n l = case l of
            []     → n
            x : xs → sum (n + x) xs
```

End program

The program simply sums numbers using a tail recursive *sum* function. When executed in a lazy functional language, this program is a classical example of a non-strict computation involving a *producer* (the *upto* function) and a *consumer* (the *sum* function). The *sum* function is the driving force of the computation, each iteration it will consume one *cons* cell produced by the *upto*

function. Note that *upto* produces a *lazy list* of numbers, where the tail of the list is always an *upto* closure. Obviously, this is not a very efficient way to sum the first 1000 integers, at least not if we compare to how we would write such a program in an imperative language (a single "for loop").

However, as we will now show our GRIN transformations can automatically transform the above program into an equivalent program that is as efficient as the imperative program.

Using the GRIN code generation rules (see section 2.5.5 on page 48) we get the initial GRIN code for our sample program:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       store (Fsum p1 p4) ; λp5 →
       eval p5 {Fsum} ; λ(CInt n'1) →
       intPrint n'1

upto p6 p7 = eval p6 {CInt} ; λ(CInt n'2) →
             eval p7 {CInt} ; λ(CInt n'3) →
             intGT n'2 n'3 ; λb'1 →
             if b'1 then
               unit (CNil)
             else
               intAdd n'2 1 ; λn'4 →
               store (CInt n'4) ; λp8 →
               store (Fupto p8 p7) ; λp9 →
               unit (CCons p6 p9)

sum p10 p11 = eval p11 {Fupto} ; λv1 →
              case v1 of
                (CNil)           → eval p10 {CInt}
                (CCons p12 p13) → eval p10 {CInt} ; λ(CInt n'5) →
                                  eval p12 {CInt} ; λ(CInt n'6) →
                                  intAdd n'5 n'6 ; λn'7 →
                                  store (CInt n'7) ; λp14 →
                                  sum p14 p13
```

End program

To save space, we have annotated all calls to *eval* in the initial GRIN code with the result of the *heap points-to analysis* (see chapter 3). For this simple program, the result of the analysis is very exact, and gives a single possible tag for each call to *eval*. Note that the annotations describe the state *before* each

call to *eval*. Several of the calls to *eval* are annotated with a single CInt tag, which means that the argument of the *eval* is already evaluated (the pointer already points to a CInt node in the heap).

The first transformation step is to inline all calls to *eval* (see section 4.2.1 on page 84), and specialise using the annotations. The result is the following:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       store (Fsum p1 p4) ; λp5 →
       fetch p5 ; λv2 →
       (case v2 of
           (Fsum p15 p16) → sum p15 p16
       ) ; λv3 →
       unit v3 ; λ(CInt n1') →
       intPrint n1'

upto p6 p7 = fetch p6 ; λv4 →
             (case v4 of
                 (CInt a8') → unit (CInt a8')
             ) ; λv5 →
             unit v5 ; λ(CInt n2') →
             fetch p7 ; λv6 →
             (case v6 of
                 (CInt n9') → unit (CInt n9')
             ) ; λv7 →
             unit v7 ; λ(CInt n3') →
             intGT n2' n3' ; λb1' →
             if b1' then
               unit (CNil)
             else
               intAdd n2' 1 ; λn4' →
               store (CInt n4') ; λp8 →
               store (Fupto p8 p7) ; λp9 →
               unit (CCons p6 p9)
```

```
sum p10 p11 = fetch p11 ; λ v8 →
              (case v8 of
                 (Fupto p17 p18) → upto p17 p18
              ) ; λ v9 →
              unit v9 ; λ v10 →
              case v10 of
                (CNil)           → fetch p10 ; λ v11 →
                                   (case v11 of
                                      (CInt n'10) → unit (CInt n'10)
                                   ) ; λ v12 →
                                   unit v12
                (CCons p12 p13)  → fetch p10 ; λ v13 →
                                   (case v13 of
                                      (CInt n'11) → unit (CInt n'11)
                                   ) ; λ v15 →
                                   unit v15 ; λ (CInt n'5) →
                                   fetch p12 ; λ v16 →
                                   (case v16 of
                                      (CInt n'12) → unit (CInt n'12)
                                   ) ; λ v17 →
                                   unit v17 ; λ (CInt n'6) →
                                   intAdd n'5 n'6 ; λ n'7 →
                                   store (CInt n'7) ; λ p14 →
                                   sum p14 p13
```

End program

As can be seen the program expands quite a bit when all calls to *eval* are unfolded, but this is only temporary, it will soon become much smaller again. Note also that we did not insert any update operations (normally part of an unfolded *eval*, see section 4.2.1) because the analysis had deduced that none of the closures were shared.

After the *eval* inlining, we can clearly see that all the case expressions that were the result of inlined calls to *eval* are unnecessary, since each of them contain only a single alternative. This is taken care of by the *evaluated case elimination* (see section 4.3.4 on page 125) and the *trivial case elimination* (see section 4.3.5 on page 126). We replace each of the unnecessary case expressions with either a unit operation or the right hand side of the case alternative, depending on which of the transformations matches:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       store (Fsum p1 p4) ; λp5 →
       fetch p5 ; λv2 →
       unit v2 ; λ(Fsum p15 p16) →
       sum p15 p16 ; λv3 →
       unit v3 ; λ(CInt n1') →
       intPrint n1'

upto p6 p7 = fetch p6 ; λv4 →
             unit v4 ; λv5 →
             unit v5 ; λ(CInt n2') →
             fetch p7 ; λv6 →
             unit v6 ; λv7 →
             unit v7 ; λ(CInt n3') →
             intGT n2' n3' ; λb1' →
             if b1' then
                unit (CNil)
             else
                intAdd n2' 1 ; λn4' →
                store (CInt n4') ; λp8 →
                store (Fupto p8 p7) ; λp9 →
                unit (CCons p6 p9)

sum p10 p11 = fetch p11 ; λv8 →
              unit v8 ; λ(Fupto p17 p18) →
              upto p17 p18 ; λv9 →
              unit v9 ; λv10 →
              case v10 of
                 (CNil)           → fetch p10 ; λv11 →
                                    unit v11 ; λv12 →
                                    unit v12
                 (CCons p12 p13)  → fetch p10 ; λv13 →
                                    unit v13 ; λv15 →
                                    unit v15 ; λ(CInt n5') →
                                    fetch p12 ; λv16 →
                                    unit v16 ; λv17 →
                                    unit v17 ; λ(CInt n6') →
                                    intAdd n5' n6' ; λn7' →
                                    store (CInt n7') ; λp14 →
                                    sum p14 p13
```

End program

The next transformation to apply is the *unboxing of return values* (see section 4.3.3 on page 118) which will unbox the integer returned by the *sum* function. However, in this particular program that is not a very important step since *sum* is tail recursive, it will only return once. In short, the unboxing will replace the "unit v_{12}" that *sum* uses to return by:

```
unit v12 ; λ(CInt n'14) →
unit n'14
```

The tail recursive call in *sum* is not changed.

Note that this kind of unboxing does not touch function arguments, that will be done later by the *arity raising* transformation.

As a result of the previous transformations a number of unit operations have been inserted, i.e., a kind of "copies". These are eliminated by the *copy propagation* (see section 4.3.2 on page 113), resulting in:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       store (Fsum p1 p4) ; λp5 →
       fetch p5 ; λ(Fsum p15 p16) →
       sum p15 p16 ; λn'13 →
       intPrint n'13

upto p6 p7 = fetch p6 ; λ(CInt n'2) →
             fetch p7 ; λ(CInt n'3) →
             intGT n'2 n'3 ; λb'1 →
             if b'1 then
               unit (CNil)
             else
               intAdd n'2 1 ; λn'4 →
               store (CInt n'4) ; λp8 →
               store (Fupto p8 p7) ; λp9 →
               unit (CCons p6 p9)
```

```
sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
                upto p17 p18 ; λ v10 →
                case v10 of
                  (CNil)           → fetch p10 ; λ(CInt n'14) →
                                       unit n'14
                  (CCons p12 p13)  → fetch p10 ; λ(CInt n'5) →
                                     fetch p12 ; λ(CInt n'6) →
                                     intAdd n'5 n'6 ; λ n'7 →
                                     store (CInt n'7) ; λ p14 →
                                     sum p14 p13
```

End program

The next task is to simplify variables containing entire *node values* (see section 2.3.3 on page 30). In our running example, after the copy propagation, there is only one remaining node variable, "v_{10}". The simplification is done in two steps, called *vectorisation* (see section 4.2.4 on page 97) and *case simplification* (see section 4.2.5 on page 100). The resulting program is shown below (the only change is in the *sum* function).

Begin program

```
main = store (CInt 0) ; λ p1 →
       store (CInt 1) ; λ p2 →
       store (CInt 1000) ; λ p3 →
       store (Fupto p2 p3) ; λ p4 →
       store (Fsum p1 p4) ; λ p5 →
       fetch p5 ; λ(Fsum p15 p16) →
       sum p15 p16 ; λ n'13 →
       intPrint n'13

upto p6 p7 = fetch p6 ; λ(CInt n'2) →
             fetch p7 ; λ(CInt n'3) →
             intGT n'2 n'3 ; λ b'1 →
             if b'1 then
               unit (CNil)
             else
               intAdd n'2 1 ; λ n'4 →
               store (CInt n'4) ; λ p8 →
               store (Fupto p8p7) ; λ p9 →
               unit (CCons p6 p9)
```

```
sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
              upto p17 p18 ; λ(n'15 p19 p20) →
              case n'15 of
                (CNil)  → fetch p10 ; λ(CInt n'14) →
                          unit n'14
                (CCons) → fetch p10 ; λ(CInt n'5) →
                          fetch p19 ; λ(CInt n'6) →
                          intAdd n'5 n'6 ; λn'7 →
                          store (CInt n'7) ; λp14 →
                          sum p14 p20
```

End program

If we look at the function calls in the program above, it is interesting to see that there is only a single call to *upto*, done inside *sum*. Recall that the original functional source code contained two calls to *upto*, one in *main* and one recursive call in *upto*, but none of these are "real" calls. It is first when all the control flow has been made explicit, by inlining calls to *eval*, that we can see the real function calls of the program. The two calls in the original source code have instead turned into "closure building".

Since *upto* is only called once, and non-recursively, it will be inlined by the *late inlining* transformation (see section 4.3.10 on page 135). After inlining we must also apply a round of *bind normalisation* (see section 4.4.1 on page 156), resulting in the following.

Begin program

```
main = { no change }
upto p6 p7 = { no change }
sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
              fetch p17 ; λ(CInt n'16) →
              fetch p18 ; λ(CInt n'17) →
              intGT n'16 n'17 ; λb'2 →
              (if b'2 then
                  unit (CNil)
               else
                  intAdd n'16 1 ; λn'18 →
                  store (CInt n'18) ; λp21 →
                  store (Fupto p21 p18) ; λp22 →
                  unit (CCons p17 p22)
              ) ; λ(n'15 p19 p20) →
              case n'15 of
                 (CNil)   → fetch p10 ; λ(CInt n'14) →
                            unit n'14
                 (CCons) → fetch p10 ; λ(CInt n'5) →
                            fetch p19 ; λ(CInt n'6) →
                            intAdd n'5 n'6 ; λn'7 →
                            store (CInt n'7) ; λp14 →
                            sum p14 p20
```

End program

As a result of the inlining we open up possibilities for *case hoisting* (see section 4.3.11 on page 137). We can see that the value returned by the if expression inside *sum* (either a CNil or a CCons node) is immediately examined by a case expression. This is unnecessary, and the case hoisting will merge the if and the case by (temporarily) duplicating the case. The result is shown below, where we have also applied *dead code elimination* (see section 4.3.15 on page 153) to remove the now unused *upto* function.

Begin program

```
main = { no change }
sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
              fetch p17 ; λ(CInt n'16) →
              fetch p18 ; λ(CInt n'17) →
              intGT n'16 n'17 ; λb'2 →
              if b'2 then
                 unit (CNil) ; λ(n'19 p23 p24) →
                 case n'19 of
                    (CNil)  → fetch p10 ; λ(CInt n'20) →
                              unit n'20
                    (CCons) → fetch p10 ; λ(CInt n'21) →
                              fetch p23 ; λ(CInt n'22) →
                              intAdd n'21 n'22 ; λn'23 →
                              store (CInt n'23) ; λp25 →
                              sum p25 p24
              else
                 intAdd n'16 1 ; λn'18 →
                 store (CInt n'18) ; λp21 →
                 store (Fupto p21 p18) ; λp22 →
                 unit (CCons p17 p22) ; λ(n'24 p26 p27) →
                 case n'24 of
                    (CNil)  → fetch p10 ; λ(CInt n'25) →
                              unit n'25
                    (CCons) → fetch p10 ; λ(CInt n'26) →
                              fetch p26 ; λ(CInt n'27) →
                              intAdd n'26 n'27 ; λn'28 →
                              store (CInt n'28) ; λp28 →
                              sum p28 p27
```

End program

The real benefit of case hoisting does not show until we have also applied *constant propagation* (see section 4.3.12 on page 143). This will do the actual "short-circuiting" of the duplicated case expression. The result is shown below (after yet another round of copy propagation, which in this case works as a kind of *constant folding*).

Begin program

```
main = { no change }
sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
              fetch p17 ; λ(CInt n'16) →
              fetch p18 ; λ(CInt n'17) →
              intGT n'16 n'17 ; λb'2 →
              if b'2 then
                 unit () ; λp24 →
                 unit () ; λp23 →
                 fetch p10 ; λ(CInt n'20) →
                 unit n'20
              else
                 intAdd n'161 ; λn'18 →
                 store (CInt n'18) ; λp21 →
                 store (Fupto p21 p18) ; λp22 →
                 fetch p10 ; λ(CInt n'26) →
                 fetch p17 ; λ(CInt n'27) →
                 intAdd n'26 n'27 ; λn'28 →
                 store (CInt n'28) ; λp28 →
                 sum p28 p22
```

End program

After the constant propagation the code duplication that was temporarily introduced by the case hoisting has disappeared.

If we now look back at the *main* function, there is an operation "fetch p_5" that is used to load a value from the heap that has just been written by the previous operation (a store). This can be optimised by the *common sub-expression elimination* (see section 4.3.14 on page 148). We regard the fetch and the preceding store as common sub-expressions. The CSE will replace the fetch with "unit (Fsum p_1 p_4)", i.e., the same node written by the store operation. As a result, the store becomes dead and can be removed by the dead code elimination.

The CSE will also find that "fetch p_{17}" is done twice in the *sum* function, and will replace the second fetch with the result of the first, i.e., "unit (CInt n'_{16})". Both unit operations inserted by the CSE will be removed by the next copy propagation.

Additionally there are two "unit ()" inside *sum* that can be removed by the dead code elimination. After this the program will look like:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       sum p1 p4 ; λn'13 →
       intPrint n'13

sum p10 p11 = fetch p11 ; λ(Fupto p17 p18) →
              fetch p17 ; λ(CInt n'16) →
              fetch p18 ; λ(CInt n'17) →
              intGT n'16 n'17 ; λb'2 →
              if b'2 then
                 fetch p10 ; λ(CInt n'20) →
                 unit n'20
              else
                 intAdd n'16 1 ; λn'18 →
                 store (CInt n'18) ; λp21 →
                 store (Fupto p21 p18) ; λp22 →
                 fetch p10 ; λ(CInt n'26) →
                 intAdd n'26 n'16 ; λn'28 →
                 store (CInt n'28) ; λp28 →
                 sum p28 p22
```

End program

If we study the program at this point we can see something very interesting. Even though the *upto* function no longer exists, we still store and fetch *upto* closures (using the Fupto tag). In fact, the *sum* function uses an *upto* closure to "communicate" with itself, it writes a closure right before doing the recursive call and then immediately loads the value back after the call. This seems very strange. Instead of passing a pointer argument (the second argument of *sum*) we should be able to pass the two arguments of the *upto* closure directly. This is exactly what the *arity raising* transformation will do (see section 4.3.13 on page 144). In fact, the arity raising will also apply to the first argument of *sum*, which will be "raised" to an unboxed integer. The fetch operations that load raised arguments will be replaced by unit operations:

Begin program

```
main = store (CInt 0) ; λp1 →
       store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       store (Fupto p2 p3) ; λp4 →
       sum 0 p2 p3 ; λn'13 →
       intPrint n'13
```

```
sum n'29 p29 p30 = unit (Fupto p29 p30) ; λ(Fupto p17 p18) →
                   fetch p17 ; λ(CInt n'16) →
                   fetch p18 ; λ(CInt n'17) →
                   intGT n'16 n'17 ; λb'2 →
                   if b'2 then
                      unit (CInt n'29) ; λ(CInt n'20) →
                      unit n'20
                   else
                      intAdd n'16 1 ; λn'18 →
                      store (CInt n'18) ; λp21 →
                      store (Fupto p21 p18) ; λp22 →
                      unit (CInt n'29) ; λ(CInt n'26) →
                      intAdd n'26 n'16 ; λn'28 →
                      store (CInt n'28) ; λp28 →
                      sum n'28 p21 p18
```

End program

As a result of the arity raising some of the store operations are now dead and can be removed by the dead code elimination. An additional copy propagation will also remove some unit operations, and we get:

Begin program

```
main = store (CInt 1) ; λp2 →
       store (CInt 1000) ; λp3 →
       sum 0 p2 p3 ; λn'13 →
       intPrint n'13
sum n'29 p29 p30 = fetch p29 ; λ(CInt n'16) →
                   fetch p30 ; λ(CInt n'17) →
                   intGT n'16 n'17 ; λb'2 →
                   if b'2 then
                      unit n'29
                   else
                      intAdd n'16 1 ; λn'18 →
                      store (CInt n'18) ; λp21 →
                      intAdd n'29 n'16 ; λn'28 →
                      sum n'28 p21 p30
```

End program

The program is now much shorter, but there is still room for improvement. There are still store and fetch operations in *sum* that seem unnecessary, for the same reasons that we discussed above. Fortunately, the arity raising will once again apply, and will transform the second and third arguments of *sum* to

unboxed integers. In this case it is essential that the arity raising can handle *invariant arguments* for the transformation to succeed ("p_{30}").

Begin program

```
main = store (CInt 1) ; λ p2 →
       store (CInt 1000) ; λ p3 →
       sum 0 1 1000 ; λ n'13 →
       intPrint n'13
sum n'29 n'30 n'31 = unit (CInt n'30) ; λ (CInt n'16) →
                     unit (CInt n'31) ; λ (CInt n'17) →
                     intGT n'16 n'17 ; λ b'2 →
                     if b'2 then
                        unit n'29
                     else
                        intAdd n'16 1 ; λ n'18 →
                        store (CInt n'18) ; λ p21 →
                        intAdd n'29 n'16 ; λ n'28 →
                        sum n'28 n'18 n'31
```

End program

After yet another round of copy propagation and dead code elimination we arrive at the following:

Begin program

```
main = sum 0 1 1000 ; λ n'13 →
       intPrint n'13
sum n'29 n'30 n'31 = intGT n'30 n'31 ; λ b'2 →
                     if b'2 then
                        unit n'29
                     else
                        intAdd n'30 1 ; λ n'18 →
                        intAdd n'29 n'30 ; λ n'28 →
                        sum n'28 n'18 n'31
```

End program

Compared to the original GRIN program, this is a much more efficient program. It uses no heap, everything is strict and unboxed, and due to the fact that *sum* was tail recursive the program will also execute in constant stack space. The RISC part of our back-end will compile this into code that is more or less the same that a good imperative compiler would produce for a program that used a single "for loop" to sum the numbers.

It is hard to characterise exactly what the "magic" is that makes this transformation example succeed so well. In fact, it is not even possible to select a

single transformation that is the most important. Instead there are many transformations that work together. The late inlining starts it all by inlining *upto* into *sum*. The case hoisting combines the inlined code with that of the original *sum* function (short-circuit), and after that the arity raising performs the elimination of the intermediate data structure (the actual *deforestation*). Note also that the initial inlining of the call to *upto* would not have been possible without the control flow analysis and the *eval* inlining, since that call is not visible in the original functional source code.

More work is needed to understand what the relationship is between this and Wadler's deforestation [Wad88]. The success of the deforestation in the above example depends heavily on the fact that the list produced by *upto* is only used by a single consumer, and hence the Fupto closure inside *sum* needs not be updated. Also, more work is needed to see how often effects like this can appear in larger programs.

Bibliography

[Aga98] J. Agat. Types for Register Allocation. *Lecture Notes in Computer Science*, 1467, 1998.

[AGT89] A. V. Aho, M. Ganapathi, and S. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AH82] Marc Auslander and M. Hopkins. An Overview of the PL.8 Compiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 22–30, June 1982.

[AHU85] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1985.

[AJ76] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[AJ89a] 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.

[AJ89b] 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.

[AJ89c] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[AM87] A. W. Appel and D. 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.

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

[App98] A. W. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4), April 1998.

[AS92] A. W. Appel and Z. Shao. Callee-save Registers in Continuation-passing Style. *Lisp and Symbolic Computation*, 5:189–219, 1992.

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

[Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, 1984.

[Aug85] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Aug86] L. Augustsson. The revised G-machine. Technical Report Memo 45, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1986.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.

[Aug93] Lennart Augustsson. Implementing Haskell Overloading. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 65–73. ACM Press, June 1993.

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 1993 Conference on Functional Programming and Computer Architecture.* ACM Press, June 1993.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equalities of Variables in Programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[BBH97] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices.* ACM Press, June 1997.

[BCKT89] P. Briggs, K. D. Cooper, K. 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.

[BCT92] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 311–321, San Francisco, CA, June 1992.

[BCT94] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[BGM+89] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 258–263, Portland, OR, June 1989.

[Bir84] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21, 1984.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell, 2nd edition.* Prentice Hall Press, 1998.

[BJ96] U. Boquist and T. Johnsson. The GRIN Project: A Highly Optimising Back End For Lazy Functional Languages. In *Selected papers from the 8th International Workshop on Implementation of Functional Languages*, Bad Godesberg, Germany, September 1996. Springer-Verlag, LNCS 1268.

[Blo94] A. Bloss. Path Analysis and the Optimization of Nonstrict Functional Languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, May 1994.

[Boe93] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. *SIGPLAN Notices*, 28(6):197–206, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

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

[Boq95b] U. Boquist. *Interprocedural Register Allocation for Lazy Functional Languages.* Licentiate thesis, Department of Computing Science, Chalmers University of Technology, 1995.

[Boq98] U. Boquist. Garbage collection support in an optimising Haskell compiler. In *Proceedings of The Wintermeeting.* Department of Computer Science, Chalmers University of Technology, January 1998.

[Bri92] P. Briggs. *Register allocation via graph coloring.* PhD thesis, Department of Computer Science, Rice University, 1992. Rice COMP TR92-183.

[BRJ88] G. Burn, J. Robson, and S. L. Peyton Jones. The Spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[BWD95] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register Allocation Using Lazy Saves, Eager Restores, and Greedy Shuffling. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, California, June 1995.

[CAC+81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, Los Angeles, 1977.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 53–65, White Plains, NY, June 1990.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The Categorical Abstract Machine. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 50–64, Nancy, France, 1985.

[CD96] W. N. Chin and J. Darlington. A Higher-Order Removal Method. *Lisp and Symbolic Computation*, 9(4):287–322, December 1996.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[CH84] 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.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, volume 17, June 1982.

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13:677–679, 1970.

[Chi98] O. Chitil. Common Subexpressions Are Uncommon in Lazy Functional Languages. *Lecture Notes in Computer Science*, 1467, 1998.

[CHKW86] F. Chow, M. Himmelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *Spring 86 COMPCON Digest of Papers*, pages 132–137. IEEE, March 1986.

[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.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, 1941.

[CK91] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 192–203, Toronto, Ontario, Canada, June 1991.

[CK94] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 Conference on Measurement and Modeling of Computer Systems*. ACM SIGMETRICS, 1994.

[DGS92] E. Duesterwald, R. Gupta, and M. L. Soffa. Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables. In *Compiler Construction, 4th International Conference, CC'92*, Paderborn, 1992. Springer-Verlag, LNCS 641.

[DW91] J. W. Davidson and D.B. Whalley. Methods for Saving and Restoring Register Values across Function Calls. *Software – Practice & Experience*, 21(2):149–165, February 1991.

[Fax95a] K. F. Faxén. Optimizing Lazy Functional Programs Using Flow-Inference. In A. Mycroft, editor, *Static Analysis Symposium (SAS)*, volume 883 of *LNCS*. Springer Verlag, September 1995.

[Fax95b] K. F. Faxén. Optimizing Lazy Functional Programs Using Flow-Inference. In Nielson and Solberg, editors, *Workshop on Types for Program Analysis*, Aarhus, Denmark, May 1995. Aarhus University.

[Fax97] K. F. Faxén. *Analyzing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Dept. of Teleinformatics, KTH, Stockholm, June 1997.

[FH92] C. W. Fraser and D. R. Hanson. Simple Register Spilling in a Retargetable Compiler. *Software – Practice & Experience*, 22(1):85–99, January 1992.

[FHP92] C. Fraser, R. Henry, and T. Proebsting. BURG–Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4), April 1992.

[FMM87] M. Flynn, C. Mitchell, and H. Mulder. And now a case for more complex instruction sets. *IEEE Computer*, 20(9):71–83, September 1987.

[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.

[GA96a] L. George and A. W. Appel. Iterated register coalescing. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[GA96b] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[GBJ82] M. L. Griss, E. Benson, and G. Q. Maguire Jr. PSL: A Portable Lisp System. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 88–97, 1982.

[Geo96] L. George. MLRISC: Customizable and Reusable Code Generators. Technical report, Bell Labs, December 1996.

[GGR94] L. George, F. Guillame, and J. 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.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[GM86] P. B. Gibbons and S. 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.

[Gro90] J. van Groningen. Implementing the ABC-machine on MC680x0 based architectures. Master's thesis, Department of Informatics, University of Nijmegen, November 1990.

[GSS89] R. Gupta, M. L. Soffa, and T. Steele. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 264–274, Portland, OR, June 1989.

[HGAM92] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. In *Compiler Construction, 4th International Conference, CC'92*, Paderborn, 1992. Springer-Verlag, LNCS 641.

[HGAM93] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Journal of Programming Languages*, 1:155–185, 1993.

[HGW91a] P. Hartel, H. Glaser, and J. Wild. Compilation of functional languages using flow graph analysis. Technical Report CSTR 91-03, University of Southampton, 1991.

[HGW91b] 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.

[HL93] P. Hartel and K. Langendoen. Benchmarking implementations of lazy functional languages. In *Proceedings 1993 Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 1993.

[HM76] P. Henderson and J. H. Morris. A Lazy Evaluator. In *3'rd conference on The Principles of Programming Languages*, pages 95–103, Atlanta, Georgia, January 1976.

[HM82] John L. Hennessy and Noah Mendelsohn. Compilation of the Pascal `case` statement. *Software — Practice and Experience*, 12(9), September 1982.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Palo Alto, 1990.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Hug82] R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, 1982.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Joh81] T. Johnsson. Code Generation for Lazy Evaluation. Technical Report Memo 22, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, 1981.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, 1984.

[Joh85] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Nancy, France, 1985. Springer Verlag.

[Joh86] T. Johnsson. Code Generation from G-machine code. In *Proceedings of the workshop on Graph Reduction*, Lecture Notes in Computer Science 279, Santa Fe, September 1986. Springer Verlag.

[Joh87a] T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Proceedings 1987 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Portland, Oregon, U.S.A., 1987. Springer Verlag.

[Joh87b] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[Joh91] T. 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. Springer Verlag, August 1991.

[Kem79] A. B. Kempe. On the Geographical Problem of the Four Colours. *American Journal of Mathematics*, 2, 1879.

[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.

[KHO96] M. Koch and T. Höjfeld Olesen. Compiling a Higher Order Call-by-Value Functional Programming Language to a RISC using a Stack of Regions. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. 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.

[Lan64] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[Lau93] J. Launchbury. Semantics for Graph Reduction. In *Proceedings 1993 Symposium Principles of Programming Languages*, Charleston, N. Carolina, 1993.

[Ler92a] Xavier Leroy. Unboxed Objects and Polymorphic Typing. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, Albuquerque, NM, January 1992. ACM Press.

[Ler92b] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albequerque, New Mexico, January 1992.

[Ler97a] Xavier Leroy. Re: C-- GC. Discussion on the C-- mailing list, 1997.

[Ler97b] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[LH86] J. R. Larus and P. 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.

[LH92] K. Langendoen and P. Hartel. FCG: a code generator for lazy functional languages. In *Compiler Construction, 4th International Conference, CC'92*, Paderborn, 1992. Springer-Verlag, LNCS 641.

[LR91] 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.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[MR85] P. Mishra and U. S. Reddy. Declaration-free Type Checking. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 7–21, New Orleans, Luisiana, 1985. ACM.

[Mye81] E. W. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[Nic90] B. R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 40–52, White Plains, NY, June 1990.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and Continuation-passing Style. *Lisp and Symbolic Computation*, 7(1), January 1994.

[Par93] W. Partain. The **nofib** benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Proc. 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 195–202. Springer–Verlag, 1993.

[Pau94] Richard P. Paul. *SPARC Architecture, Assembly Language Programming, & C.* Prentice Hall, 1994.

[PF92] T. Proebsting and C. N. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 300–310, San Francisco, CA, June 1992.

[PH93] John Peterson and Paul Hudak. The Yale Haskell Common Lisp Interface. Research Report YALEU/DCS/RR-972, Yale University Department of Computer Science, New Haven, Connecticut, June 1993.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

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

[PJHH+93] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, March 1993.

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

[PJLST98] S. L. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: A common intermediate language for ML and Haskell. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

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

[PJW93] S. L. Peyton Jones and P. Wadler. Imperative Functional programming. In *Proceedings 1993 Symposium Principles of Programming Languages*, Charleston, N.Carolina, 1993.

[Plo81] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, September 1981.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM National Conference*, volume 2, August 1972.

[Rey98a] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Journal of Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[Rey98b] John C. Reynolds. Definitional Interpreters Revisited. *Journal of Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[SA95] Z. Shao and A. W. Appel. A Type-Based Compiler for Standard ML. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'95)*, pages 116–129, La Jolla, June 1995. ACM.

[SA94] Z. Shao and A. W. Appel. Space-Efficient Closure Representations. In *Conference on Lisp and Functional programming*, June 94.

[Sal81] Arthur Sale. The Implementation of Case Statements in Pascal. *Software — Practice and Experience*, 11(9):929–942, September 1981.

[San95] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

[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), January 1989.

[SH97] M. Shapiro and S. 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.

[Shi88] O. Shivers. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, Atlanta, GA, June 1988.

[SNvGP91] S. Smetsers, E. Nöcker, J. 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.

[SO90] 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.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*, 1992.

[Spu94] D. A. Spuler. Compiler Code Generation for Multiway Branch Statements as a Static Search Problem. Technical Report JCU-CS-94/3, Department of Computer Science, James Cook University, 1994.

[Sta88] W. Stallings. Reduced Instruction Set Computer Architecture. *Proceedings of the IEEE*, 76:38–55, 1988.

[Ste78] G. L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, 1978.

[Ste84] G. L. Steele. *Common LISP - the language*. Digital Press, 1984.

[Ste91] Peter A. Steenkiste. Advanced Register Allocation. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1991.

[SU70] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17:715–728, October 1970.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[TMC+96] D. Tarditi, G. Morriset, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96)*, pages 181–192, Philadelphia, May 1996. ACM.

[TO99] A. Tolmach and D. Oliva. From LM to Ada: Strongly-typed Language Interoperability via source Translation. *Journal of Functional Programming*, 1999. To appear.

[Tol97] A. Tolmach. Combining Closure Conversion with Closure Analysis using Algebraic Types. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[Tur75] D. A. Turner. An implementation of SASL. Technical report 4, University of St. Andrews, 1975.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

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

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

[Wad90] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–77, Nice, France, 1990.

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

[Wal88] D. W. Wall. Register windows vs. register allocation. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 67–78, Atlanta, GA, June 1988.

[WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Mangement*, Lecture Notes in Computer Science, pages 1 – 42. Springer-Verlag, September 1992.

[WJW+75] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Elsevier Computer Science Library, New York, 1975.

[WW90] B. Weisgerber and R. Wilhelm. Two Tree Pattern Matchers for Code Selection (Including Targeting). In *Proceedings of the Workshop on Compiler Compilers and High Speed Compilation, Lecture Notes in Computer Science*, volume 371, 1990.