UNIVERSITY
of
GLASGOW

Department of
Computing Science

Compilation by Transformation in Non-Strict Functional Languages

André Luís de Medeiros Santos

Submitted for a Doctor of Philosophy Degree in Computing Science at the University of Glasgow

July 1995

Abstract

In this thesis we present and analyse a set of automatic source-to-source program transformations that are suitable for incorporation in optimising compilers for lazy functional languages. These transformations improve the quality of code in many different respects, such as execution time and memory usage.

The transformations presented are divided in two sets: global transformations, which are performed once (or sometimes twice) during the compilation process; and a set of local transformations, which are performed before and after each of the global transformations, so that they can simplify the code before applying the global transformations and also take advantage of them afterwards.

Many of the local transformations are simple, well known, and do not have major effects on their own. They become important as they interact with each other and with global transformations, sometimes in non-obvious ways. We present how and why they improve the code, and perform extensive experiments with real application programs.

We describe four global transformations, two of which have not been used in any lazy functional compiler we know of: the static argument transformation and `let` floating transformations. The other two are well known transformations for lazy functional languages, but for which no major studies of their effects have been performed: full laziness and lambda lifting. We also study and measure the effects of different inlining strategies.

We also present a *Cost Semantics* as a way of reasoning about the effects of program transformations in lazy functional languages.

To
my wife Ana.

Acknowledgements

First I would like to thank my supervisor, Professor Simon Peyton Jones, for his guidance and support during this research and for being such a friendly and informal person. I will certainly miss our weekly meetings.

I would like to thank the members of the examination committee, Lennart Augustsson, Satnam Singh and David Watt, for providing many interesting corrections and suggestions to improve this thesis.

The GRASP/AQUA team, by producing and supporting the Glasgow Haskell Compiler, was an essential part of this work. A very special thanks to Will Partain for his friendship, his patience in answering an endless stream of questions, and for providing helpful comments on a draft of this thesis.

I would like to thank my roommates Andy Gill, David King and Simon Marlow, who provided a most friendly and relaxed atmosphere in G162, and were always available to discuss and provide feedback on my work.

The Computing Science Department at the University of Glasgow provided a nice and informal working environment. The Functional Programming Group, with its annual workshops and weekly seminars, provided important feedback on my work. I also thank John Launchbury and Andy Gordon for helping me with the Cost Semantics.

A special thanks to my friend Hermano Moura, for taking me as a guest in his home when I first arrived in Glasgow, and for helping me with all sorts of things. I also thank the friends I made in Glasgow for their support and friendship, specially the Felix Family, Joaquim Martins Filho and the Barnett family.

I acknowledge the financial support I received from CAPES (Brazilian Federal Agency for Postgraduate Education), which allowed me to carry out the research described in this thesis.

I would also like to thank Professor Silvio Meira, who first introduced me to functional programming in Recife in 1986, and supervised me both as an undergraduate and MSc. student.

I thank my family, specially my parents, for their love and support.

Finally, my gratitude and love to my wife Ana for all her support, patience, and love.

André Santos

Contents

Chapter 1

Introduction

Due to their semantic properties, functional languages are very suitable for program transformations, more so than their imperative counterparts. The high level of abstraction, absence of side-effects and their clear and simple semantics are just a few of the characteristics that make it relatively easy to establish properties of functional programs [Hug89, Tur81].

Program transformation can be broadly classified into two groups:

- *Non-automatic* program transformations, which are performed manually or assisted by a computer, but need human intervention to select which transformations to use or to provide new transformations when needed. This is often used as a program development technique.
- *Automatic* program transformations, that can be entirely automated and incorporated into a compiler (although sometimes this is not practical due to performance issues).

In this thesis we describe *automatic* program transformations, suitable to be incorporated into an optimising compiler.

Traditional compilers often have the original language translated into different intermediate representations before generating object code. Although most optimisations performed in compilers can be regarded as program transformations, they are often implemented in these intermediate representations, which are often quite different from the original source language. The approach of compilation by program transformation [Kel89] uses a single intermediate representation, often based on the lambda calculus [Chu41, Bar84], during most of the compilation process. This approach has two important advantages:

- The source-to-source transformations are easier to be proven correct, and implemented correctly.
- It allows many optimisations often performed in an obscure way (sometimes during code generation) to be implemented as high level program transformations.

In this thesis we present and analyse the effects of a large set of optimisations that are expressed as program transformations in a functional language.

1.1 Contributions of the thesis

This thesis presents a detailed study of a large set of automatic program transformations. The study has several distinctive features:

- A large set of transformations is discussed in a single framework. Although many are simple, not all of them are obvious, and some of them are new transformations that were suggested by inspecting the intermediate code of our compiler. What we have found is that many of these transformations, although not presenting large benefits on their own, when combined can actually achieve major improvements in program performance. Although some of them are present in virtually every compiler in some form, they are seldom systematically described and analysed, and therefore their importance and effectiveness in real programs is not well known.
- The transformations are embedded into a real production-quality compiler, and therefore there are no hidden costs being paid due to unoptimised aspects of code generation.
- The measurements are performed using a large set of applications, from many sources. Many of them are real applications, with hundreds (and sometimes thousands) of lines, not small toy benchmark programs.
- We present and measure the effect of two new transformations: `let` floating and the static argument transformations, which were both suggested by code inspection. Both are shown to be important transformations, with `let` floating improving programs' performance by up to 38%, and the static argument transformation up to 10%.

- We discuss, evaluate and suggest improvements to two known transformations: full laziness and lambda lifting. We show that the risks of creating space leaks due to full laziness are much smaller in practice than what is suggested in the literature, and present ways of reducing it. We achieve an average performance improvement of 8% with full laziness, with a peak improvement of 52%, without any space leaks being created. Lambda lifting is shown to have a heavy penalty cost if always done (as in most implementations of functional languages), worsening the performance by up to 48%, and by 9% on average. Nevertheless we show that a more *selective* approach to lambda lifting can produce modest performance improvements.

- We present and measure the effect of different inlining strategies in the Glasgow Haskell Compiler, showing that a point where the improvements from inlining start to be too small to be worthwhile is quickly reached in our experimental framework. Inlining is shown to be very important, improving programs on average by about 40%.

- We present a *cost semantics* as a way to reason about the cost of expressions before and after a transformation. This allows a more rigorous definition of code improvement, which can be used to reason about the effects of a program transformation in a more formal framework.

Parts of this work have been previously presented in [SP92, PS94].

1.2 Structure of the thesis

We start by describing the framework we will use to present and measure the effectiveness of our transformations, introducing the Core language, how we measured the effect of the transformations and what benchmark programs we used (**Chapter 2**).

We then present the set of small local transformations we use (**Chapter 3**). We describe each of the transformations, presenting why they improve the code and what (if any) risks are involved in performing each of the transformations. We also present measurements on the effect of some of the transformations, whenever there are different options for performing it, and compare the results. In **Chapter 4** we describe some details of how the local transformations were implemented and measure how often they are actually used and their effect.

In **Chapter 5** we introduce and evaluate the `let` floating inwards transformation, and discuss the full laziness transformation and its effects. We present ways to reduce the risk of creating space leaks when performing the full laziness transformation. We also discuss the constraints on ordering these transformations, and how we ordered them.

Chapter 6 presents the different inlining strategies and their effect. We measure the effect of increasing the amount of functions inlined on many aspects: code size, compilation time, heap allocated and instructions executed.

In **Chapter 7** we introduce and evaluate the static argument transformation and discuss the lambda lifting transformation and its effects. We first show that the static argument transformation can have some positive effect in a few programs. We then proceed to discuss lambda lifting, showing the problems with always performing it, and then try to restrict it to cases where it can be beneficial. Finally, we try to combine the two transformations.

In **Chapter 8** we discuss the different approaches to program transformations in the literature and compare the transformations we use with the ones used in other functional and imperative languages' compilers.

In **Chapter 9** we introduce a *cost semantics*, which can be used to reason about the cost of expressions before and after program transformations.

Finally in **Chapter 10** we present our conclusions and future work.

Chapter 2

Framework

In this chapter we describe the experimental framework in which our measurements are made: we present the language we use to describe the transformations and explain how we measure the effect of the transformations in our benchmark programs.

We initially present an overview of the Glasgow Haskell Compiler (Section 2.1), which is the system in which the transformations were implemented and experimented with. We then present some characteristics of the intermediate language of the compiler: the Core language (Section 2.2). Finally we discuss how we can measure the claimed improvements performed by the transformations (Sections 2.3 and 2.4) and introduce the benchmark programs we will use to substantiate our claims on program improvement (Section 2.5).

2.1 Overview of the compiler

The Glasgow Haskell Compiler has a modular design, making it relatively easy to modify or introduce extra passes into it. Furthermore it is a production-quality compiler, capable of dealing with substantial "real" Haskell programs, which ensures meaningful results. Therefore it was the ideal tool to implement and measure the effectiveness of the program transformations.

The compiler is structured as a series of passes, as presented in Figure 2.1. The main passes are:

- the *parser*, written in Lex and Yacc;

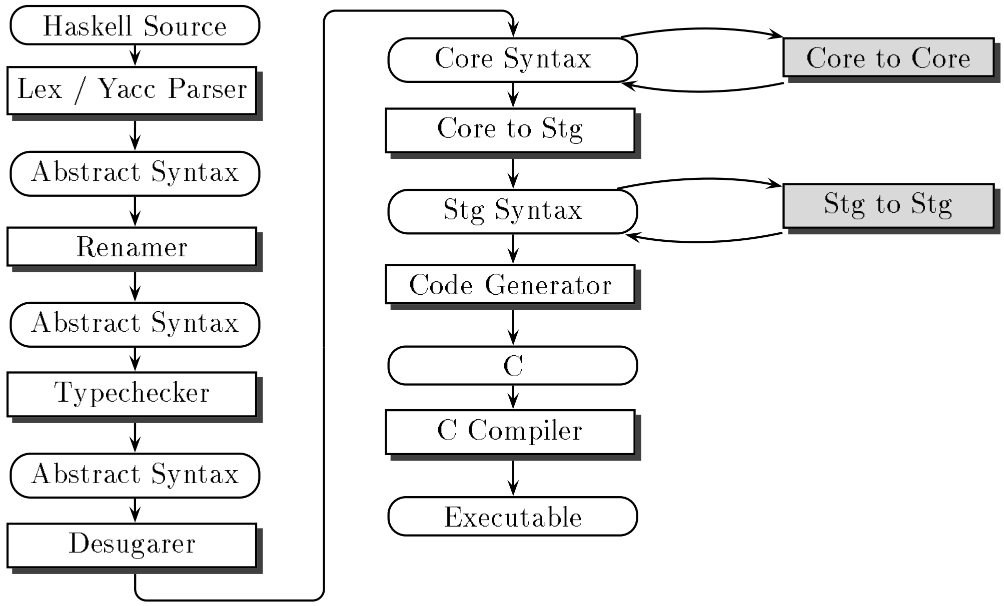

Figure 2.1 The Glasgow Haskell Compiler

- the *renamer*, which resolves scoping and naming issues, especially those concerned with module imports and exports;
- the *type inference pass*, which annotates the program with type information and transforms out overloading [WB89];
- the *desugarer*, which transforms out the high level constructs of Haskell (e.g. pattern matching and list comprehensions) to a much simpler functional language called the *Core language*, which we describe in Section 2.2;
- a series of transformation passes over the Core language, most of which we describe in this thesis, that aim at improving the efficiency of the code;
- a translator from the Core language to the *Shared Term Graph (STG) language*[1] [Pey92], which is a purely functional language even simpler than the Core language;
- transformation passes in the STG language, some of which are described in this thesis;

[1] "STG language" was originally short for Spineless Tagless G-machine language, but in fact the language is entirely independent of the abstract machine model used to implement it.

- the *code generator*, which converts the STG language to Abstract C, an internal data type that can easily be printed in C syntax;
- a pass that flattens and prints out the C code, which is then compiled by a C compiler. Optionally the compiler can also generate assembly code directly for some architectures.

As one can see most of the compilation process is expressed as correctness-preserving transformations of a purely functional program, as the intermediate languages used by the compiler up to code generation are pure functional languages themselves.

2.2 The Core language

The Core language is intended to be the simplest language into which Haskell can be translated (or desugared) without loss of efficiency. List comprehensions, pattern matching, guarded equations and conditionals are all translated out, but simple `case` expressions, `let` (`rec`) expressions and constructors remain. The abstract syntax of the Core language is given on Figure 2.2. The Core language is essentially the second-order lambda calculus augmented with `case`, `let`, constants, constructors and primitive operators.

The concrete syntax we use is conventional, but we allow ourselves the use of the following conventions and liberties:

- parentheses are used to disambiguate;
- application associates to the left and binds more tightly than any other operator;
- the body of a lambda abstraction extends as far to the right as possible;
- the usual infix arithmetic operators are permitted;
- the usual syntax for lists is allowed, with infix constructor "`:`" and empty list `[]`;
- where the layout makes the meaning clear we omit semicolons between bindings and `case` alternatives.
- sometimes we use `\` to denote λ and `/\` to denote Λ.

Program	*Prog*	$\rightarrow$	$Binding_1$; ... ; $Binding_n$	$n \geq 1$
Bindings	*Binding*	$\rightarrow$	*Bind*	
		$\mid$	`rec` $Bind_1 \ldots Bind_n$	
	Bind	$\rightarrow$	*var* `=` *Expr*	
Expression	*Expr*	$\rightarrow$	*Expr Atom*	Application
		$\mid$	*Expr ty*	Type application
		$\mid$	$\lambda\ var_1 \ldots var_n$ `->` *Expr*	Lambda abstraction
		$\mid$	Λ *ty* `->` *Expr*	Type abstraction
		$\mid$	`case` *Expr* `of` *Alts*	Case expression
		$\mid$	`let` *Binding* `in` *Expr*	Local definition
		$\mid$	$con\ Atom_1 \ldots Atom_n$	Constructor $n \geq 0$
		$\mid$	$prim\ Atom_1 \ldots Atom_n$	Primitive $n \geq 0$
		$\mid$	*Atom*	
Atoms	*Atom*	$\rightarrow$	*var*	Variable
		$\mid$	*Literal*	Unboxed Object
Literal values	*Literal*	$\rightarrow$	*integer* $\mid$ *float* $\mid$ ...	
Alternatives	*Alts*	$\rightarrow$	$Calt_1$; ... ; $Calt_n$; [*Default*]	$n \geq 0$
		$\mid$	$Lalt_1$; ... ; $Lalt_n$; [*Default*]	$n \geq 0$
Constr. alt	*Calt*	$\rightarrow$	$con\ var_1 \ldots var_n$ `->` *Expr*	$n \geq 0$
Literal alt	*Lalt*	$\rightarrow$	*Literal* `->` *Expr*	
Default alt	*Default*	$\rightarrow$	*var* `->` *Expr*	

Figure 2.2 Syntax of the Core language

- function bindings are expressed by binding a variable to a lambda abstraction, although sometimes we write the arguments of function bindings to the left of the `=` sign.

The main points to be noted about the Core language are:

- *The bindings in* `let` *expressions are all simple.* That is, the left hand side of the binding is always a variable. Similarly, the patterns in `case` expressions are all simple; nested pattern matching has been compiled to nested `case` expressions.

- *Explicit type abstraction and application.* Since type information is preserved in the Core language, program transformations done in the Core language must preserve the type correctness of the program. This is made easier through the use of explicit type abstractions and applications, as we describe in Section 2.2.1. Type information is desirable and sometimes essential to later stages of compilation: for example, a higher-order strictness analyser may need accurate type information in order to construct correct fixed points.

- *Atomic Arguments.* The arguments of an application or constructor are restricted to be atoms (variables or literals). This restriction allows us to have smaller and simpler sets of transformations than we would if the arguments were arbitrary expressions. An example of the benefits of this design decision is presented in Section 3.1.

- *Applications of constructors and primitive operators are saturated.* Partial applications of constructors and primitive operators can still be obtained through the use of lambda abstractions, e.g.

  ```
  (:) a    ==>    \b -> (:) a b
  ```

- *Unboxed values.* The Core language allows us to express boxed as well as unboxed values, therefore allowing many transformations usually left for the code generator and handled in a completely ad-hoc way to be expressed as Core language program transformations [PL91a]. The definition of the `+` operation for example is

  ```
  (+) = \ x -> \ y -> case x of
                        MkInt x# -> case y of
                                      MkInt y# -> case (x# +# y#) of
                                                    r# -> MkInt r#
  ```

where we have the definition of a function that receives two (boxed) arguments, unboxes the first one (`MkInt` is the constructor for a boxed integer), then the second one, applies the *unboxed* operator `+#` to the two unboxed values and finally returns a *boxed* result (using again the constructor `MkInt`). We will often append the character `#` to *primitive* operators or *unboxed* variables.

- *Core language programs have a direct operational interpretation:*
  - all heap allocation is represented by `let`s;
  - evaluation is always denoted by `case`s.

  Notice that `case`s in the Core language are always strict. This means that they are not identical to `case`s in Haskell. Specifically, expressions such as

  ```
  case e of v -> b
  ```

  in Haskell are equivalent to

  ```
  let v = e in b
  ```

  but not in Core. In Core the former denotes that `e` is evaluated and its value then bound to `v`, while the latter means that a closure is built for `e` (unevaluated) and bound to `v`.

2.2.1 Preserving type information

To illustrate the importance of the use of the second-order lambda calculus to preserve type information between transformations we will consider the following example from [PHH+93]: consider the function `compose`, whose type is

$$\texttt{compose} :: \forall\alpha\beta\gamma.(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

The function might be defined like this in an untyped Core language:

```
compose = \f -> \g -> \x ->
          let y = g x in f y
```

Now, suppose that we want to inline a particular call to the `compose` function, e.g. `(compose show double v)` where `v` is an `Int`, `double` doubles it, and `show` converts the result to a `String`. The result of inlining the call to `compose` is an instance of the body of `compose`, thus:

```
let y = double v in show y
```

Now, we want to be able to identify the type of every variable and sub-expression, so we must be able to *calculate* the type of `y`. In this case, it has type `Int`, but in another application of `compose` it may have a different type. All this is because its type in the body of `compose` itself is just a type variable, β. It is clear that, in a polymorphic world, it is insufficient merely to tag every variable of the original program with its type, because this information does not survive across program transformations. Indeed no other compiler known to us for a polymorphically-typed language preserves type information across arbitrary transformations.

Clearly, the program must be decorated with type information in some way, and every program transformation must be sure to preserve it.

Using the second-order lambda calculus, the idea is that every polymorphic function, such as `compose` receives a type argument for each universally-quantified polymorphic variable in its type (α, β, and γ in the case of `compose`). Whenever a polymorphic function is called, it is passed extra type arguments to indicate the types to which its polymorphic type variables are to be instantiated. The definition of `compose` now becomes[2]:

```
compose = /\a,b,c    ->
          \f::(b->c) ->
          \g::(a->b) ->
          \x::a      ->
          let y::b = g x in f y
```

The function takes three type arguments (`a`, `b` and `c`), as well as its value arguments `f`, `g` and `x`. The types of the latter can now be given explicitly, as can the type of the local variable `y`. A call of `compose` is now given three extra type arguments, which instantiate `a`, `b` and `c` just as the "normal" arguments instantiate `f`, `g` and `x`. For example, the call of `compose` we looked at earlier is now written like this:

```
compose Int Int String show double v
```

It is now simple to inline this call, by instantiating the body of `compose` with the supplied arguments, to give the expression

```
let y::Int = double v in show y
```

[2]we take the liberty of presenting some types explicitly.

The `let`-bound variable `y` is now *automatically* attributed the correct type.

In short, the second-order lambda calculus provides a well-founded notation in which to express and transform polymorphically-typed programs. The type inference pass produces a translated program in which the "extra" type abstractions and applications are made explicit.

The propagation and use of type information is beyond the scope of this thesis, so we do not discuss it further. In all subsequent example programs type abstractions and applications are omitted when they are not relevant.

2.3 What is an optimisation?

The aim of any optimisation technique is to reduce either the time or the space needs of an executing program. In the functional language context the time and space costs of a program can be measured in the following ways:

- *Execution time.* This is certainly one of the major goals of any optimisation, to make the program run in less time. Execution time unfortunately is not an easy number to measure in modern multi-tasking multi-user computers. This is due to a number of factors:

  - The computer is running various other processes: even when there is only one user, the machine is still running operating system tasks, like dealing with network traffic. Even in single user mode, without any network connection, one has to repeatedly perform the measurements and average them to have a reliable data execution time. These factors affect wall clock time (elapsed time) as well as the so called *user* time when performing measurements.
  - Due to the large number of experiments we perform in this thesis it was not practical to have dedicated a powerful non-networked machine to perform them. Also the necessity to run the experiments many times makes the task even more time consuming.
  - Even in ideal circumstances, just the behaviour of the computer cache is enough to generate very different results every time a program is run [HBH93].

  When one is looking for considerable changes in performance it is often reasonable to accept a small error margin in the measurements. In our case we will sometimes be looking for small improvements caused by a small transformation, therefore we cannot easily get within an acceptable error margin. Due to the reasons above we have decided to measure the *instructions executed* by each program instead of the time. This is our next item.

- *Total instructions executed.* To measure the total instructions executed by a program instead of its execution time has the following advantages:

  - it is a repeatable number.
  - it is not affected by other programs or cache behaviour (or even paging behaviour).
  - it does not need a standalone machine to be measured.
  - it is a good predictor of run-time improvements. Although not all instructions have the same execution time, and each program uses a different mix of them, on a given program in which different transformations are performed we have observed that the run-time improvement is very close to the improvement on the total number of instructions executed.

  This same approach is used for example in [App92], for similar reasons.

- *Memory traffic.* One way of measuring the amount of memory traffic is by counting the number of instructions that access memory. In a RISC machine, this is usually made explicit by the use of load and store instructions, so it basically amounts to counting those instructions.

  When performing our measurements we often measured the improvement in memory traffic, but since it was often very close to the improvements we get on the total of instructions executed we decided only to present the latter.

- *Amount of heap allocation.* The amount of heap allocated (measured in bytes) indicates the amount of memory used by the closures built on the heap. There are many costs involved in allocating each closure:

  - a heap check, to verify if there is space available for the closure in the heap (otherwise the garbage collector must be called).
  - initialising the fields of the closure in the heap.
  - possibly evaluating the closure, if it is ever demanded.
  - if the closure is updatable, the cost of the update.

The amount of heap allocated by a program is also directly related to the number of garbage collections performed. The more heap that is allocated, the more garbage collections will be performed. When performing our measurements we have observed that the amount of heap allocated is not directly correlated to the run-time behaviour of a program, since we have seen sometimes major variations in heap allocation which had minor effects on execution time. Indeed, the allocation rate of the programs in our benchmarks (i.e. number of bytes allocated per second) varied a lot, from as little as 1.2Mb/s up to 11.2 Mb/s.

- *Number of updates.* An updatable closure is expensive due to its cost in memory accesses: it is written to memory (created) and (if entered) is read from memory again and later updated with its result (another write operation to memory). Usually a high proportion of the updated closures are never entered again [SP93], and therefore were unnecessary. In the Glasgow Haskell Compiler itself has been measured that about 77% of the updates performed are unnecessary [SP93]. Some optimisation techniques try to reduce the number of updates performed. This can be achieved by:

  – early evaluation of strict (demanded) closures: we use strictness analysis together with some transformations to achieve this result (Section 3.6).

  – exposing weak head normal form closures: this is done by `let` floating (Section 3.4).

  – finding which closures will be entered only once, using update analysis [LGH+92, Mar93].

- *Heap residency.* Heap residency is the amount of heap that is considered live (that is, not garbage) at a given time. Therefore the peak (maximum) heap residency in a program run defines (approximately) the minimum amount of heap which the program must have available to execute. Heap residency also affects the number of garbage collections by defining at each garbage collection the amount of live data and therefore the amount of free space. If the free space is too small there will soon occur another garbage collection and so on. By reducing the peak heap residency one reduces the actual minimum amount of heap in which the program runs.

- *Code size.* Some transformations may affect code size by duplicating code, e.g. inlining (Chapter 6).

- *Stack depth vs. heap allocation.* Depending on the way a function is defined one can use more heap or more stack. Let us take the following function definition that takes the sum of a list:

```
let sum l = case l of
            []     -> 0
            (x:xs) -> x + sum xs
in sum [1..100000]
```

It consumes very little heap because as the list is built, it is consumed by `sum` and can be garbage collected. But it uses a stack proportional to the length of the list, since we have to compute all the calls to `sum` before proceeding with the additions. Another possible definition uses an accumulating parameter:

```
let sum a l = case l of
                []     -> a
                (x:xs) -> sum (a+x) xs
in sum 0 [1..100000]
```

In this case an implementation based on graph reduction performs the evaluation in constant stack space (due to the fact that we are using tail recursion), although (usually) at the cost of increasing heap usage. Actually, by using strictness analysis one can perform the evaluation in constant heap and stack.

2.4 How we performed the measurements

All our measurements are performed on a SparcStation 10 with 80Mb of memory. The tool used to count the number of instructions executed was *SpixTools* from Sun Microsystems [Sun93]. All programs, unless stated otherwise, are run on a 50Mb heap, to minimise the effects of garbage collection on the instruction counting. We use the Glasgow Haskell Compiler version 0.23[3] for our measurements. We do not exclude the effects of garbage collection in our measurements, but by using a large heap when running the programs the effect of garbage collection is minimised, with many of the programs not performing any garbage collection[4]. One may be concerned that a minor change in the amount of allocation in a program may have a major effect in instructions executed due to triggering (or eliminating) one or more garbage collections. Although this is a real risk, we have not found any such cases.

[3]With some improvements which will be incorporated in versions 0.24 and above.

[4]The number of garbage collections typically performed by each program is presented in Table 2.3 in the end of this chapter, together with the percentage of the run-time of the program spent on garbage collection.

We also always perform the measurements with a Haskell prelude compiled with the same compiler options that we are measuring. This has the advantage of extending our testing (and measurements) to include the effects on the parts of the prelude used by the programs.

Sometimes we try to be even more precise and present measurements showing exactly where and how that time and or space is being saved. This is done using the profiling tools available in the Glasgow Haskell Compiler, which provide us with many fine-grain measures, like number of updates, number of heap allocations (heap checks) etc. An example of the information given by such a tool is given in Figure 2.3.

Heap residency is particularly hard to measure, since we are looking for the maximum amount of live data (i.e. data that cannot be garbage collected) at any one time. It is not practical to perform a garbage collection after every heap object is allocated, therefore we have to rely on performing garbage collections after every n bytes are allocated, and rely on having enough samples to make the data reliable. For our benchmark programs, due to the amount of heap they allocate, we have decided (based on measuring residency for different values for n) that measuring the residency at every 1Mbytes allocated was a good compromise. Of course one is always risking that if a residency peak occurs within that 1Mbytes allocated it could possibly go unnoticed.

When presenting the results our tables will often look like the following:

Transformation Name Residency			
program	option 1	option 2	option 3
`queens`	1.00	0.75	0.50
`hidden`	1.00	0.80	0.90
⋮	⋮	⋮	⋮
n other programs	1.00	1.00	1.00
Minimum	-	0.75	0.50
Maximum	-	1.00	1.05
Geometric Mean	-	0.96	0.92

First we specify what transformation we are measuring and what we are measuring (e.g. residency, total instructions executed, total heap allocated). Then we list the options we tried, and start listing the programs. One of the columns is the baseline (always 1.00), and the other columns are normalised with respect to that column, e.g. if we were measuring the execution time the first program took 200 seconds to run with option 1, it would have taken 150 seconds with option 2 and 100 seconds

```
queens_b +RTS -H50m -r

ALLOCATIONS: 920057 (1980927 words total: 920065 admin,
                                         1060824 goods, 38 slop)
                                total words:     2     3     4     5    6+
  34825 (  3.8%) function values              0.0 100.0   0.0   0.0   0.0
  70415 (  7.7%) thunks                       0.0 100.0   0.0   0.0   0.0
 814809 ( 88.6%) data values                 95.6   4.4   0.0   0.0   0.0
      0 (  0.0%) big tuples
      4 (  0.0%) black holes                  0.0 100.0   0.0   0.0   0.0
      2 (  0.0%) prim things                  0.0   0.0 100.0   0.0   0.0
      2 (  0.0%) partial applications         0.0   0.0   0.0  50.0  50.0

Total storage-manager allocations: 1021812 (2325171 words)
        [344244 words lost to speculative heap-checks]

STACK USAGE:
        A stack slots stubbed: 2043228
        A stack max. depth: 27 words
        B stack max. depth: 226 words

ENTERS: 6261977  of which 1546006 (24.7%) direct to the entry code
                  [the rest indirected via Node's info ptr]
  70416 (  1.1%) thunks
4645510 ( 74.2%) data values
1546041 ( 24.7%) function values
                  [of which 1546004 (100.0%) bypassed arg-satisfaction chk]
      0 (  0.0%) partial applications
     10 (  0.0%) indirections

RETURNS: 5029300
5029297 (100.0%) in registers [the rest in the heap]
 383790 (  7.6%) from entering a new constructor
                  [the rest from entering an existing constructor]
1894132 ( 37.7%) vectored [the rest unvectored]

UPDATE FRAMES: 70413 (3 omitted from thunks)
  70413 (100.0%) standard frames
      0 (  0.0%) constructor frames
                  [of which 0 (0.0%) were for black-holes]

UPDATES: 70413
  35613 ( 50.6%) data values
                  [35613 in place, 0 allocated new space, 0 with Node]
      5 (  0.0%) partial applications
                  [3 in place, 2 allocated new space]
  34795 ( 49.4%) updates to existing heap objects
      3 (  0.0%) in-place updates copied
```

Figure 2.3 Profiling Output

with option 3. The programs are usually sorted with respect to one of the columns. Programs that did not show any variation greater than 0.5% (as the numbers are rounded) are grouped in a separate row stating how many programs were omitted. Finally we summarise the best and worst results, and present the geometric mean for each column (because we are using normalised results [FW86]).

2.5 The benchmark programs

Many papers present performance measurements that use very small programs to measure the effect of optimisations. These programs are sometimes specially designed to demonstrate the effect of a particular optimisation. Although these are relevant measurements, they only present an upper bound on the effect of an optimisation, giving no insight on its effect on real programs. In order to present more realistic results, we measure the effect of transformations in many medium and large size programs, most of them being real application programs written by different people. These programs are grouped in the publically available **nofib** benchmark suite [Par92]. These programs are divided in 3 subsets[5], which we describe below, together with a short description of the programs in Table 2.1:

- the real subset: programs that perform a useful task, not written for demonstration or tutorial purposes;
- the imaginary subset: small toy benchmarks;
- the spectral subset: programs that don't meet the criteria of the real or the imaginary subset.

Pieter Hartel's benchmark suite programs [HL93, Har94] are part the spectral subset, and a short description of his programs is in Table 2.2.

In Table 2.3 we have a summary of the characteristics of the programs, compiling them with full optimisation in the Glasgow Haskell Compiler (`ghc -O`).

[5]Although we do not make distinctions between them when presenting our results.

Program	Subset	Description	Origin
`exp3_8`	Imaginary	3^8, using Peano arithmetic	–
`gen_regexps`	Imaginary	expands regular expressions	–
`primes`	Imaginary	Calculate prime numbers	–
`queens`	Imaginary	n-queens	–
`boyer2`	Spectral	Gabriel suite 'boyer' benchmark	–
`boyer`	Spectral	Gabriel suite 'boyer' benchmark	Denis Howe (Imperial)
`cichelli`	Spectral	Perfect hashing function	Iain Checkland (York)
`clausify`	Spectral	Propositions to clausal form	Colin Runciman (York)
`fft2`	Spectral	Fourier Transformation	Rex Page (Amoco)
`knights`	Spectral	Knight's tour	Jon Hill (QMW)
`mandel2`	Spectral	Mandelbrot sets	David Hanley
`mandel`	Spectral	Mandelbrot sets	Jon Hill (QMW)
`minimax`	Spectral	tic-tac-toe (0s and Xs)	Iain Checkland (York)
`multiplier`	Spectral	Binary-multiplier simulator	John O'Donnell (Glasgow)
`pretty`	Spectral	Pretty-printer	John Hughes (Chalmers)
`primetest`	Spectral	Primality testing	David Lester (Manchester)
`rewrite`	Spectral	Rewriting system	Mike Spivey (Oxford)
`sorting`	Spectral	Sorting algorithms	Will Partain (Glasgow)
`treejoin`	Spectral	Tree joining	Kevin Hammond (Glasgow)
`compress`	Real	Text compression	Paul Sanders (BT)
`fluid`	Real	Fluid-dynamics program	Xiaoming Zhang (Swansea)
`gg`	Real	Graphs from GRIP statistics	Iain Checkland (York)
`hidden`	Real	Hidden line removal	Mark Ramaer/Stef Joosten
`hpg`	Real	Haskell program generator	Nick North (NPL)
`infer`	Real	Hindley-Milner type inference	Phil Wadler (Glasgow)
`lift`	Real	Fully-lazy lambda lifter	David Lester (Manchester) & Simon Peyton Jones (Glasgow)
`maillist`	Real	Mailing-list generator	Paul Hudak (Yale)
`parser`	Real	Partial Haskell parser	Julian Seward (Manchester)
`prolog`	Real	"mini-Prolog" interpreter	Mark Jones (Oxford)
`reptile`	Real	Escher tiling program	Sandra Foubister (York)
`rsa`	Real	RSA encryption	John Launchbury (Glasgow)
`veritas`	Real	Theorem-prover	Gareth Howells (Kent)

Table 2.1 `nofib` benchmark programs

Program	Description
`comp_lab_zift`	Image processing application
`event`	Event driven simulation of a set-reset flipflop
`fft`	Two fast fourier transforms
`genfft`	generation of synthetic FFT programs
`ida`	Solution of a particular configuration of the n-puzzle
`listcompr`	Compilation of list comprehensions
`listcopy`	Compilation of list comprehensions (with extra list copying function for output)
`parstof`	Lexing and parsing based on Wadler's parsing method
`sched`	Calculation of an optimum schedule of parallel jobs with a branch and bound algorithm
`solid`	Point membership classification algorithm from a solid modeling library for computational geometry
`transform`	Transformation of a number of programs represented as synchronous process networks into master/slave style parallel programs
`typecheck`	Polymorphic type checking of a set of function definitions
`wang`	Wang's algorithm for solving system of linear equations
`wave4main`	Calculation of the water heights in a square area of 8×8 grid points of the North Sea over a long time period

Table 2.2 `nofib` benchmark: Hartel's benchmark programs

program	files	lines	object size	bytes allocated	exec. time	total instructions	total GC	% time in GC	alloc.rate (Mb/s)
exp3_8	1	89	311,296	96,895,736	20.3	648,408,237	5	29.7%	6.19
gen_regexps	1	30	335,872	2,840,152	0.3	11,492,926	0	-	6.60
primes	1	14	303,104	14,107,180	6.8	216,594,440	0	-	2.08
queens	1	14	303,104	9,300,792	2.6	109,540,247	0	-	3.59
boyer	1	1,016	352,256	21,752,256	4.2	125,068,303	0	-	5.54
boyer2	5	723	385,024	2,200,300	0.8	22,420,684	0	-	2.58
cichelli	5	246	352,256	30,731,260	11.4	381,576,703	1	2.0%	2.46
clausify	1	177	319,488	20,723,172	3.9	142,859,189	0	-	4.81
fft2	3	215	475,136	24,499,984	6.0	167,581,876	0	-	4.26
knights	5	716	352,256	708,264	0.5	20,103,715	0	-	1.22
mandel	3	348	466,944	231,301,868	22.4	670,897,904	9	1.0%	10.62
mandel2	1	222	491,520	10,617,812	1.6	47,373,219	0	-	6.63
minimax	6	257	335,872	1,973,488	0.4	10,414,070	0	-	5.48
multiplier	1	490	352,256	84,656,260	17.5	491,221,609	3	22.3%	6.01
pretty	3	265	458,752	33,080	0.0	129,221	0	-	1.65
primetest	4	276	360,448	124,957,516	93.5	5,512,615,356	5	0.1%	1.21
rewrite	1	631	393,216	21,509,044	4.5	135,238,182	0	-	4.66
sorting	2	160	327,680	413,376	0.1	2,723,426	0	-	4.59
treejoin	1	125	327,680	67,027,492	18.2	490,363,774	3	31.5%	5.40
compress	5	267	320,856	146,943,920	30.8	979,251,949	6	5.4%	4.77
fluid	18	2,391	696,416	3,980,736	0.7	21,140,623	0	-	4.42
gg	9	810	720,896	7,896,104	1.5	47,603,722	0	-	4.66
hidden	15	509	589,824	463,808,832	80.1	2,322,693,507	18	0.3%	6.59
hpg	8	2,059	630,784	63,307,176	12.0	320,360,709	2	3.9%	4.56
infer	13	556	385,024	10,357,420	5.2	141,877,744	0	-	1.78
lift	5	2,023	409,600	340,300	0.0	1,522,920	0	-	6.79
maillist	1	177	335,872	3,929,240	2.2	20,336,178	0	-	2.25
parser	1	1,383	607,368	12,324,460	3.1	104,298,216	0	-	3.64
prolog	7	538	360,448	698,636	0.1	3,869,850	0	-	4.99
reptile	13	1,519	484,440	5,345,360	0.8	26,316,368	0	-	5.99
rsa	2	74	352,256	30,994,940	19.8	1,106,435,907	1	-	1.54
veritas	32	11,147	1,114,112	377,368	0.0	1,719,029	0	-	4.07
comp_lab_zift	1	880	344,064	113,224,012	20.0	591,986,205	4	12.4%	6.04
event	1	447	311,296	42,368,948	7.8	269,719,168	1	4.6%	5.21
fft	1	408	491,520	36,953,572	4.0	101,989,972	1	13.6%	11.19
genfft	1	498	352,256	21,909,028	2.8	90,189,593	0	-	7.20
ida	1	486	319,488	52,559,492	8.2	286,774,000	2	1.9%	5.51
listcompr	1	518	319,488	71,743,480	12.8	402,726,723	3	24.6%	6.76
listcopy	1	523	319,488	79,255,540	14.6	443,998,181	3	24.4%	7.22
parstof	1	1,271	557,056	48,370,780	13.9	464,564,385	1	0.8%	3.12
sched	1	551	311,296	21,103,752	2.3	73,752,139	0	-	7.48
solid	1	1,240	581,632	67,183,572	16.0	424,638,256	2	6.4%	5.34
transform	1	1,138	466,944	206,994,208	35.5	1,154,309,303	8	0.3%	5.72
typecheck	1	654	344,064	130,980,872	27.5	882,766,418	5	2.1%	4.61
wang	1	353	458,752	28,480,820	5.0	134,129,894	1	3.7%	5.80
wave4main	1	595	466,944	221,120,860	68.2	2,134,551,209	10	8.7%	3.46

Table 2.3 `nofib` benchmark programs compiled with `ghc-0.23 -O`

Chapter 3

Local Transformations

In this chapter we describe a large set of local program transformations, all of which are implemented in the Glasgow Haskell Compiler. The transformations are presented as source-to-source transformations in a simple functional language. The idea is that by composing these simple and small high level transformations one can achieve most of the benefits of more complicated and specialised transformations, many of which are often implemented as code generation optimisations.

Many of these transformations manipulate expressions that a programmer is unlikely to write, but that are often generated by desugaring Haskell to the Core language, or by other transformations.

Many of these transformations were suggested by inspection of actual intermediate code from the Glasgow Haskell Compiler. Most of them offer very small improvements on their own, but they also have the purpose of enabling other transformations; when these transformations interact, the results achieved can be quite impressive, as we show in Section 3.8.

We classify the transformations into the following groups:

- transformations that remove Core language constructs: β-reduction (removes lambdas), `let` elimination and `case` elimination transformations (Sections 3.2 and 3.3);
- transformations that move Core language constructs: `let`-floating and `case`-floating (Sections 3.4 and 3.5);
- transformations that exploit strictness[1] (Section 3.6);

[1]Some other transformations in other sections also use strictness information.

- other transformations that do not fit in the above categories (Section 3.7).

We also present some examples of how the transformations interact (Section 3.8) and briefly discuss confluence and termination of the transformation system (Section 3.9).

In the next chapter we discuss the implementation of the transformations (Section 4.1), and present results from using the transformations (Section 4.2). We ignore the issue of name capture during the presentation of the transformations; this is discussed in the next chapter.

In Table 3 we summarise most of the transformations discussed in this chapter. Some of these transformations can only be applied when some side conditions are met. These side conditions are discussed in their respective sections.

We also present results on the effect of some transformations in this chapter, often to highlight the importance of a transformation or to compare the effect of different strategies that can be adopted for a given transformation.

3.1 Beta-reduction

An application of a lambda abstraction is always reduced:

$$\boxed{(\lambda x \texttt{ -> } \mathit{body})\ y \quad\Longrightarrow\quad \mathit{body}[y/x]}$$

This applies equally to ordinary lambda abstractions and type abstractions:

$$\boxed{(\Lambda t \texttt{ -> } \mathit{body})\ \mathit{ty} \quad\Longrightarrow\quad \mathit{body}[\mathit{ty}/t]}$$

The beta-reduction transformation is actually doing evaluation at compile time.

The Core language syntactic restriction that arguments are *always* atoms allows us to replace all occurrences of x by y without any risk of duplicating work. If we had allowed arbitrary expressions as arguments, the same transformation would have to be done in stages: if x occurred more than once in *body*, we would have to `let`-bind the argument expression to avoid duplicating it and thereby (possibly) evaluating it many times. In this case, the transformation would have to be changed to:

$$(\lambda x \texttt{ -> } \mathit{body})\ e \quad\Longrightarrow\quad \texttt{let } x \texttt{ = } e \texttt{ in } \mathit{body}$$

section	transformation	before	after
3.1	beta reduction	$(\lambda v.e)x$	$e[x/v]$
3.2.1	dead code removal	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e$	e
3.2.2	inlining	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e$	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e[e_v/v]$
3.2.3	constructor reuse	$\texttt{let}\ v\ \texttt{=}\ C\ v_1 \ldots v_n$ $\texttt{in let}\ w\ \texttt{=}\ C\ v_1 \ldots v_n\ \texttt{in}\ e$	$\texttt{let}\ v\ \texttt{=}\ C\ v_1 \ldots v_n$ $\texttt{in let}\ w\ \texttt{=}\ v\ \texttt{in}\ e$
3.3.1	`case` reduction	$\texttt{case}\ C_i\ v_1 \ldots v_n\ \texttt{of}$ $\ldots; C_i\ w_1 \ldots w_n\ \texttt{->}\ e_i; \ldots$	$e_i[v_1/w_1 \ldots v_n/w_n]$
3.3.2	`case` elimination	$\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e$	$e[v_1/v_2]$
3.3.3	`case` merging	$\texttt{case}\ v\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1$ $\ldots$ $d\ \texttt{->}\ \texttt{case}\ v\ \texttt{of}$ $\mathit{alt}_m\ \texttt{->}\ e_m$ $\ldots$	$\texttt{case}\ v\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1$ $\ldots$ $\mathit{alt}_m\ \texttt{->}\ e_m[v/d]$ $\ldots$
3.3.5	default binding elimination	$\texttt{case}\ v_1\ \texttt{of}$ $\ldots; v_2\ \texttt{->}\ e$	$\texttt{case}\ v_1\ \texttt{of}$ $\ldots; v_2\ \texttt{->}\ e[v_1/v_2]$
3.4.1	`let` float from app	$(\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e)\ x$	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e\ x$
3.4.2	`let` float from `let`	$\texttt{let}\ v\ \texttt{=let}\ w\ \texttt{=}\ e_w$ $\texttt{in}\ e_v$ $\texttt{in}\ e$	$\texttt{let}\ w\ \texttt{=}\ e_w$ $\texttt{in let}\ v\ \texttt{=}\ e_v$ $\texttt{in}\ e$
3.4.3	`let` float from `case` scrutinee	$\texttt{case}\ (\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e)\ \texttt{of}$ $\ldots$	$\texttt{let}\ v\ \texttt{=}\ e_v$ $\texttt{in case}\ e\ \texttt{of}\ \ldots$
3.5.1	`case` float from app	$\left(\begin{array}{l}\texttt{case}\ e_c\ \texttt{of}\\ \quad \mathit{alt}_1\ \texttt{->}\ e_1\\ \quad \ldots\\ \quad \mathit{alt}_n\ \texttt{->}\ e_n\end{array}\right)\ v$	$\texttt{case}\ e_c\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1\ v$ $\ldots$ $\mathit{alt}_n\ \texttt{->}\ e_n\ v$
3.5.2	`case` float from `case` (`case` of `case`)	$\texttt{case}\ \left(\begin{array}{l}\texttt{case}\ e_c\ \texttt{of}\\ \quad \mathit{alt}_{c1}\ \texttt{->}\ e_{c1}\\ \quad \ldots\\ \quad \mathit{alt}_{cm}\ \texttt{->}\ e_{cm}\end{array}\right)\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1$ $\ldots$ $\mathit{alt}_n\ \texttt{->}\ e_n$	$\texttt{case}\ e_c\ \texttt{of}$ $\mathit{alt}_{c1}\ \texttt{->}\ \texttt{case}\ e_{c1}\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1$ $\ldots$ $\mathit{alt}_n\ \texttt{->}\ e_n$ $\ldots$ $\mathit{alt}_{cm}\ \texttt{->case}\ e_{cm}\ \texttt{of}$ $\mathit{alt}_1\ \texttt{->}\ e_1$ $\ldots$ $\mathit{alt}_n\ \texttt{->}\ e_n$
3.5.3	`case` float from `let`	$\texttt{let}\ v\ \texttt{=}\ \texttt{case}\ e_c\ \texttt{of}$ $\mathit{alt}_{c1}\ \texttt{->}\ e_{c1}$ $\ldots$ $\mathit{alt}_{cm}\ \texttt{->}\ e_{cm}$ $\texttt{in}\ e$	$\texttt{case}\ e_c\ \texttt{of}$ $\mathit{alt}_{c1}\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ e_{c1}\ \texttt{in}\ e$ $\ldots$ $\mathit{alt}_{cm}\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ e_{cm}\ \texttt{in}\ e$
3.6.1	`let` to `case`	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e$	$\texttt{case}\ e_v\ \texttt{of}\ v\ \texttt{->}\ e$
3.6.2	unboxing `let` to `case`	$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e$	$\texttt{case}\ e_v\ \texttt{of}$ $C\ v_1 \ldots v_n\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ C\ v_1 \ldots v_n$ $\texttt{in}\ e$
3.7.2	eta expansion	e	$\lambda x.e\ x$

Table 3.1 Local Transformations

The beta-reduction transformation is always good, because[2]:

- ✓ it moves the execution of the beta-reduction from run-time to compile-time. This will often reduce heap allocation and execution time, as the lambda expression will not be allocated or evaluated;
- ✓ it is particularly effective in exposing other transformations, since it turns a lambda-bound variable (for which we have no information) into a `let`-bound variable (for which we may obtain some information from its right hand side). For example, if the argument variable is bound to a constructor it may enable the `case` reduction transformation (Section 3.3.1).

3.2 `let` elimination

3.2.1 Dead code removal

A `let` binding that is not referred to in its body can be removed from the program:

`let` x `=` e `in` *body* $\Longrightarrow$ *body*
x not used in *body*

The same happens for `let rec`s in which none of its bindings occur in its body:

bindings `in` *body* $\Longrightarrow$ *body*
none of the binders in *bindings* is used in *body*

The dead code removal transformation:

- ✓ Saves the allocation of the closure for the `let`, therefore reducing heap allocation.
- ✓ Reduces code size.

Notice that as we are performing this transformation in a side-effect-free language there is no danger of accidentally discarding a right hand side that performs a side effect. side effects like SML.

[2]Advantages are marked with ✓ and disadvantages with ×. □ indicates the effect may be good or bad.

3.2.2 Inlining

Inlining occurs when we replace some or all occurrences of a `let`-bound variable by its right hand side:

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \ldots x \ldots \Longrightarrow \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \ldots e \ldots$$

Due to the Core syntax, inlining can only be performed if x occurs in a function position or on its own, i.e. it cannot be performed if x occurs in an argument position.

The main advantages that come from inlining are:

- ✓ it enables dead code elimination if all occurrences are inlined.
- ✓ the definition is now available in the place of its use, allowing transformations such as β-reduction (Section 3.1) to occur.
- ✓ better (local) context information, e.g. more things may be known to be evaluated in the place of use, allowing transformations such as `case` reduction (Section 3.3.1) to occur. For example, in the expression

  ```
  let v = case x of (a,b) -> a
  in case x of (c,d) -> ...v...
  ```

  if `v` is inlined we will be able to know in the (new) local context that `x` was already evaluated, and therefore avoid evaluating the identical `case`s twice.

But inlining also has the following risks

- × code duplication, if expressions are inlined when they occur multiple times.
- × work duplication if the inlining is not done carefully (redex copying).

All these points, including the key issue of choosing which expressions to inline are discussed in detail in Chapter 6.

3.2.3 Constructor reuse

The constructor reuse transformation avoids allocating a new object (constructor) when there is an identical object in scope. This may occur in two circumstances:

1. There is an identical constructor expression bound by a `let`:

$$\begin{array}{l}\texttt{let } v \texttt{ = } C\ v_1 \ldots v_n \\ \texttt{in } \ldots\ C\ v_1 \ldots v_n\ \ldots\end{array} \implies \begin{array}{l}\texttt{let } v \texttt{ = } C\ v_1 \ldots v_n \\ \texttt{in } \ldots\ v\ \ldots\end{array}$$

2. There is an identical constructor expression "bound" by a variable `case` scrutinee:

$$\begin{array}{l}\texttt{case } v \texttt{ of} \\ \quad \ldots \\ \quad C\ v_1 \ldots v_n \texttt{ ->} \ldots\ C\ v_1 \ldots v_n\ \ldots \\ \quad \ldots\end{array} \implies \begin{array}{l}\texttt{case } v \texttt{ of} \\ \quad \ldots \\ \quad C\ v_1 \ldots v_n \texttt{ ->} \ldots\ v\ \ldots \\ \quad \ldots\end{array}$$

The main characteristics of this transformation are:

- ✓ It avoids the heap allocation of an object when an already existing object can be used instead.
- □ It increases the lifetime of objects, possibly affecting heap residency.

In the Glasgow Haskell Compiler, since we keep type information during compilation, we can only implement this transformation when it preserves type correctness. In particular, we are not able to reuse constructors in cases like this:

```
data Either a b = Left a | Right b
f :: Either String String -> Either String Int
f x = case x of
        Left y -> Left y
        _      -> Right 5
```

Although the value for `x` and the resulting expression (`Left y`) seem to be the same (and actually will have the same "form" when code is generated), they have different types: `x` has type `Either String String`, while `Left y` on the right hand side of the `case` alternative has type `Either String Int`.

Depending on the position of the eliminated constructor, there are some other issues involved:

- `let` right hand side: this is where the biggest benefit from reusing constructors comes from, according to our experiments, since we will actually end up eliminating a `let`.
- `case` scrutinee: other transformations (Section 3.3.1) eliminate a `case` if it is scrutinising a constructor or a variable known to be bound to a constructor, therefore this case is not relevant.
- `case` alternative, `let` body or lambda body: in these cases the cost of reusing a constructor may sometimes not be worthwhile. For a 0-arity constructor, for example, there would be no space saved (since 0-arity constructors are allocated statically) and we are still introducing an extra indirection, which is less efficient to execute:

```
case y of                case y of
  True -> True     =/=>    True  -> y
  False -> False           False -> y
```

  Now consider

```
        f = \x -> case x of
                    (y:ys) -> y:ys
                    []     -> ...
```

  Is it a good idea to replace the right hand side `y:ys` with `x`? This actually depends on the specific compiler technology being used. In the STG machine we believe not. Another reason not to do this occurs in the following code fragment (from a real program):

```
max = \ x#  y# -> let a = I#! x#
                  in case (tagCmp x# y#) of
                       _LT -> I#! y#
                       _EQ -> a
                       _GT -> a
```

  The `a` is allocated regardless of which branch of the `case` is taken. We would be better off inlining it[3].

[3]Actually, as we will see later in this chapter, this particular `let` can be compiled very efficiently (into a jump), and therefore the code isn't as bad as it looks.

The current strategy in the Glasgow Haskell Compiler is to inline all known-form constructors, and only do the reverse (turn a constructor application back into a variable) when we know it is in a `let` right hand side. This decision was supported also by experiments in which we did reuse constructors more aggressively, and the results were that the effects on heap usage were very small and more often than not the number of instructions executed was increased with the more aggressive strategy, as can be seen in Table 3.2.

Constructor Reuse Total Instructions Executed			
program	never	in `let` rhs	always
`parser`	1.00	0.97	0.97
`solid`	1.00	0.97	0.97
`wang`	1.00	0.97	0.97
`event`	1.00	0.99	1.00
`gen_regexps`	1.00	0.99	0.99
`knights`	1.00	0.99	0.99
`prolog`	1.00	0.99	1.00
`sched`	1.00	0.99	0.99
`boyer`	1.00	1.00	1.01
`clausify`	1.00	1.00	1.05
`fluid`	1.00	1.00	1.01
`multiplier`	1.00	1.00	1.04
`rewrite`	1.00	1.00	1.02
`transform`	1.00	1.00	1.01
`treejoin`	1.00	1.00	1.01
`compress`	1.00	1.01	1.01
30 other progs.	1.00	1.00	1.00
Minimum	-	0.97	0.97
Maximum	-	1.01	1.05
Geometric mean	-	1.00	1.00

Constructor Reuse Total Heap Allocated			
program	never	in `let` rhs	always
`compress`	1.00	0.64	0.64
`knights`	1.00	0.77	0.77
`parser`	1.00	0.78	0.77
`solid`	1.00	0.81	0.81
`event`	1.00	0.90	0.90
`sched`	1.00	0.92	0.92
`wang`	1.00	0.93	0.93
`boyer2`	1.00	0.97	0.95
`pretty`	1.00	0.97	0.97
`lift`	1.00	0.98	0.98
`transform`	1.00	0.98	0.98
`treejoin`	1.00	0.98	0.98
`comp_lab_zift`	1.00	0.99	0.99
`fluid`	1.00	0.99	1.00
`gg`	1.00	0.99	0.99
`hpg`	1.00	0.99	0.99
`infer`	1.00	0.99	0.99
`maillist`	1.00	0.99	0.99
`minimax`	1.00	0.99	0.99
`prolog`	1.00	0.99	1.01
`rewrite`	1.00	0.99	0.99
`typecheck`	1.00	0.99	0.99
24 other progs.	1.00	1.00	1.00
Minimum	-	0.64	0.64
Maximum	-	1.00	1.01
Geometric mean	-	0.96	0.96

Table 3.2 Constructor Reuse: instructions executed and bytes allocated

The effect of the constructor reuse transformation (in `let` right hand sides only) on residency is presented in Table 3.3. We forced a garbage collection at every 1Mbytes allocated, and restricted our sample to programs that performed at least 5 garbage collections (34 programs), so that we could have at least 5 samples.

The results showed that actually the residency was often reduced. This can be explained by the fact that if two identical constructor expressions' lifetime overlap we would be better off with only one copy.

Constructor Reuse Residency					
program	off		on		ratio
	GCs	residency	GCs	residency	
`hidden`	463	358912	462	326848	0.91
`parser`	15	939296	12	872480	0.93
`sched`	22	2324	21	2204	0.95
`gg`	7	383412	7	375264	0.98
`solid`	83	533912	67	521760	0.98
`comp_lab_zift`	112	1239712	111	1228664	0.99
`event`	49	4052008	44	4010772	0.99
`genfft`	21	3544	21	3496	0.99
`rewrite`	21	17960	21	17700	0.99
`clausify`	20	39748	20	39952	1.01
`multiplier`	85	1804280	85	1813728	1.01
`infer`	10	1972016	10	2010228	1.02
`typecheck`	132	10284	131	10596	1.03
21 other progs.	-	-	-	-	1.00
Minimum	-	-	-	-	0.91
Maximum	-	-	-	-	1.03
Geometric Mean	-	-	-	-	0.99

Table 3.3 Constructor Reuse: Residency

Sometimes the code where this transformation is applied comes directly from the source code, from places where the programmer could use `@`-patterns to achieve the same effect, for example:

```
f (a:as) = ... (a:as) ... a ... as ...
```

could have been written

```
f l@(a:as) = ... l ... a ... as ...
```

3.3 `case` elimination

3.3.1 `case` reduction

There are three instances of the `case` reduction transformation:

1. If a `case` expression scrutinises a constructor application, it can be eliminated:

$$\begin{array}{l}\texttt{case}\ C\ v_1 \ldots v_n\ \texttt{of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n\ \texttt{->}\ e \\ \quad \ldots \end{array} \quad \Longrightarrow \quad e[v_i/x_i]_{i=1}^{n}$$

2. The `case` expression might be scrutinising a variable which has already been scrutinised:

$$\begin{array}{l}\texttt{case}\ v\ \texttt{of} \\ \ \ldots \\ \ C\ x_1 \ldots x_n\ \texttt{->}\ \ldots \left(\begin{array}{l}\texttt{case}\ v\ \texttt{of} \\ \ \ldots \\ \ C\ y_1 \ldots y_n\ \texttt{->}\ e \\ \ \ldots\end{array}\right) \\ \ \ldots \end{array} \Longrightarrow \begin{array}{l}\texttt{case}\ v\ \texttt{of} \\ \quad \ldots \\ \quad C\ x_1 \ldots x_n\ \texttt{->}\ \ldots e[x_i/y_i]_{i=1}^{n} \\ \quad \ldots \end{array}$$

3. It might be scrutinising a variable which is `let`-bound to a constructor application:

$$\begin{array}{l}\texttt{let}\ x\ \texttt{=}\ C\ x_1 \ldots x_n \\ \texttt{in}\ \ldots \left(\begin{array}{l}\texttt{case}\ x\ \texttt{of} \\ \quad \ldots \\ \quad C\ y_1 \ldots y_n\ \texttt{->}\ e \\ \quad \ldots\end{array}\right) \ldots \end{array} \Longrightarrow \begin{array}{l}\texttt{let}\ x\ \texttt{=}\ C\ x_1 \ldots x_n \\ \texttt{in}\ \ldots e[x_i/y_i]_{i=1}^{n} \ldots \end{array}$$

   This third transformation is useful when x occurs many times in its scope, so the `let` expression might not be inlined[4].

Again, since arguments to constructors are *always* atoms, no loss of sharing occurs. As with function arguments, if we allowed arbitrary expressions as constructor arguments we would need to use `let` bindings instead of substitution to preserve the sharing properties.

The `case` reduction transformations are always good:

✓ they eliminate redundant evaluations that would be done at run-time;

[4]Since in Glasgow Haskell Compiler we always inline constructors, this version is not needed.

✓ they expose opportunities for other transformations. We will see how this occurs in Sections 3.4.3 and 3.5.2.

A related transformation

If the `case` scrutinee matches only the default alternative, we can eliminate the `case` by `let`-binding the default variable to the constructor:

$$\begin{array}{l}\texttt{case}\ C\ v_1 \ldots v_n\ \texttt{of} \\ \ w\ \texttt{->}\ e\end{array} \quad\Longrightarrow\quad \begin{array}{l}\texttt{let}\ w\ \texttt{=}\ C\ v_1 \ldots v_n \\ \texttt{in}\ e\end{array}$$

This is more efficient because there would be no evaluation done by the `case`, as the `case` scrutinee is in weak head normal form. Therefore:

✓ we are saving the cost of entering an expression that is already in weak head normal form;

✓ `w` may be eliminated by the constructor reuse transformation.

3.3.2 `case` elimination

If a primitive `case` is scrutinising a variable, that variable is guaranteed to be already evaluated (since it is an unboxed value). Therefore the following is a valid transformation:

$$\texttt{case}\ v_1\ \texttt{of}\ v_2\ \texttt{->}\ e \quad\Longrightarrow\quad e[v_1/v_2]$$

As above, this transformation is eliminating a redundant evaluation. The transformation is also valid if we know that the variable was already evaluated, or if we know v_2 is used strictly in e. This is another example of a transformation that is done in an obscure way in code generators (e.g. [Pey87], pp. 352).

If we applied this transformation regardless of any conditions we could only *improve* termination, that is, possibly transform a failing program into a non-failing one[5].

When this transformation is not applied, the default binding elimination transformation (Section 3.3.5) may be applied instead.

[5]The Glasgow Haskell Compiler provides a flag to enable this transformation.

3.3.3 case merging

The `case` merging transformation combines `case`s that scrutinise the same variable into a single `case` expression:

$$
\begin{array}{l}
\texttt{case } x \texttt{ of} \\
\quad p_1 \texttt{ -> } e_1 \\
\quad \ldots \\
\quad p_n \texttt{ -> } e_n \\
\quad d \texttt{ -> case } x \texttt{ of} \\
\qquad\qquad p_o \texttt{ -> } e_o \\
\qquad\qquad \ldots \\
\qquad\qquad p_q \texttt{ -> } e_q
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\texttt{case } x \texttt{ of} \\
\quad p_1 \texttt{ -> } e_1 \\
\quad \ldots \\
\quad p_n \texttt{ -> } e_n \\
\quad p_o \texttt{ -> } e_o[x/d] \\
\quad \ldots \\
\quad p_q \texttt{ -> } e_q[x/d]
\end{array}
$$

Consider the following code fragment:

```
g :: Int -> Int -> Int
g x y = f x + f y

f 0 = 1
f 1 = 2
f 2 = 3
```

As the type of `f` is not given, a Haskell compiler will assume it is an overloaded function, and therefore the code generated for `f` (with a standard compilation of overloading [WB89, HHaPW92, Aug93]) could be:

```
f = \ dict -> \ x  -> case eq dict x 0 of
                        True -> 1
                        False -> case eq dict x 1 of
                                     True -> 2
                                     False -> case eq dict x 2 of
                                                  True -> 3
                                                  False -> fail
```

If we knew that `f` had type `Int -> Int`, the code generated would be:

```
f = \ x -> case x of
             0 -> 1
             1 -> 2
             2 -> 3
             _ -> fail
```

which is much more efficient. But although we cannot transform the first version into the second directly, if we inline the call to `f` in `g` (or even decide to generate a specialised version of `f` with type `Int -> Int`), we get the following:

```
f = \ x  -> case eqInt x 0 of
              True -> 1
              False -> case eqInt x 1 of
                         True -> 2
                         False -> case eqInt x 2 of
                                    True -> 3
                                    False -> fail
```

which uses the `eqInt` function but still compares `x` very inefficiently. What to do?

1. Have the constant folder (Section 3.7.1) recognise the following identity:

   ```
                            case v of
   eqInt v k       ==>        k -> True
                              _ -> False
   ```

   where `v` is a variable and `k` is an explicit constant (e.g. `1`, `2`, etc.). We will then get three instances of the `case` of `case` transformation (Section 3.5.2), which eventually will give us the following code:

   ```
   f = \ x  -> case x of
                 0 -> 1
                 _ -> case x of
                        1 -> 2
                        _ -> case x of
                               2 -> 3
                               _ -> fail
   ```

2. Apply the `case` merging transformation (twice). This will give us the efficient version of `f` we wanted:

   ```
   f = \ x -> case x of
                0 -> 1
                1 -> 2
                2 -> 3
                _ -> fail
   ```

3.3.4 `case` of `error`

`error` is a predefined function in Haskell, usually associated with pattern matching failures and other run-time errors. Its semantic value is the same as $\perp$.

Sometimes we may end up with `error` as a `case` scrutinee, to which we can apply the following transformation

$$\texttt{case}\ (\texttt{error}\ E)\ \texttt{of}\ \ldots \implies \texttt{error}\ E$$

The `case` of `error` transformation is often exposed by the `case` of `case` transformation (Section 3.5.2). Consider

```
case (hd xs) of {True  -> E1; False -> E2}
```

After inlining `hd`, we get

```
case (case xs of [] -> error "hd"; (x:_) -> x) of True  -> E1
                                                  False -> E2
```

Now doing `case` of `case` we get

```
let e1 = E1 ; e2 = E2
in case xs of
     []    -> case (error "hd") of { True -> e1; False -> e2 }
     (x:_) -> case x            of { True -> e1; False -> e2 }
```

Now the `case` of `error` transformation springs to life, after which we can inline `e1` and `e2` to get the efficient result

```
case xs of []    -> error "hd"
           (x:_) -> case x of {True -> E1; False -> E2}
```

The type of `error` in these two expressions is different, because we are replacing `case` $\perp$ `of` ... by $\perp$. This transformation not only reduces code size, but may enable other transformations (e.g. inlining, as above).

The Glasgow Haskell Compiler is clever enough to notice "disguised" forms of error expressions and handle them in the same way (e.g. `let`-bound error expressions, functions that always return errors and `case`s with all alternatives returning errors).

3.3.5 Default binding elimination

$$\begin{array}{l} \texttt{case}\ v_1\ \texttt{of} \\ \quad \ldots \quad v_2\ \texttt{->}\ e \end{array} \Longrightarrow \begin{array}{l} \texttt{case}\ v_1\ \texttt{of} \\ \quad \ldots \quad v_2\ \texttt{->}\ e[v_1/v_2] \end{array}$$

The code generator can generate better code if the default variable is not used in its right hand side (it does not need to bind the result of the `case` evaluation to the default variable).

But there is a possible disadvantage of this transformation: it increases the number of occurrences of v_1, and therefore may avoid some inlining from taking place. Actually as we always inline variables bound to constructors, there is no risk that we may miss a `case` reduction due to this transformation.

3.3.6 Dead alternative elimination

Dead alternative elimination is similar to the `case` reduction transformation, but deals with the case when all we know about a variable is that it is *not* bound to some constructors. Assuming x is *not* bound to constructor C_k, we have:

$$\begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad C_1\ \ldots\ \texttt{->}\ e_1 \\ \quad \ldots \\ \quad C_l\ \ldots\ \texttt{->}\ e_l \end{array} \Longrightarrow \begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad C_1\ \ldots\ \texttt{->}\ e_1 \\ \quad \ldots \\ \quad C_{k-1}\ \ldots\ \texttt{->}\ e_{k-1} \\ \quad C_{k+1}\ \ldots\ \texttt{->}\ e_{k+1} \\ \quad \ldots \\ \quad C_l\ \ldots\ \texttt{->}\ e_l \end{array}$$

We might know that x is not bound to a particular constructor because of an enclosing `case`:

```
case x of C ... -> E1
          other -> E2
```

Inside `E1` we know that `x` *is* bound to `C`. However inside `E2` all we know is that `x` is *not* bound to `C`.

This applies to unboxed `case`s also, in the obvious way.

The importance of this transformation is that:

- ✓ it reduces code size;
- ✓ it may enable inlining, as it reduces the number of occurrences of variables;
- ✓ it may enable other `case` elimination transformations.

This third possibility is less obvious, but usually occurs with relation to operations that check for invalid arguments (out of range arguments). Let us suppose we have an expression like

```
(x `mod` y) + (x `div` y)
```

the `mod` and `div` operations do not accept a second argument with value `0`. Supposing this check was performed before the actual operation takes place, we would end up with a code fragment similar to

```
... case y# of
      0# -> error "mod"
      m# -> ... case y# of
                  0# -> error "div"
                  n# -> ...
```

Clearly if we know in the inner `case` that `y#` cannot have a value of `0` we can eventually eliminate this inner `case` completely.

An example of the use of this transformation is presented in Section 3.8.3.

3.4 Floating `let`s outwards

The transformations in this section increase the scope of `let`-bindings in order to turn the expression into a more efficient form, to increase the possibility of other transformations becoming applicable, or both.

3.4.1 `let` floating from application

A `let`-binding can be floated out of an application to facilitate other transformations, without introducing (or removing) extra work:

$$\texttt{(let (rec)}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e\texttt{)}\ x \implies \texttt{let (rec)}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e\ x$$

An example of how this transformation exposes other transformations occurs when the `let` body is a lambda expression:

```
(let x = ... in \a -> body) y
```

in this case an opportunity for β-reduction occurs if the transformation is applied:

```
let x = ... in (\a -> body) y
```

3.4.2 `let` floating from `let` right hand side

`let` floating from a `let` right hand side is a transformation that moves bindings defined in the right hand side of a `let` to outside the `let`:

$$\begin{array}{l} \texttt{let}\ x\ \texttt{=let (rec)}\ bind \\ \qquad\quad \texttt{in}\ e_x \\ \texttt{in}\ b \end{array} \implies \begin{array}{l} \texttt{let (rec)}\ bind \\ \texttt{in let}\ x\ \texttt{=}\ e_x \\ \quad \texttt{in}\ b \end{array}$$

$$\begin{array}{l} \texttt{let rec}\ x\ \texttt{= let (rec)}\ binds \\ \qquad\qquad\quad \texttt{in}\ e_x \\ \texttt{in}\ b \end{array} \implies \begin{array}{l} \texttt{let rec} \left\{ \begin{array}{l} binds \\ x\ \texttt{=}\ e_x \end{array} \right\} \\ \texttt{in}\ b \end{array}$$

To illustrate our goal in floating out `let`s from `let` right hand sides consider the following simple expression:

```
let x = [1,2,3] in E
```

A possible translation into the Core language, which makes explicit the three closures, is:

```
let x = let v1 = let v2 = 3:[]
                 in 2:v2
        in 1:v1
in E
```

In this translation:

- x and v1 are not in weak head normal form, therefore they will be updated if they are evaluated, but v2 is in weak head normal form and therefore requires no update;
- if the closure x is entered (evaluated) the closure v1 is allocated, and if v1 is entered then v2 is allocated. Although this strategy saves heap space (i.e. allocates fewer closures) if v1 is never entered (since v2 is never allocated), the cost of allocating each closure separately implies one heap check for each such allocation.

An alternative for the translation above is:

```
let v2 = 3:[] ;
    v1 = 2:v2 ;
    x  = 1:v1
in E
```

This strategy — floating the internal lets to an outer level — has the following advantages:

✓ A single heap check is done for the three allocations.

✓ Weak head normal forms are exposed. All three closures are weak head normal forms and therefore no updates are required.

✓ It may expose other transformations, e.g. case reduction:

```
let x = let y = 1:[]              let y = 1:[]
        in 2:y                        x = 2:y          let y = 1:[]
in case x of                ==>   in case x of    ==>     x = 2:y
     (a:as) -> as                      (a:as) -> as    in y
     []     -> []                      []     -> []

                            ==>   let y = 1:[]    ==> 1:[]
                                  in y
```

Unfortunately it is not always good to float lets:

☐ We may allocate more closures than are really needed. In our first example, if we do not need the value of x during the evaluation of the expression E, we would only allocate the closure for x, instead of allocating three closures (x, v1 and v2). But if the value of x is demanded we would be better off with the second translation. As we cannot predict precisely which closures will be evaluated, we have to decide how to take advantage of let floating, while minimising the risks of extra heap allocation.

There are three possible strategies for floating lets out of lets, which we discuss below.

Float out of strict lets

Floating lets out of strict lets consists of using strictness information to decide if we want to float out of a particular let. If a let is used in a strict context we know that it will be evaluated and therefore lets defined immediately within it are guaranteed to be allocated. Floating out of these lets we:

- ✓ reduce the number of heap checks, since more closures will be allocated at the same time;
- ✓ do not increase heap allocation, since the let is guaranteed to be evaluated;
- ✓ possibly expose weak head normal forms, reducing the number of updates;
- ✓ possibly expose opportunities for transformations, as presented above;
- □ modify the number of free variables. In the STG machine, each free variable has to be saved in the stack when entering a closure (see [Pey92]). More free variables means more stack saves. In the example below, u is a free variable in v after being floated, therefore v has more free variables. But w has less free variables after x is floated, as although x is now a free variable in w, y and z are not. Also, the number of free variables in a closure affects the size of the closure in the heap.

```
let v = let u = 1          let u = 1 ;
        in u + 1 ;             v = u + 1 ;
    y = 2 ;                    y = 2
    z = 3 ;          ==>       z = 3 ;
    w = let x = y + z          x = y + z ;
        in x + 1               w = x + 1
in ...                     in ...
```

× may increase heap residency, due to the early allocation of closures that would only be allocated later, or due to the change in the number of free variables.

When the `let`-binding is guaranteed to be demanded (strict) a better result is achieved if the strictness information is used to implement the `let`-to-`case` transformation (Section 3.6.1), therefore this is not a very useful option.

Float out of `lets` to expose weak head normal forms

Floating `lets` out of `lets` to expose weak head normal forms takes advantage of the fact that weak head normal form `lets` (closures) are cheaper in the sense that they do not require updates, which are rather expensive. With this strategy we risk building unnecessary closures (if they are not demanded), but we benefit from creating weak head normal form closures, instead of updatable ones. An example of the risks of this strategy can be seen by looking again at our first example:

```
let x = let v1 = let v2 = 3:[]       let v2 = 3:[] ;
                 in 2:v2        ==>      v1 = 2:v2 ;
        in 1:v1                          x  = 1:v1
in f x                               f x
```

With the standard translation, if `f` is the `head` function (which returns the first element of a list), to get `head x` we would allocate `x`, enter it, allocate `v1`, update `x` with `1:v1` and then we get the result (`1`). With the `let` floated version we allocate the three closures `v2`, `v1` and `x` together (one heap check) and we need no updates, as they are in weak head normal form. But as we are computing only the head of the list, we would not need to allocate `v2`. Therefore the `let` floated version would only be good if the cost of the update and heap check was greater than the cost of allocating `v2`. If `f` happens to be the `last` function (which returns the last element of a list), we would need to enter the three closures, the floated version would certainly be better. With this strategy we:

✓ reduce the number of heap checks, since more closures will be allocated at the same time;

✓ expose weak head normal forms, reducing the number of updates;

✓ possibly expose opportunities for transformations, e.g. `case` reduction and constructor reuse;

- □ modify the number of free variables;
- × may increase heap allocation, depending on whether the closures will be demanded or not;
- × may increase heap residency, due to the early allocation of closures that would only be allocated later (or never), or due to the change in the number of free variables.

Always float `lets` out of `lets`

By always floating `lets` out of `lets` we increase the risk of allocating unnecessary closures but expect that most of the closures will be entered and therefore we are minimising heap checks and still having the same advantages and disadvantages of the previous strategy:

- ✓ possibly reduce number of heap checks even further, since more closures will be allocated at the same time;
- × may increase heap allocation, depending on whether the closures will be demanded or not;
- ✓ possibly expose weak head normal forms, reducing the number of updates;
- ✓ possibly expose opportunities for transformations, e.g. `case` reduction, constructor reuse and inlining;
- □ modify the number of free variables;
- × may increase heap residency, due to the early allocation of closures that would only be allocated later (or never), or due to the change in the number of free variables.

We try to exploit not only the previously described cases when we are either sure to enter a closure, or we are trying to avoid building updatable closures, but also the simple fact that if closures are entered at all, it would have been cheaper to allocate them in groups (doing a single heap check) rather than one at a time.

Comparing the different strategies

Of course we can never get an optimal decision, as the result will always depend on whether the `let` will be actually used (in which case the transformation is a win) or not (in which case the transformation will worsen the code). We know the benefits are bigger if we are exposing a weak head normal form (because we will be avoiding updates) and much more modest otherwise (we are only saving heap checks).

We have experimented with the three different strategies for floating `lets` out of `let` right hand sides (never float, float to expose weak head normal form, always float), as presented in Figure 3.4 (As we mentioned before, when the `let`-binding is strict a better result is achieved with the `let`-to-`case` transformation (Section 3.6.1), therefore we did not experiment with this option). All these results include the effect of the `let` floating inwards transformation we present in Chapter 5, which actually increases the number of `lets` occurring in `let` right hand sides. We have obtained similar effects if that transformation is turned off. We discuss the interaction of these seemingly incompatible transformations in Chapter 5.

As we expected, exposing weak head normal forms is a worthwhile improvement on not doing any floating. Always floating, on the other hand, has mixed results, and therefore has a higher risk of actually making programs worse. This lead us to adopt the option of floating to expose weak head normal forms as a worthwhile optimisation in our compiler.

The average closure size (measured during execution) of the programs was on average 2% smaller when floating to expose weak head normal forms than with no floating at all. Only one of the programs increased its average closure size, by 1%.

The effect on updates was much more dramatic, with some programs reducing the number of updates by up to 48%, and on average performing 11% fewer updates when floating to expose weak head normal forms compared to not floating at all.

3.4.3 `let` floating from `case` scrutinee

The benefit of floating a `let` from a `case` scrutinee comes from exposing other transformations, and not directly from the transformation itself:

$$\texttt{case}\left(\begin{array}{l}\texttt{let(rec)}\ v\ \texttt{=}\ e_v\\ \texttt{in}\ e\end{array}\right)\texttt{of}\ alts \quad\Longrightarrow\quad \begin{array}{l}\texttt{let(rec)}\ v\ \texttt{=}\ e_v\\ \texttt{in case}\ e\ \texttt{of}\ alts\end{array}$$

let floating from let Total Instructions Executed			
program	never float	expose WHNF	always float
sched	1.00	0.87	0.87
hidden	1.00	0.90	0.90
infer	1.00	0.90	0.90
prolog	1.00	0.90	0.89
queens	1.00	0.91	0.91
solid	1.00	0.91	0.90
wang	1.00	0.91	0.88
knights	1.00	0.92	0.92
sorting	1.00	0.92	0.92
pretty	1.00	0.94	0.94
rewrite	1.00	0.94	0.93
boyer	1.00	0.95	0.95
cichelli	1.00	0.95	0.95
lift	1.00	0.95	0.94
boyer2	1.00	0.96	0.96
compress	1.00	0.96	0.96
fluid	1.00	0.96	0.97
gg	1.00	0.96	0.96
reptile	1.00	0.96	0.95
genfft	1.00	0.97	0.96
ida	1.00	0.97	0.97
maillist	1.00	0.97	0.98
comp_lab_zift	1.00	0.98	0.97
fft	1.00	0.98	0.98
hpg	1.00	0.98	0.98
mandel	1.00	0.98	0.98
parser	1.00	0.98	0.95
transform	1.00	0.98	0.98
event	1.00	0.99	0.98
listcompr	1.00	0.99	0.99
typecheck	1.00	0.99	0.93
veritas	1.00	0.99	0.98
clausify	1.00	1.00	1.02
listcopy	1.00	1.00	0.99
minimax	1.00	1.00	0.99
parstof	1.00	1.00	0.95
wave4main	1.00	1.00	1.07
treejoin	1.00	1.09	1.04
8 other progs.	1.00	1.00	1.00
Minimum	-	0.87	0.87
Maximum	-	1.09	1.07
Geometric mean	-	0.97	0.96

let floating from let Total Heap Allocated			
program	never float	expose WHNF	always float
wang	1.00	0.88	0.82
compress	1.00	0.91	0.92
infer	1.00	0.91	0.91
prolog	1.00	0.92	0.96
solid	1.00	0.92	0.89
cichelli	1.00	0.93	0.93
queens	1.00	0.93	0.93
rewrite	1.00	0.93	0.93
boyer	1.00	0.94	1.02
hidden	1.00	0.96	0.96
fluid	1.00	0.97	1.03
lift	1.00	0.97	0.96
parstof	1.00	0.98	1.00
pretty	1.00	0.98	0.97
hpg	1.00	0.99	1.02
listcompr	1.00	0.99	1.00
listcopy	1.00	0.99	1.00
mandel	1.00	0.99	0.99
parser	1.00	0.99	1.03
clausify	1.00	1.00	1.22
gen_regexps	1.00	1.00	1.04
gg	1.00	1.00	1.01
ida	1.00	1.00	0.99
knights	1.00	1.00	1.02
maillist	1.00	1.00	1.08
minimax	1.00	1.00	0.99
multiplier	1.00	1.00	1.01
wave4main	1.00	1.00	1.83
genfft	1.00	1.01	1.01
reptile	1.00	1.01	1.00
sched	1.00	1.01	1.01
boyer2	1.00	1.02	1.02
event	1.00	1.02	1.03
typecheck	1.00	1.02	1.00
fft	1.00	1.03	1.07
transform	1.00	1.07	1.07
comp_lab_zift	1.00	1.14	1.15
treejoin	1.00	1.16	1.41
8 other progs.	1.00	1.00	1.00
Minimum	-	0.88	0.82
Maximum	-	1.16	1.83
Geometric mean	-	0.99	1.02

Table 3.4 let floating: instructions executed and bytes allocated

An example of a transformation that is exposed by floating a `let` from a `case` scrutinee occurs when the `let` body is an explicit constructor:

```
    case (let x = ... in C a b) of C c d -> body
==>
    let x = ... in case C a b of C c d -> body
==>
    let x = ... in body[a/c,b/d]
```

In this case it exposed the `case` reduction transformation (Section 3.3.1).

3.4.4 Other `let` floating transformations

There are a few other constructors from which a `let` could be floated from, namely:

- from lambdas: this is better done as a global transformation (full laziness), which we discuss in Section 5.2.
- from `case` alternatives: If there are multiple alternatives there is a major problem in doing that: We will be allocating the `let` regardless of which alternative will be taken, instead of only if a particular one is taken. This will increase heap allocation, and therefore is not a good idea. If there is a single `case` alternative then we might gain something if the `let` is going to join other `let`s and be allocated using a single heap check. On the other hand one may actually lose opportunities for transformations like `case` reduction if the `let` right hand side happens to scrutinise the same variable of the `case` it is being floated from. For more details on this issue see Section 5.1, where we present the opposite transformation.
- from `let` body: this amounts to swapping the order of allocation of the `let`s involved, and therefore usually brings no benefits. Also it is only possible if the inner `let` right hand side does not mention binders introduced by the outer one. One instance in which the ordering of the `let`s may be relevant occurs in the following example:

  ```
  let a = case x of (c,d) -> c
  in let b = case x of (c,d) -> d
     in e
  ```

If only `b` is used strictly in `e`, we would be able to use the `let` to `case` transformation (Section 3.6.1) to improve the code, but `a` would get no benefit from that:

```
let a = case x of (c,d) -> c
in case x of (c,d) -> let b = d in e
```

On the other hand, if `a` was used strictly in `e`, the same transformation would allow us to eliminate the inner `case`, resulting in more efficient code:

```
case x of (c,d) -> let a = c
                   in let b = d in e
```

Actually, the floating inwards transformation (described in Chapter 5) would eventually lead to the same improved program.

3.5 Floating cases outwards

`case`s have similar properties to `let`s except for being strict. But this should not forbid us from doing similar transformations for `case`s.

3.5.1 `case` floating from application

A `case` expression can be floated out past an application:

$$\left(\begin{array}{l}\texttt{case } e \texttt{ of} \\ \quad p_1 \texttt{ -> } e_1 \\ \quad \ldots \\ \quad p_n \texttt{ -> } e_n\end{array}\right) x \quad \Longrightarrow \quad \begin{array}{l}\texttt{case } e \texttt{ of} \\ \quad p_1 \texttt{ -> } e_1\ x \\ \quad \ldots \\ \quad p_n \texttt{ -> } e_n\ x\end{array}$$

The main points about `case` floating from application are:

✓ to try to expose other transformations, e.g. β-reduction if any e_i is a λ-expression.

× it has only a small amount of code duplication, since x is always an atom.

3.5.2 `case` of `case` (`case` floating from `case` scrutinee)

The `case` of `case` transformation simplifies expressions in which a `case` is the scrutinee of another `case` expression:

$$
\begin{array}{l}
\texttt{case} \left(\begin{array}{l} \texttt{case}\ e_c\ \texttt{of} \\ \quad alt_{c1}\ \texttt{->}\ e_{c1} \\ \quad \ldots \\ \quad alt_{cm}\ \texttt{->}\ e_{cm} \end{array} \right) \texttt{of} \\
\quad alt_1\ \texttt{->}\ e_1 \\
\quad \ldots \\
\quad alt_n\ \texttt{->}\ e_n
\end{array}
\Longrightarrow
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{case}\ e_c\ \texttt{of}} \\
\ alt_{c1}\ \texttt{->} & \texttt{case}\ e_{c1}\ \texttt{of} \\
 & \quad alt_1\ \texttt{->}\ e_1 \\
 & \quad \ldots \\
 & \quad alt_n\ \texttt{->}\ e_n \\
\ \ldots & \\
\ alt_{cm}\ \texttt{->} & \texttt{case}\ e_{cm}\ \texttt{of} \\
 & \quad alt_1\ \texttt{->}\ e_1 \\
 & \quad \ldots \\
 & \quad alt_n\ \texttt{->}\ e_n
\end{array}
$$

A particular instance of the `case` of `case` transformation is described in [Aug87] and in [Kel89] (using `if`s). They were concerned, among other things, with short-circuiting boolean conditionals. For example, consider the expression:

```
if (b1 && b2) then e1 else e2
```

where `b1` and `b2` are boolean expressions, and `&&` is boolean conjunction. If `b1` turns out to be false there is no point in testing `b2`, because the result will be `e2` in either case. The definition of `&&` encapsulates this property:

```
(&&) b1 b2 = case b1 of
                True  -> b2
                False -> False
```

Let us now try some transformations. For a start, the if-then-else construct is just syntactic sugar for a `case` expression, so the original expression is really just:

```
case (b1 && b2) of True  -> e1; False -> e2
```

Inlining the definition of `&&` gives:

```
case (case b1 of True -> b2; False -> False) of
True  -> e1
False -> e2
```

Applying the `case` of `case` transformation we get:

```
case b1 of
True  -> case b2    of True  -> e1; False -> e2
False -> case False of True  -> e1; False -> e2
```

The second of the inner `case` expressions is scrutinising a known constructor, and hence can be simplified:

```
case b1 of
True  -> case b2 of True  -> e1; False -> e2
False -> e2
```

Operationally, we can read this expression as: "Evaluate `b1`; if the result is `False` return `e2`; otherwise evaluate `b2` and return `e1` if the result is `True` and `e2` otherwise". The "short-circuiting" of the conditional is now expressed directly.

The above example shows up a problem with the `case` of `case` transformation: `e2` appears twice in the transformed expression. It will be evaluated at most once, since the two occurrences are in different branches of the `case` expression, but there is a danger of code explosion if we are not careful.

Code duplication

Although there is a major risk of code duplication due to the `case` of `case` transformation, there are some particular instances which do not have this problem:

- if the inner `case` has a single alternative;
- if the inner `case` has one non-error alternative. This instance deals with cases where all but one of the branches in the case are *error* branches, that is, they are branches introduced by the compiler to handle pattern matching failures and are semantically equivalent to bottom ($\perp$). In the Haskell code fragment:

  ```
  case e of (a:as) -> eas
  ```

  there will be a pattern match failure if the evaluation of `e` results in an empty list `[]`:

  ```
  case e of (a:as) -> eas
            []     -> error "Error: Pattern Match failure"
  ```

where `error` is a function that will print the error message and abort execution. If we have instances of the `case` of `case` transformation in which the inner `case` only has one non-error branch we have a situation similar to the one we described above, in which we have only a single branch:

```
case (case e of
        (a:as) -> eas
        []     -> error "Error: Pattern Match failure") of
  p1 -> a1
  p2 -> a2
==>
case e of (a:as) -> case eas of p1 -> a1
                                p2 -> a2
          []     -> case error "Error: Pattern Match failure" of
                                p1 -> a1
                                p2 -> a2
==>
case e of (a:as) -> case eas of p1 -> a1
                                p2 -> a2
          []     -> error "Error: Pattern Match failure"
```

where we use the `case` of `error` transformation (Section 3.3.4) in the last step.

Using join points

Recall the result of transforming the boolean short-circuiting example:

```
case b1 of
  True  -> case b2 of True  -> e1; False -> e2
  False -> e2
```

Here `e2` has been duplicated. What does a C compiler do when short-circuiting boolean expressions? It inserts jumps to share the code for `e2`. At first it looks as if this is hard to express in our present universe of discourse. Indeed, in [Aug87] the `case` of `case` transformation is not implemented as a program transformation at all, it is implemented in the code generator so that it can be compiled into a jump. We would like to avoid this.

We cannot eliminate the code generator's involvement altogether, because we need to compile a jump, but we can reduce the complexity of its involvement. All we need to do is bind `e2` to a common variable, `$cont`[6], thus:

```
let $cont = e2
in
case b1 of
  True  -> case b2 of True  -> e1; False -> $cont
  False -> $cont
```

Now, a naïve compiler for a non-strict language would build a *heap-allocated closure* for `$cont`. After all, it might not be evaluated (if `e1` was returned), so it certainly isn't safe to evaluate it before performing the case analysis on `b1`. This is a perfectly correct implementation, but it is rather inefficient compared to compiling a jump. Why can references to `$cont` be compiled into a jump? Because `$cont` is only used in a rather special way, as the *continuation* of one or more branches of the current execution path. So our solution is this:

- Perform a simple analysis to discover which bindings cannot "escape" from the current dynamic environment. Escape analysis is common in Lisp compilers (Orbit, for example [KKR+86]), but it is less successful in a non-strict language, because many more expressions escape. It is rare to find non-escaping continuations in untransformed code written by a programmer.
- Identify them with some sort of annotation (we have used a `$` sign for this purpose).
- Compile a jump (together, perhaps, with some adjustment of the stack pointer) for occurrences of the continuation.

One advantage of this approach is that it allows the decision of whether to duplicate the continuation (in our example, by substituting `e2` for `$cont` throughout) or to share it (by retaining the `let` expression binding `$cont`), to be taken subsequently to, and quite independently from, the `case` of `case` transformation itself. Indeed the question of whether or not to eliminate `let`-bindings by substitution is one which applies to all `let` expressions, not just those binding continuations.

A second advantage to this approach to shared continuations is that it copes with other commonly-occurring situations as well. For example, another situation which is

[6] Here we tag these `lets` ("continuations") with a `$`, but these are "normal" `lets`.

often handled in an *ad hoc* manner is pattern matching failure. Consider the following Haskell function definition:

```
f [] [] = e1
f xs ys = e2
```

The point about this example is that the pattern matching for the first equation can fail to match at two points: on the first empty list and on the second. In either case, `e2` should be returned. In [Pey87] this is solved by extending the language with a special `FAIL` value, which is treated by yet another special case in the code generator. In contrast, here is a translation of `f` into the Core language which avoids inventing special constructs:

```
f xs ys = let $fail = e2
          in case xs of
               []     -> case ys of [] -> e1; (y:ys) -> $fail
               (x:xs) -> $fail
```

Like `$cont`, `$fail` is a variable like any other, but it is detected as a non-escaping continuation, and so can be compiled into a jump. The question of whether to duplicate the continuation or share it is again handled by the general `let` elimination transformation (inlining).

In concluding, we note that there is one further complication in the general case, which has not shown up so far. Consider the following expression:

```
case (case e of True -> e1; False -> e2) of
        []     -> c1
        (x:xs) -> c2
```

The `case` of `case` transformation would duplicate `c1` and `c2`, but now we cannot bind `c2` to a simple variable because it has free variables `x` and `xs`. The solution is to use a lambda abstraction to turn the free variables into arguments:

```
let $cont1 = c1
    $cont2 = \x -> \xs -> c2
in case e of
     True -> case e1 of
               []     -> $cont1
               (x:xs) -> $cont2 x xs
     False -> case e2 of
                []     -> $cont1
                (x:xs) -> $cont2 x xs
```

Effects of the `case` of `case` transformation

In Table 3.5 we see the effects the `case` of `case` transformation has on programs. The first column presents the results with `case` of `case` off, the second one the effect of performing `case` of `case` only if we will not duplicate code (without using join points), and the third column presents our `case` of `case` with join points, which lets us *always* perform the `case` of `case` transformation.

The effect of the `case` of `case` transformation on the number of instructions executed is quite significant, reducing the number of instructions executed on average by 8%, but the use of join points only gives us an extra 1%. The effects on heap usage are mixed, with some programs allocating more heap and others allocating less.

We also expected the `case` of `case` transformation to expose opportunities for many other transformations, specially the `case` reduction transformation. Indeed, the simple version of the transformation increases the number of `case` reductions on average by 35% (sometimes up to 300%!), although the version using join points has no major extra effect (1% more, on average).

The use of join points allowed us to perform on average 10% more `case` of `case` transformations. As we said before, to use join points it is essential that the compiler can indeed optimise these "special" `let`s into jumps. If one does not do that, then join points have actually a negative effect, as we can see in Table 3.6, in which we compare the effect of turning off this "special" compilation of non-escaping `let`s.

We believe that the approach we use for the `case` of `case` transformation is not only more elegant, but generalises the previous descriptions of this transformations by allowing it to be *always* performed *without* code duplication.

3.5.3 `case` floating from `let` right hand side

`case`s may be floated out of strict (demanded) `let`s:

$$\begin{array}{l} \texttt{let}\ v\ \texttt{=}\ \texttt{case}\ e_v\ \texttt{of} \\ \qquad\qquad \{C_i\ v_{i1}\ldots v_{ik}\ \texttt{->}\ e_i\}_{i=1}^{n} \\ \texttt{in}\ e \end{array} \quad\Longrightarrow\quad \begin{array}{l} \texttt{case}\ e_v\ \texttt{of} \\ \ \{C_i\ v_{i1}\ldots v_{ik}\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ e_i\ \texttt{in}\ e\}_{i=1}^{n} \end{array}$$

e is strict in v, $v \notin fv\ e_v$ and $\{v_{i1},\ldots,v_{ik}\} \bigcap fv\ e = \emptyset$

This transformation increases the scope of the `case`, and therefore it might expose transformations, such as `case` reduction, in e. It is also good if e_i is a weak head

case of case Total Instructions Executed			
program	never	without join pts.	with join pts.
queens	1.00	0.53	0.53
mandel2	1.00	0.62	0.62
sched	1.00	0.73	0.63
parstof	1.00	0.83	0.82
sorting	1.00	0.85	0.85
solid	1.00	0.88	0.88
infer	1.00	0.90	0.90
boyer2	1.00	0.91	0.91
fluid	1.00	0.91	0.91
primes	1.00	0.91	0.89
wave4main	1.00	0.91	0.87
gen_regexps	1.00	0.92	0.92
reptile	1.00	0.92	0.92
prolog	1.00	0.93	0.92
cichelli	1.00	0.94	0.92
clausify	1.00	0.94	0.94
compress	1.00	0.94	0.94
event	1.00	0.94	0.94
genfft	1.00	0.94	0.94
hidden	1.00	0.94	0.94
parser	1.00	0.94	0.94
pretty	1.00	0.94	0.94
treejoin	1.00	0.94	0.92
typecheck	1.00	0.94	0.94
wang	1.00	0.94	0.93
gg	1.00	0.95	0.95
hpg	1.00	0.95	0.95
lift	1.00	0.95	0.95
transform	1.00	0.95	0.95
ida	1.00	0.96	0.96
mandel	1.00	0.96	0.95
rewrite	1.00	0.96	0.96
comp_lab_zift	1.00	0.97	0.97
listcompr	1.00	0.97	0.97
listcopy	1.00	0.97	0.97
maillist	1.00	0.97	0.96
veritas	1.00	0.97	0.97
fft	1.00	0.98	0.98
fft2	1.00	0.98	0.98
minimax	1.00	0.98	0.98
multiplier	1.00	0.98	0.98
knights	1.00	0.99	0.99
4 other progs.	1.00	1.00	1.00
Minimum	-	0.53	0.53
Maximum	-	1.00	1.00
Geom. mean	-	0.92	0.91

case of case Total Heap Allocated			
program	never	without join pts.	with join pts.
sorting	1.00	0.77	0.77
queens	1.00	0.84	0.84
sched	1.00	0.86	0.84
parser	1.00	0.88	0.89
gen_regexps	1.00	0.91	0.91
compress	1.00	0.92	0.92
pretty	1.00	0.93	0.93
gg	1.00	0.94	0.94
listcompr	1.00	0.94	0.94
listcopy	1.00	0.94	0.95
reptile	1.00	0.94	0.94
fluid	1.00	0.96	0.96
lift	1.00	0.96	0.96
comp_lab_zift	1.00	0.97	0.98
veritas	1.00	0.97	0.97
prolog	1.00	0.98	0.96
hpg	1.00	0.99	0.99
ida	1.00	0.99	0.99
infer	1.00	0.99	0.99
multiplier	1.00	0.99	0.99
rewrite	1.00	0.99	1.00
cichelli	1.00	1.00	0.93
solid	1.00	1.00	1.03
treejoin	1.00	1.01	1.01
knights	1.00	1.02	1.02
event	1.00	1.03	1.03
genfft	1.00	1.03	1.03
fft	1.00	1.04	1.04
mandel2	1.00	1.04	1.04
typecheck	1.00	1.06	1.06
boyer2	1.00	1.07	1.09
primetest	1.00	1.07	1.07
rsa	1.00	1.07	1.07
transform	1.00	1.09	1.09
parstof	1.00	1.19	1.29
clausify	1.00	1.20	1.20
10 other progs.	1.00	1.00	1.00
Minimum	-	0.77	0.77
Maximum	-	1.20	1.29
Geom. mean	-	0.99	0.99

Table 3.5 case of case: instructions executed and bytes allocated

detect non escaping `lets` Total Instructions Executed		
program	off	on
`primes`	1.00	0.80
`wave4main`	1.00	0.93
`parser`	1.00	0.96
`clausify`	1.00	0.98
`maillist`	1.00	0.98
`mandel2`	1.00	0.98
`boyer2`	1.00	0.99
`fft`	1.00	0.99
`fluid`	1.00	0.99
`hpg`	1.00	0.99
`mandel`	1.00	0.99
`prolog`	1.00	0.99
`reptile`	1.00	0.99
`rewrite`	1.00	0.99
`wang`	1.00	0.99
`treejoin`	1.00	1.05
30 other progs.	1.00	1.00
Minimum	-	0.80
Maximum	-	1.05
Geometric mean	-	0.99

detect non escaping `lets` Total Heap Allocated		
program	off	on
`primes`	1.00	0.50
`wave4main`	1.00	0.55
`parser`	1.00	0.81
`clausify`	1.00	0.82
`treejoin`	1.00	0.82
`maillist`	1.00	0.87
`mandel2`	1.00	0.88
`hpg`	1.00	0.89
`boyer2`	1.00	0.91
`fluid`	1.00	0.91
`parstof`	1.00	0.94
`mandel`	1.00	0.95
`prolog`	1.00	0.95
`event`	1.00	0.97
`fft`	1.00	0.97
`gg`	1.00	0.97
`knights`	1.00	0.97
`reptile`	1.00	0.97
`typecheck`	1.00	0.97
`wang`	1.00	0.97
`comp_lab_zift`	1.00	0.98
`genfft`	1.00	0.98
`multiplier`	1.00	0.98
`rewrite`	1.00	0.98
`compress`	1.00	0.99
`hidden`	1.00	0.99
`listcompr`	1.00	0.99
`listcopy`	1.00	0.99
`primetest`	1.00	0.99
`transform`	1.00	0.99
`veritas`	1.00	0.99
15 other progs.	1.00	1.00
Minimum	-	0.50
Maximum	-	1.00
Geometric mean	-	0.94

Table 3.6 non-escaping `lets`: instructions executed and bytes allocated

normal form expression, since v will no longer be an updatable closure (i.e. a thunk), and therefore no updates will be performed on it.

If the `case` has multiple branches we can still do the transformation, but we would have some code duplication, since e would now occur in each of the branches. This can be avoided using the same technique we used for the `case` of `case` transformation (Section 3.5.2), in which we create a new `let`-binding (a *join point*) for the code that would otherwise be duplicated:

```
let v = case E1 of                let j v = E4
         C1 a b -> E2     ===>   in case E1 of
         C2 a b -> E3                C1 a b -> let v = E2 in j v
in E4                                C2 a b -> let v = E3 in j v
```

This avoids duplicating `E4` in each of the branches. The newly-created `let` can be implemented very efficiently (as discussed in Section 3.5.2) and therefore does not introduce any major efficiency or allocation costs. Although we lose the benefit of increasing the scope of the `case` to include `E4`, we will still benefit in the cases in which `E2` or `E3` are weak head normal form expressions (no updates then).

Even if one is already using the `let` to `case` transformation, which would remove many of the opportunities for this transformation, this transformation is still useful in cases when the `let` to `case` transformation cannot be applied, like when the `let` right hand side has a functional type.

Often both transformations can be used, and we obtain the same result with either of them, as we can see in the following example:

```
let v = case e of
         [] -> e1
         (a:as) -> e2
in e3

(a) ==> let to case + case-of-case
let f v = e3
in case e of [] -> case e1 of v -> f v
             (a:as) -> case e2 of v -> f v

(b) ==> case floating from let
let f v = e3
in case e of [] -> let v = e1 in f v
             (a:as) -> let v = e2 in f v
```

As we do not (and should not) change the strictness information on `v`, we can get (b) to be further transformed to (a).

But priority should be given to the `let` to `case` transformation, for a very subtle reason: if v is of a single constructor type (e.g. a pair) we will use the *unboxing* `let` to `case` transformation, leading us to the following sequence:

```
(c) ==> unboxing let to case
case (case e of [] -> e1; (a:as) -> e2) of
  (x,y) -> let v = (x,y) in e3
==> case-of-case
let f x y = let v = (x,y) in e3   -- e3 "knows" shape of v
in case e of
     [] -> case e1 of
             (x,y) -> f x y
     (a:as) -> case e2 of
                 (x,y) -> f x y
```

the reason for this is that for a `case` of `case` we always abstract the join point with respect to the outer case alternatives' binders. In (a) this was `v`, but in (c) we have `x` and `y` as free variables. The advantage of (c) is that `e3` may be further simplified, e.g. if it scrutinises `v` (which may well be the case, since `v` is strict in `e3`).

Cheap eagerness

There is an interesting optimisation that uses the `case` floating from `let` transformation, but without the restriction on the `let` being strict. But how can we keep the same semantic meaning after the transformation if the `let` is not strict? First let us see why the `let` must be strict, and then see in which circumstances the restriction can be relaxed.

The restriction is needed to avoid problems like the following:

```
let x = case y of
          (a,b) -> e1
in e2
```

If `x` does not get evaluated in `e2`, then `y` will not be evaluated either. If we float the `case` out of the `let` then `y` will get evaluated even if `x` is not. Also, if the evaluation of `y` fails or diverges (i.e. it is $\perp$), the program will also fail or diverge if the transformation is applied. Therefore the two problems of doing the transformation on lazy `let`s are:

- unbounded extra evaluation may occur;
- the program may fail or diverge when it did not before, therefore we will be changing the semantics of the program.

We cannot change the semantics of the program, therefore if we are going to do this transformation for lazy `let`s we will have to guarantee that the expression the `case` is scrutinising cannot fail.

The cost of the extra evaluation is another problem. Actually if the cost is small enough we might be willing to pay it, as the expression could end up being evaluated anyway and we are also benefiting from increasing the scope of the `case` expression by exposing transformations. Therefore we actually do this transformation in some very specific cases: for cheap non-failing `case`s. These are `case`s scrutinising some primitive operations on unboxed values, like primitive `Int` addition, subtraction, multiplication, and similar operations for `Float`s and `Double`s.

In this case we are doing an optimisation called cheap eagerness [Myc81, Aug87], in which we perform some (possibly unnecessary) small amount of work to take advantage of exposing other optimisations. This is another transformation that is often implemented in the code generator of compilers, and not presented as a source-to-source transformation.

The following is an example of the transformation:

```
let v = \ a# -> let w = case a# +# 1# of
                          r# -> MkInt r#
                in f w
in ...
==>
let v = \ a# -> case a# +# 1# of
                   r# -> let w = MkInt r#
                         in f w
in ...
```

The cost of creating a closure for `w` and possibly updating it is certainly greater than that of evaluating `a# +# 1#`. We may also be exposing other transformations, as `w` is now directly bound to a constructor.

A more aggressive version of this transformation could be used if we had a "cheapness analysis", that could select other (possibly bigger) cheap non-failing expressions to be eagerly evaluated.

Other `case` floating transformations

Other possible `case` floating transformations are:

- Floating a `case` from a `let` body. This is precisely of the transformation of pushing a `let` into `case` branches which we discuss in Section 5.1. These are just different ways of looking at the same transformation, either as pushing the `let` into the `case` branch or floating the `case` out of the `let` body.
- Floating a `case` from `case` alternatives. This is similar to swapping the order of `let`s, which does not achieve much, and the same is true for swapping an inner `case` (in a branch) with an outer one. This would only be possible for `case`s with a single branch, otherwise it would not be correct.
- Floating `case`s out of lambdas. This achieves a similar effect to full laziness (Section 5.2), by allowing the possibility of sharing the evaluation of the scrutinee. We discuss this transformation in Section 5.3.

3.6 Strictness based transformations

Some local transformations rely on strictness information. Strictness analysis [Myc81] is an analysis widely used in lazy functional languages that can give information on whether a function argument is guaranteed to be evaluated in the function body or not. If it is known that it is going to be evaluated one can safely transform call-by-need to call-by-value (i.e. evaluate the arguments before the call), which can be implemented more efficiently.

The same analysis can be used to identify which `let`-bindings are sure to be evaluated (demanded) by its body. These `let`s can then be transformed to be evaluated earlier with no change in the semantics of the expression.

The transformations we describe in this section are also described in [PP93], together with other transformations based on strictness information (e.g. the worker-wrapper transformation). In [PP93] experimental results are also presented, therefore we will not present results on the effectiveness of these transformations in particular.

3.6.1 `let` to `case`

The `let` to `case` transformation can be done whenever we have a strict `let` (i.e. one whose bound variable is guaranteed to be demanded during the evaluation of its body) whose right hand side is not already in weak head normal form:

`let` v `=` e_v `in` e $\Longrightarrow$ `case` e_v `of` v `->` e
if v is of a constructor type, e is strict in v and e_v is not in weak head normal form

In the original expression we are allocating a closure for v in the heap which only later will be evaluated (as it is strict) and possibly updated (if v's closure was updatable). After the transformation we evaluate e_v first and bind it to v, therefore saving the cost of the update and some heap allocation if the update was not done originally in place. Even if the closure was not updatable we would avoid allocating a closure that would be later entered, by evaluating it in advance.

If e_v is a weak head normal form we also do not perform this transformation, as there is no evaluation to be done in e_v. We would in this case prefer the `let`-bound form, and we actually do the opposite transformation (Section 3.3.1).

In our compiler we introduce an extra restriction for doing this transformation: the type of v must *not* be a function type or a type variable (which can be instantiated to a function type). This restriction is due to implementation details of the STG machine, as `case`s cannot scrutinise objects which have a function type.

3.6.2 Unboxing `let` to `case`

The unboxing `let` to `case` transformation is similar to the previous one, but it has the advantage of exposing the structure of the expression, by explicitly exposing its constructor. To avoid code duplication this is only used when the type of the `let`-binding is a single constructor data type, like n-tuples, boxed integers, etc.

`let` v `=` e_v	$\Longrightarrow$	`case` e_v `of`
`in` e		$C_k\ v_{k1} \ldots v_{kl}$ `-> let` v `=` $C_k\ v_{k1} \ldots v_{kl}$ `in` e
if v is of a single constructor type, e is strict in v and e_v is not in weak head normal form		

The extra advantage here compared to the previous transformation is that, since the structure is exposed, transformations like the `case` reduction may be exposed. Also, often the `let`-binding introduced by the transformation is eliminated later, as in the following example:

```
    let v = f a
    in ... case v of
             (x,y) -> e
==>
    case f a of
      (x',y') -> let v = (x',y')
                 in ... case v of
                          (x,y) -> e
==>
    case f a of
      (x',y') -> ... e[x'/x,y'/y]
```

3.7 Other transformations

3.7.1 Constant folding

We do constant folding exclusively on primitive operations on basic literals. This means that we do the following transformation:

`3# +# 5#` $\Longrightarrow$ `8#`

but no simplification is done for (overloaded) expressions:

```
(Num.+) dict 3 5
```

This is correct as it is possible to define an instance of `Num.+` in which the result of the above expression is not `8`.

Some problems arise from doing constant folding in a later phase of the compiler, as some expressions that could be simplified are not easy to spot. The expression `(a + 1 + 2)` (of type `Int`), for example, would be easily spotted if we did the second addition first, since it would be translated to:

```
case 1# +# 2# of               case 3# of
  r# -> case a# +# r# of  ==>    r# -> case a# +# r# of
          s# -> ...                      s# -> ...
```

(which could be further simplified). But doing the first addition first gives us:

```
case a# +# 1# of
  r# -> case r# +# 2# of
          s# -> ...
```

in which it is not so obvious that we could simplify the `(1 + 2)`. Unfortunately it is not easy to spot and use the associativity of `+#` at this level, and also the associativity of `+#` may not actually hold (e.g. `(maxInt + (1 - 1))` may differ from different from `((maxInt + 1) - 1)`[7], if the machine checks for `Int` overflow). We therefore do not try to exploit associativity or commutativity to increase opportunities for constant folding.

We do constant folding for many of the basic predefined operations on `Int`s, `Char`s, `Float`s, `Double`s and `Bool`s:

- negation, addition, subtraction, multiplication, remainder and division on `Int`s, `Float`s and `Double`s;
- type conversion functions between `Int`s, `Char`s, `Float`s and `Double`s;
- comparison operators on `Int`s, `Char`s, `Float`s and `Double`s;

One should check for overflows and/or invalid operations when constant folding. Although we do check for division by zero, we currently do not check, for example, that the addition of two `Int`s will be greater than the `maxInt` defined by Haskell. Since the compiler represents `Int`s internally as infinite precision `Integer`s it would be easy to check if the result of an operation is above a given `maxInt`.

We also sometimes transform an expression into a similar one, which has roughly the same cost, but exposes possibilities for transformations to occur. An example of this was presented in Section 3.3.3 on `case` merging, where we transform

```
                         case v of
eqInt v k      ==>         k -> True
                           _ -> False
```

where `v` is a variable and `k` is an explicit constant (e.g. `1, 2` etc.).

[7] assuming we inline the constant `maxInt`.

3.7.2 Eta expansion

We perform general η-expansion when we have an expression with a functional type that has arity greater than the number of lambdas enclosing it:

```
v = \ a b -> f a b      ===> v = \ a b c -> f a b c
```

(assuming `f` has arity 3). This improves the efficiency because instead of creating a partial application of `f` when `v` is entered, (if it is being called with 3 arguments) `f` will be called directly. This also saves an argument satisfaction check (to check if enough arguments are already available) in some implementations.

The notion of arity in this case is a bit different from the usual notion, as we do not intend to lose laziness by adding extra arguments to a function. We do not, for example, perform the following transformation:

```
v = \a b -> let x = e  =/=> v = \a b c -> let x = e
            in f x b                        in f x b c
```

Although `v` can receive 3 arguments (we assume that `f` receives 3 arguments), if it is partially applied to two arguments, we would have a very different behaviour for the two expressions:

- in the first one a closure for `x` is allocated and would be shared by the partial application (if the partial application was applied many times), while
- in the second one, as it only does any work after receiving the 3 arguments, the closure would be allocated and evaluated as many times as the partial application was applied, thus losing laziness.

Therefore the concept of arity we use is not directly related to the maximum number of arguments that a function may receive, but to the number of lambdas in its definition, i.e. the number of arguments that can be passed to the function before it performs any actual "work", like evaluate a `case` or a `let` expression.

`case` η-expansion

Actually we sometimes do η-expansion when we have a `case` expression. Let us analyse this case in more detail. Assuming $e_1 \ldots e_n$ have a functional type:

$$\begin{array}{l} \texttt{case}\ e\ \texttt{of} \\ \quad p_1\ \texttt{->}\ e_1 \\ \quad \ldots \\ \quad p_n\ \texttt{->}\ e_n \end{array} \Longrightarrow \begin{array}{l} \lambda y.\texttt{case}\ e\ \texttt{of} \\ \quad p_1\ \texttt{->}\ e_1\ y \\ \quad \ldots \\ \quad p_n\ \texttt{->}\ e_n\ y \end{array}$$

- It is a bad idea to do this if e is not a simple variable, because it pushes a redex e inside a lambda. Even if e is a variable, doing this transformation moves an evaluation inside a lambda, which loses a small amount of work for each call of the lambda.
- If any of the e_i are redexes, it would also probably be a bad idea, for the same reason.

But if the two problems above do not occur, in particular if the scrutinee is a variable and therefore the (possible) work duplication is basically restricted to entering the variable, it is sometimes a very useful transformation, e.g.:

```
putChar (MkChar c#) = putC c#   `thenIO_`
                      returnIO ()
```

`ThenIO_` is then inlined, giving:

```
putChar = \ c -> case c of
                   MkChar c# -> \ s -> ...
```

The `thenIO_` (which has arity 3) exposed an explicit lambda, but even if not, it would be better to make a saturated call to `thenIO_` than (the existing) unsaturated one. Therefore we would prefer to have the function in the form:

```
putChar = \ c -> \ s -> case c of
                          MkChar c# -> ...
```

although we may be reentering the closure for `c` multiple times (if `putChar` is partially applied).

So, the strategy is to do it if:

- the right hand sides have functional type;
- e is a variable;
- all the right hand sides are manifestly weak head normal forms.

Effects of η-expansion

In Table 3.7 we can see the effect of η-expansion on our benchmark programs. The effects are clearly positive, with an average improvement of 5% on the total of instructions executed, and of 6% on the total heap allocated.

3.8 The Transformations interacting

In this section we will follow a few examples of how big effects can be achieved by using the transformations we described in the previous sections. Many of these motivating examples have shown up in real application programs. The effects usually involve a combination of many of the transformations and therefore give an idea of how the transformations interact with each other to improve the code generated. Some interesting examples of the transformations interacting have already been presented in the previous sections, such as the use of `case` of `case` and `case` reduction transformations to achieve the effect of short circuiting boolean expressions.

3.8.1 Repeated evaluations

The expression `x+x` (where `x` is of type `Int`) in the source code generates the following code in the compiler:

```
case x of
  MkInt x# -> case x of
                 MkInt y# -> case x# +# y# of r# -> MkInt r#
```

due to the inlining of the (boxed) operator `+`, which unboxes its two arguments, applies the primitive (unboxed) operator `+#` to them and finally boxes the resulting value. In this case it unboxes `x` twice, but the `case` reduction transformation can eliminate the second evaluation of `x` and generate the code we expect:

```
case x of MkInt x# -> case x# +# x# of r# -> MkInt r#
```

The transformations are using unboxed data types, as presented in [PL91a].

η-expansion Total Instructions Executed		
program	off	on
prolog	1.00	0.76
parser	1.00	0.77
gen_regexps	1.00	0.79
pretty	1.00	0.79
listcompr	1.00	0.82
listcopy	1.00	0.83
reptile	1.00	0.83
maillist	1.00	0.86
treejoin	1.00	0.86
rewrite	1.00	0.87
sorting	1.00	0.87
fft	1.00	0.89
knights	1.00	0.89
lift	1.00	0.89
mandel	1.00	0.89
typecheck	1.00	0.89
veritas	1.00	0.89
gg	1.00	0.91
hpg	1.00	0.92
multiplier	1.00	0.92
minimax	1.00	0.94
mandel2	1.00	0.95
fluid	1.00	0.96
parstof	1.00	0.96
genfft	1.00	0.98
boyer	1.00	0.99
compress	1.00	0.99
fft2	1.00	0.99
hidden	1.00	0.99
infer	1.00	0.99
solid	1.00	0.99
wave4main	1.00	1.03
14 other progs.	1.00	1.00
Minimum	-	0.76
Maximum	-	1.03
Geometric mean	-	0.93

η-expansion Total Heap Allocated		
program	off	on
treejoin	1.00	0.70
gen_regexps	1.00	0.75
pretty	1.00	0.77
sorting	1.00	0.77
maillist	1.00	0.78
listcompr	1.00	0.80
reptile	1.00	0.80
listcopy	1.00	0.82
parser	1.00	0.82
lift	1.00	0.85
prolog	1.00	0.86
veritas	1.00	0.88
hpg	1.00	0.89
gg	1.00	0.91
mandel2	1.00	0.93
typecheck	1.00	0.93
fluid	1.00	0.96
multiplier	1.00	0.96
knights	1.00	0.97
rewrite	1.00	0.97
compress	1.00	0.98
fft	1.00	0.98
rsa	1.00	0.98
boyer	1.00	0.99
mandel	1.00	0.99
minimax	1.00	0.99
boyer2	1.00	1.02
wave4main	1.00	1.03
parstof	1.00	1.07
17 other progs.	1.00	1.00
Minimum	-	0.70
Maximum	-	1.07
Geometric mean	-	0.93

Table 3.7 η-expansion: instructions executed and bytes allocated

3.8.2 Lazy pattern matching

Lazy pattern matching is very inefficient. Consider:

```
let (x,y) = E in B
```

This desugars to:

```
let t = E
    x = case t of (x,y) -> x
    y = case t of (x,y) -> y
in B
```

It allocates three thunks (updatable closures)! However, if `B` is strict in *either* `x` *or* `y`, then the strictness analyser will easily spot that the binding for `t` is strict, so we can do an unboxing `let` to `case` transformation:

```
case E of (x,y) -> let t = (x,y) in
                   let x = case t of (x,y) -> x
                       y = case t of (x,y) -> y
                   in B
```

whereupon the `case` reduction transformation eliminates the `case` expressions in the right hand side of `x` and `y`, and `t` is then spotted as being dead code, and we get

```
case E of (x,y) -> B
```

which is much more efficient than the original version.

3.8.3 Error tests eliminated

The elimination of redundant alternatives, and then of redundant `case`s, arises when we inline functions which do error checking. A typical example is this:

```
if (x `rem` y) == 0 then (x `div` y) else y
```

Here, both `rem` and `div` do an error-check for `y` being zero. The second check is eliminated by the transformations. After transformation the code becomes:

```
case y# of 0# -> error "rem: zero divisor"
           _  -> case x# rem# y# of
                   0# -> case x# div# y# of r# -> MkInt r#
                   _  -> y
```

3.8.4 Compiling the factorial program

In this section we show how the transformations interact when generating a more efficient version for the factorial program.

A definition of the factorial function in the Core language is:

```
fact :: Int -> Int
fact = \ n -> case (n < (MkInt 1#)) of
                True -> MkInt 1#
                False -> n * fact (n - (MkInt 1#))
```

We initially inline the definition of `-`, `*` and `<` to make explicit the unboxing/boxing operations on its arguments/results. These inlinings lead us to many cases where we are unboxing a value that has previously been unboxed or that has just been boxed, which are redundant operations.

The first time the simplifier is applied it transforms the code by:

- inlining basic operations;
- applying β-reductions where appropriate;
- avoiding redundant boxing/unboxing of values;
- doing `case` of `case` transformations where appropriate.

By doing this the code is transformed to

```
fact = \ n -> case n of
                MkInt n# ->
                    case (n# <# 1#) of
                      0# -> case (fact (case (n# -# 1#) of
                                              v# -> MkInt v#)) of
                              MkInt v'# -> case (n# *# v'#) of
                                                 v''# -> MkInt v''#
                       _ ->  MkInt 1#
```

We are already avoiding many unnecessary boxing/unboxing operations, which is an improvement by itself. But it can do an even better job if we use a strictness analyser together with the worker/wrapper transformation [PP93], which will split the function into a worker/wrapper pair of functions. The transformation tries to split functions with strict arguments into two functions:

- the *wrapper* function that unboxes the strict arguments (when they have single constructor data types), and then calls
- the *worker* function, which is the same original function, but which receives the strict arguments already unboxed.

Here we can see the code for `fact` after the transformation, which has split it into `fact` (the wrapper) and `fact.wrk` (the worker):

```
fact :: Int -> Int
fact = \n -> case n of
               MkInt n# -> fact.wrk n#
fact.wrk :: Int# -> Int
fact.wrk = \ n# -> let n = MkInt n#   -- could be needed in the body
                   in case (n# <# 1#) of
                        0# -> case (fact (case (n# -# 1#) of
                                              v# -> MkInt v#)) of
                                MkInt v'# -> case (n# *# v'#) of
                                                v''# -> MkInt v''#
                        _ -> MkInt 1#)
```

The idea of the worker/wrapper transformation as done in the Glasgow Haskell Compiler is to make minimal changes from the original functions while splitting, and let the simplifier do the rest of the job. Therefore we get an inefficient worker/wrapper pair which will become a more efficient one through the transformations. Now the simplifier is called again to inline the wrapper (`fact`) into the worker (`fact.wrk`), to get the worker to call itself. By doing this we get more opportunities for removing extra boxing/unboxing operations, `case` of `case` transformations, β-reductions, etc.

```
fact.wrk = \ n# -> case (n# <# 1#) of
                    0# -> case (n# -# 1#) of
                           v# -> case fact.wrk v# of
                                  MkInt v'# -> case (n# *# v'#) of
                                                v''# -> MkInt v''#
                    _ -> MkInt 1#
```

This definition is a huge improvement on the initial one, by keeping the values unboxed during most of the computation.

3.9 Confluence and termination

Our set of transformations can be seen as a set of term rewriting rules. We would like the set of transformations we use to be:

- correct: that is, the transformed code always has the same semantics as the original code. We prove the correctness of some transformations in Chapter 9.
- efficiency improving: that is, the transformed code costs less to execute than the original. We return to this topic in Chapter 9.

In addition it would be a considerable practical advantage if the set of transformations was:

- confluent: that is, we can apply the transformations in any order (when more than one is applicable) and we still get the same result. This is important to make sure that we are not losing transformations or generating worse code by choosing to apply one transformation before another one, when both are applicable.
- terminating: that is, the process of simplification terminates, meaning that we always get to a point where no transformation is applicable. One has to be particularly careful that one transformation cannot generate code that can be transformed back to the original code by other transformations, i.e. that no transformations undo the work of other transformations.

Since the transformations are in a very simple left to right form with very few side conditions they are good candidates to be treated as rewrite rules in a term rewriting system. In [Mat94] a proof of confluence and termination of a *subset* of the rules was obtained, using the order-sorted equational theorem proving system MERILL [Mat93], developed at Glasgow University. Initially the system was used to prove confluence and termination for the *subset* of the rules containing the `let` and `case` floating rules. Later the set was extended to include the constructor reuse, beta reduction and inlining, retaining the same properties.

The *full* set of transformations is clearly non-confluent, as actually there are instances in which we have to make a choice between rules that can be applied at a given point that *do* result in different code, and therefore are *not* confluent (e.g. `let` to `case` vs. `case` float from `let`, in Section 3.5.3).

3.10 Conclusions

We have presented the complete set of local transformations performed by the simplifier pass of the Glasgow Haskell Compiler.

This set of transformations, together with the overall design of the simplifier and the Core language, allows complex transformations to be performed by composing simple transformations.

The combined effect of the transformations is discussed in the next chapter.

Chapter 4

Local Transformations: Implementation and Results

In this chapter we present details on the implementation of the transformations presented in the previous chapter (Section 4.1) and their effect on real programs (Section 4.2).

4.1 Implementation

The transformations presented in the previous chapter are implemented in the *simplifier* pass of the compiler, which consists of the following (sub-) passes:

1. Analyse: perform occurrence analysis and dependency analysis.
2. Simplify: apply as many transformations as possible.
3. Iterate: repeat steps 1 and 2 above until no further transformations take place (or optionally when a predefined maximum number of iterations is reached).

The occurrence analyser collects information about binders' occurrences, in particular the number of occurrences and their location:

- whether it occurs inside a lambda abstraction or not;
- how many times it occurs;
- whether it occurs as an argument to a function or a constructor.

This information is used for inlining decisions, which are discussed in Chapter 6. This is "global" information, therefore it could not be gathered while the simplifier pass is being run.

Dependency analysis is needed because, while floating `let`s out of `let`s (Section 3.4.2), we may leave recursive bindings that are not necessarily recursive. Knowing precisely which `let`s are recursive is useful for some transformations and lets us generate more efficient code. Since all the information needed for dependency analysis is already gathered by the occurrence analysis, we do them together.

In step 2 we apply as many transformations as possible in one traversal of the input program. To see the importance of performing as many transformations as possible in one pass, consider a sequence of transformations in which each transformation enables the next. If each iteration of step 2 only performed one transformation, then the entire program would have to be re-analysed by step 1, and re-traversed by step 2, for each transformation. Sometimes multiple iterations are unavoidable, but it is often possible to do a sequence of transformations in a single pass.

The compiler repeats steps 1 and 2 until a fixed point is reached or (optionally) until a supplied maximum number of iterations is reached. To reduce the number of iterations the algorithm recursively simplifies components of the language constructs (subexpressions) and then checks if any of the transformations for that constructor can be applied, as we will see in Section 4.1.2. For all the benchmark programs the simplifier never has to iterate more than 4 times, typically needing only 2 iterations (i.e. 2 traversals of the code, where the second one did not perform any simplification) to reach a point in which no transformations can be applied.

The compiler applies the simplifier both before and after each of the global transformations. Simplifying before a global transformation makes the global transformation more effective, and simplifying after a global transformation allows the simplifier to take advantage of the changes made by the global transformation.

4.1.1 Renaming

Every program-transformation system has to worry about name capture. For example, here is an erroneous transformation:

```
let y = E                               =/=>  let y = E
in (\x -> \y -> x + y) (y+3)                  in (\y -> (y+3) + y)
```

The transformation fails because the originally free-occurrence of `y` in the argument `y+3` has been "captured" by the `\y`-abstraction.

There are various sophisticated solutions to this problem but we adopted a very simple one: we uniquely rename every locally-bound identifier on every pass of the simplifier. Since we are producing an entirely new program anyway (rather than side-effecting an existing one), it costs very little extra to rename the identifiers as we go.

So our example would become:

```
let y = E                          ===>  let y1 = E
in (\x -> \y -> x + y) (y+3)             in (\y2 -> (y1+3) + y2)
```

The simplifier accepts as input a program which has arbitraryly bound variable names, including "shadowing" (where a binding hides an outer binding for the same identifier), but it produces a program in which every bound identifier has a distinct name.

This is also useful for other passes of the compiler, but is also essential to keep the simplification process as simple as possible, as one does not have to worry about name clash problems.

Of course the simplifier could be implemented without renaming, but this would introduce extra work to avoid name clashes. Even so, renaming would still be needed in some circumstances (e.g. when performing β-reduction).

4.1.2 The simplifier function

The key function used to simplify expressions has the following type:

```
simplExpr :: SimplEnv -> InExpr -> [OutArg] -> SmplM OutExpr
```

This type signature can be understood as:

- The environment, of type `SimplEnv`, provides two kinds of information:
  - a mapping from old identifiers to new identifiers, used for renaming;
  - information about what is bound to an identifier in the enclosing context, e.g. that a variable is bound to a constructor or information about its right hand side that is used for inlining decisions (Chapter 6).
- The second and third arguments together specify the expression to be simplified (an expression and a list of its arguments, if it was being applied to any).

- The result is the simplified expression, wrapped up by the `SmplM` monad. The monad `SmplM` has only two purposes:
  - It plumbs around a supply of unique names, so that the simplifier can easily invent new names for binders when renaming.
  - It gathers together counts of how many of each kind of transformation have been applied, for statistical purposes. These counts are also used in step 3 to decide when the simplification process has reached a fix point.

The simplifier's invariant is this:

$$\texttt{simplExpr}\ env\ expr\ [a_1, \ldots, a_n] = expr[env]\ a_1\ \ldots\ a_n$$

That is, the expression returned by `simplExpr` $env\ expr\ [a_1, \ldots, a_n]$ is semantically equal to (although hopefully more efficient than) $expr$, with the renamings in env applied to it, applied to the arguments $a_1, \ldots, a_n$.

The arguments are carried "inwards" by `simplExpr`, as an accumulating parameter. This is a convenient way of implementing the transformations which float `let`s and `case`s out of applications.

The order in which each of the language constructs is simplified is:

- $e\ v$ (applications): e is simplified with v in its argument list. If e (after simplification) turns out to be a lambda expression we can apply beta reduction. If it turns out to be a `let` or a `case` we can float the `let` or the `case` out of the application, and then simplify again still with v in the argument list.
- $\lambda v.e$ (lambda expression): The body e is simplified.
- `let` v `=` e_v `in` e (let expression): the right hand side of the `let` is simplified first, since e_v may turn out to be a `let` or a `case`, exposing "floating from `let`" transformations. If the e_v turns out to be a constructor we record that information in the environment. One may also apply the `let` to `case` transformation, if this is a strict `let`. Finally the body is simplified using an environment possibly augmented with information about the `let` right hand side.
- `case` e `of` *alts* (case expressions): The expression e is simplified, possibly exposing "floatings from `case` scrutinee" transformations. These may expose a variable or a constructor in the `case` scrutinee, leading to the `case` reduction transformation, for example. After that the `case` alternatives are simplified.

4.2 Results

To verify the effectiveness of our set of transformations, we performed a series of experiments with the transformations enabled/disabled. All measurements use the `nofib` benchmark suite as described in Chapter 2.

We will first discuss how often each transformation occurs during the compilation of the `nofib` programs, then what effect the simplifier has in the overall performance of the programs.

4.2.1 How often is each transformation used?

Other passes in the Glasgow Haskell Compiler are aware of the existence of the simplifier, and therefore sometimes produce inefficient code, knowing that the code will be improved by the simplifier. Therefore it would be unfair to compare directly a *simplified* program with one that had no simplification at all. To minimise this effect and still give an idea of the overall benefit of the simplifier, we present 5 sets of results in this section:

(a) Completely unsimplified program. As this leaves even trivial bindings in the code (e.g. `lets` binding variables to variables), we also present a "minimally simplified" version, which is the next set.

(b) "minimal simplification", consisting of a single non-iterative run of the simplifier, which has most of the transformations turned off, except `let` and `case` floating from application, beta reduction and inlining of trivial right hand sides, e.g. variables and literals.

(c) a full run of the simplifier (up to 4 iterations, although this limit was never reached). This excludes the effects of strictness analysis information, and therefore excludes the strictness-based transformations.

(d) a full run of the simplifier, followed by strictness analysis (which includes worker-wrapper transformation, see [PP93]), followed by a second full run of the simplifier.

(e) a fully optimised run of the compiler. This includes all the optimisations in the compiler, including ones we describe in other chapters of this thesis.

In Tables 4.1 and 4.2 we present a raw count of the number of times each transformation is applied in each of the programs in the `nofib` benchmark suite. Whenever a transformation is disabled or has not occurred at all during a particular run we have omitted that column. The second column presents the number of tokens (counted by the lexer) for each of the programs, giving a rough idea of their size.

We have not measured a few transformations, either because their numbers would not be very meaningful (e.g. many of its instances are explicitly created by other transformations) or for purely practical reasons. They are:

- dead code elimination;
- dead alternative elimination;
- default binding elimination;
- constant folding.

We also do not measure the effect of cheap eagerness, which is mixed with the other forms of `case` floating from `let`s.

There are many opportunities for transformations such as `case` reduction, although they rarely occur explicitly in the source code. This is true for many of the transformations, i.e., they are generated by the compilation process after desugaring and inlining of expressions take place. Also, due to the way the transformations interact, if one transformation is turned off the numbers for the other transformations will also be affected.

Only 11 programs perform 4 iterations of the simplifier (where the fourth iteration did not perform any transformation), showing that the system (for these benchmark programs) always reaches a fixed point after at most 3 iterations. This is the maximum number of iterations for all runs of the simplifier (6 runs in the fully optimised version).

4.2.2 Overall effect of the transformations

In Table 4.1 we can see the overall effect of the transformations on instructions executed and heap allocation for the benchmark programs.

Since the compiler relies on the use of some of the transformations during the process of desugaring, we decided to present the results in relation to (b).

program	tokens	β-reduction				Inlinings			constructor reuse			case reduction				case elim.		case merge		case of error			max. iter.			
		3.1				3.2.2			3.2.3			3.3.1				3.3.2		3.3.3		3.3.4						
		b	c	d	e	c	d	e	c	d	e	b	c	d	e	d	e	d	e	c	d	e	b	c	d	e
boyer	2549	0	2	3	7	6	7	18	53	53	0	1	2	6	6	0	0	0	0	0	0	0	2	3	3	2
boyer2	2005	1	4	45	72	11	52	113	22	22	10	0	0	61	84	1	3	0	3	0	3	3	2	3	3	3
cichelli	1379	3	7	43	103	6	53	152	4	4	2	1	1	27	51	0	0	0	0	0	0	0	2	3	3	3
clausify	988	1	2	14	44	1	15	52	5	5	4	0	0	12	17	0	1	0	0	0	0	21	2	2	3	4
comp_lab_z	9980	1	63	177	185	71	205	216	19	19	11	2	7	108	140	0	5	0	0	1	1	1	2	3	4	4
compress	1317	1	4	15	227	6	21	245	17	20	7	0	0	4	25	0	1	0	0	0	0	0	2	2	2	3
event	5576	1	13	57	58	16	63	68	11	11	6	2	0	41	64	0	0	0	0	0	0	0	2	2	3	3
exp3_8	130	1	5	15	17	4	15	22	1	1	0	0	13	27	24	0	0	0	0	0	0	0	2	2	2	2
fft	5860	1	20	142	132	22	182	182	7	9	11	0	1	89	144	0	0	0	0	0	0	0	2	2	4	3
fft2	1092	10	14	59	100	14	66	125	5	5	1	0	0	22	52	0	0	0	0	0	0	0	2	2	3	3
fluid	12495	13	191	726	969	234	941	1295	42	43	73	3	12	397	921	2	13	0	0	0	1	2	2	3	3	4
gen_regexp	280	0	1	20	30	1	27	39	2	2	1	0	2	20	28	0	0	0	0	0	2	2	2	2	2	2
genfft	7455	4	21	103	96	19	121	116	17	17	5	0	0	52	87	0	0	0	0	0	0	0	2	2	2	2
gg	6769	14	38	409	664	33	466	842	73	77	27	5	8	296	440	0	5	0	1	1	2	3	2	3	3	4
hidden	3159	2	29	255	378	35	348	541	8	9	23	0	7	170	313	0	0	0	0	0	0	0	2	3	3	3
hpg	6345	6	13	277	516	7	295	733	23	25	13	1	2	132	239	0	5	0	0	0	1	0	2	3	3	3
ida	5876	1	59	132	136	67	161	171	9	10	6	1	7	86	126	1	2	0	0	0	0	0	2	3	4	4
infer	4624	0	33	79	182	45	111	312	10	10	2	2	2	43	57	0	0	0	0	0	0	0	2	3	3	3
knights	2784	5	15	112	181	10	124	257	20	20	11	2	2	92	214	0	0	0	0	0	0	6	2	2	3	3
lift	4094	3	12	76	167	15	91	210	251	264	19	5	5	60	97	0	4	0	0	0	1	0	2	3	3	4
listcompr	6992	1	7	29	30	7	38	34	146	146	3	3	1	15	37	2	0	0	0	0	0	0	2	3	3	2
listcopy	7076	1	7	29	30	7	38	34	146	146	3	3	1	15	37	2	0	0	0	0	0	0	2	3	3	2
maillist	545	1	1	14	50	0	15	82	3	3	2	0	0	8	27	0	1	0	1	0	0	0	2	2	2	2
mandel	682	1	2	38	81	4	55	125	0	0	0	0	0	8	24	0	0	0	0	0	0	0	2	2	2	3
mandel2	969	0	4	103	106	6	132	138	5	5	5	0	4	129	202	0	0	0	0	0	0	0	2	2	2	4
minimax	1745	0	0	33	53	0	34	73	89	89	4	4	4	48	46	0	1	0	0	0	0	0	2	3	3	2
multiplier	2803	2	11	103	144	9	112	174	11	11	2	12	12	122	183	0	7	0	0	0	0	0	2	3	3	3
parser	5867	3	8	289	474	5	296	694	59	59	16	2	2	335	310	0	2	0	6	0	0	3	2	3	3	3
parstof	15548	69	129	331	340	83	421	421	140	142	4	0	3	169	263	2	2	0	0	0	0	0	2	3	3	3
pretty	1384	0	13	70	87	25	94	117	15	17	6	0	10	35	43	2	2	0	0	0	0	0	2	3	3	3
primes	84	0	0	6	6	0	7	7	0	0	0	0	0	5	6	0	0	0	0	0	0	0	2	2	2	2
primetest	1076	5	9	114	165	4	135	201	2	2	2	0	4	109	190	0	1	0	0	0	1	1	2	3	3	4
prolog	2812	6	10	59	123	7	61	194	15	19	10	2	2	43	74	1	2	0	1	0	0	0	2	3	3	3
queens	123	0	0	10	13	0	12	21	1	1	1	0	0	7	15	0	0	0	0	0	0	0	2	2	2	2
reptile	7818	6	60	426	483	57	548	662	57	57	28	1	12	338	580	6	5	0	3	0	0	17	2	3	4	4
rewrite	4495	4	34	100	153	56	130	237	35	38	26	1	7	94	128	2	3	0	0	0	0	0	2	3	3	3
rsa	500	2	5	72	92	3	83	100	2	4	5	2	2	39	66	0	0	0	0	0	0	0	2	2	3	3
sched	6672	1	32	70	91	36	79	100	47	47	5	0	15	60	138	1	3	0	1	0	0	4	2	3	3	4
solid	14430	2	71	200	188	97	279	271	51	51	14	0	33	158	220	0	2	0	0	0	0	0	2	3	3	3
sorting	1348	14	19	23	44	7	11	61	21	21	22	17	17	23	25	0	0	0	0	0	0	1	2	3	3	3
transform	15187	1	66	195	233	140	312	360	132	132	28	3	11	108	182	0	7	0	0	0	0	7	2	3	3	3
treejoin	622	0	0	27	51	0	32	78	2	4	2	0	0	27	38	0	0	0	0	0	0	0	2	3	3	3
typecheck	7371	3	38	57	75	43	63	81	44	44	6	2	5	24	39	1	1	0	0	0	0	0	2	3	3	3
veritas	36308	25	43	632	907	51	795	1420	456	457	51	8	19	485	764	8	41	1	13	5	13	26	2	4	4	4
wang	5316	1	11	70	76	12	107	118	16	16	4	2	2	22	63	0	0	0	0	0	0	0	2	2	2	3
wave4main	8194	11	65	248	268	70	342	347	19	19	8	0	3	122	373	0	0	0	2	0	0	0	2	2	2	3

Figure 4.1 Transformation Count (1)

program	let float from											case float from									let to				
	app.				let			case				app.				case			let		case		η-exp.		
	3.4.1				3.4.2			3.4.3				3.5.1				3.5.2			3.5.3		3.6.1		3.7.2		
	b	c	d	e	c	d	e	b	c	d	e	b	c	d	e	c	d	e	d	e	d	e	c	d	e
boyer	70	63	63	70	833	838	901	0	0	0	55	0	0	0	0	3	4	7	0	0	9	5	0	0	0
boyer2	41	41	41	41	5640	5645	162	24	36	46	61	0	0	0	0	2	20	31	0	0	50	45	0	0	0
cichelli	38	38	38	39	308	308	90	6	6	14	15	0	0	0	0	0	16	37	0	0	23	14	1	1	2
clausify	17	16	16	17	46	51	39	6	8	8	11	0	0	0	0	0	6	43	0	0	9	15	0	0	0
comp_lab_z	146	83	84	147	217	262	87	21	9	18	45	0	0	0	0	3	77	119	0	12	60	72	0	1	0
compress	15	15	15	18	7916	7917	33	1	1	1	2	0	0	0	2	0	9	22	0	0	4	6	0	0	1
event	71	26	26	72	13	17	28	15	2	6	22	0	1	1	1	0	38	43	0	2	25	25	0	0	0
exp3_8	8	8	8	8	13	14	6	3	7	7	3	0	0	0	0	7	10	11	0	0	12	9	0	0	0
fft	86	47	47	86	33	66	70	16	7	29	96	0	0	0	0	1	90	118	0	7	41	41	0	0	0
fft2	37	32	44	38	32	44	47	3	3	3	34	0	0	1	0	0	16	34	0	0	34	46	0	3	2
fluid	478	478	478	489	1316	1435	1448	25	29	135	351	0	2	2	4	4	294	579	8	23	216	265	12	12	20
gen_regexp	13	13	13	13	16	26	5	2	4	9	11	0	0	0	0	0	20	19	0	1	17	14	0	0	0
genfft	98	35	35	98	914	928	1749	15	1	12	36	0	0	0	0	0	37	75	0	6	27	31	0	0	0
gg	303	306	309	312	472	592	738	51	65	71	198	0	2	2	9	1	159	244	0	21	225	192	6	8	9
hidden	112	116	116	113	149	179	271	21	20	51	107	0	0	1	0	4	184	232	0	0	50	38	7	8	8
hpg	246	259	271	299	768	810	854	7	7	13	24	0	2	6	16	0	79	87	5	13	91	49	15	27	30
ida	90	51	51	92	58	76	106	19	13	20	77	0	0	0	0	8	84	126	0	2	37	39	0	0	0
infer	148	167	171	171	198	227	241	3	3	8	19	0	4	5	5	0	23	28	1	3	36	26	11	14	24
knights	92	80	80	93	89	107	28	18	17	20	52	0	0	4	3	0	58	131	0	11	44	35	0	1	1
lift	71	71	72	76	4767	4779	446	6	7	8	28	0	0	0	2	0	22	51	2	2	35	51	1	2	7
listcompr	86	26	26	87	2317	2331	441	16	9	11	21	0	0	0	0	1	33	37	0	4	23	21	0	0	0
listcopy	87	26	26	88	2317	2332	443	16	9	13	21	0	0	0	0	1	37	41	0	4	24	22	0	0	0
maillist	48	48	48	48	10	16	60	1	1	1	1	0	0	0	0	0	8	22	0	4	8	6	0	0	1
mandel	31	31	31	33	47	49	64	1	1	1	17	0	0	0	0	0	44	45	0	0	4	4	0	0	1
mandel2	24	24	24	24	11	14	18	10	14	22	37	0	0	0	0	6	77	111	0	2	6	7	0	0	0
minimax	39	39	39	39	416	427	84	6	6	10	18	0	0	0	0	0	12	18	0	1	41	38	0	0	0
multiplier	78	84	93	86	165	173	359	5	7	18	24	0	0	0	0	0	22	50	0	14	42	50	0	3	3
parser	414	411	411	419	1862	2703	1073	40	46	51	839	0	0	0	0	0	779	849	0	2	315	246	0	2	4
parstof	450	418	419	453	9893	10123	2251	26	31	43	38	1	1	1	1	4	106	170	0	6	35	33	0	1	1
pretty	22	20	20	22	41	44	132	18	20	20	36	0	0	0	0	11	29	29	0	7	8	5	0	0	0
primes	3	3	3	3	0	1	1	0	0	0	1	0	0	0	0	0	2	2	0	0	1	1	0	0	0
primetest	63	64	65	64	33	36	28	9	9	22	33	0	0	0	0	1	110	194	0	2	23	22	1	2	2
prolog	117	128	127	117	253	272	383	16	16	21	41	0	0	0	2	0	19	38	0	3	39	40	3	3	3
queens	4	4	4	4	0	0	3	0	0	1	2	0	0	0	0	0	15	17	0	2	7	2	0	0	0
reptile	251	244	244	251	595	653	638	11	33	49	82	0	0	2	1	7	364	480	1	12	148	130	0	1	0
rewrite	145	139	138	147	123	195	241	35	47	55	118	0	0	1	1	3	31	61	0	1	79	74	1	2	3
rsa	27	27	33	28	31	38	31	0	0	18	16	0	0	0	0	0	36	60	0	4	14	24	0	1	1
sched	73	30	30	73	40	50	87	11	2	3	18	0	0	0	0	10	40	74	0	5	12	16	0	0	0
solid	191	56	56	193	495	506	201	19	18	41	71	0	0	0	0	31	218	270	0	4	13	21	0	0	0
sorting	36	34	32	39	33	42	65	2	2	2	13	0	0	0	8	0	3	16	0	0	18	22	1	1	1
transform	242	194	196	242	708	821	1108	30	35	41	114	0	0	0	0	9	108	153	0	6	95	110	0	1	1
treejoin	23	25	25	24	35	41	55	4	4	4	10	0	0	0	0	0	20	27	0	1	27	19	2	2	1
typecheck	87	52	52	88	750	779	530	15	9	14	56	0	0	0	0	1	23	34	0	1	44	47	0	0	0
veritas	906	935	943	940	9980	10108	6208	190	370	411	459	14	22	22	44	19	246	424	9	43	442	377	22	35	40
wang	65	27	27	66	53	112	127	11	1	3	34	0	0	0	0	0	106	160	0	0	17	27	0	0	0
wave4main	127	61	61	128	61	82	143	11	4	79	124	0	0	0	0	3	244	297	0	7	16	27	0	0	0

Figure 4.2 Transformation Count (2)

Simplifier Total Instructions Executed					
program	a	b	c	d	e
compress	1.22	1.00	0.68	0.67	0.53
reptile	1.12	1.00	0.73	0.70	0.50
sorting	1.13	1.00	0.74	0.77	0.51
listcompr	1.07	1.00	0.75	0.78	0.64
treejoin	1.20	1.00	0.75	0.57	0.26
listcopy	1.07	1.00	0.77	0.81	0.66
pretty	1.22	1.00	0.78	0.81	0.58
solid	1.20	1.00	0.82	0.49	0.39
ida	1.06	1.00	0.83	0.60	0.39
gen_regexps	1.67	1.00	0.84	0.71	0.45
hpg	1.20	1.00	0.84	0.84	0.43
maillist	1.28	1.00	0.84	0.80	0.41
gg	1.16	1.00	0.86	0.88	0.56
parser	1.32	1.00	0.86	0.80	0.55
parstof	1.43	1.00	0.86	0.57	0.45
prolog	1.19	1.00	0.86	0.79	0.48
genfft	1.06	1.00	0.88	0.79	0.53
lift	1.16	1.00	0.88	0.89	0.56
primes	1.17	1.00	0.88	0.68	0.29
sched	1.14	1.00	0.88	0.71	0.36
veritas	1.10	1.00	0.89	0.88	0.74
fluid	1.15	1.00	0.90	0.82	0.44
boyer2	1.25	1.00	0.91	0.83	0.59
comp_lab_z	1.19	1.00	0.91	0.73	0.62
fft2	1.06	1.00	0.91	0.83	0.30
rewrite	1.15	1.00	0.91	0.80	0.46
boyer	1.21	1.00	0.92	0.86	0.72
event	1.31	1.00	0.92	0.94	0.49
hidden	1.23	1.00	0.92	1.59	0.27
multiplier	1.13	1.00	0.92	0.90	0.65
typecheck	1.15	1.00	0.92	0.91	0.59
fft	1.13	1.00	0.93	0.76	0.55
mandel2	1.13	1.00	0.93	0.71	0.17
infer	1.20	1.00	0.95	0.92	0.52
transform	1.29	1.00	0.95	0.71	0.61
wave4main	1.33	1.00	0.95	0.96	0.33
mandel	1.19	1.00	0.96	1.11	0.36
minimax	1.42	1.00	0.97	0.95	0.61
cichelli	1.15	1.00	0.98	0.87	0.56
knights	1.19	1.00	0.98	0.87	0.57
wang	1.05	1.00	0.98	0.63	0.48
clausify	1.33	1.00	0.99	0.86	0.57
primetest	1.01	1.00	0.99	0.99	0.98
queens	1.19	1.00	0.99	0.77	0.17
rsa	1.02	1.00	0.99	0.98	0.98
exp3_8	1.00	1.00	1.00	1.00	1.00
Minimum	1.00	-	0.68	0.49	0.17
Maximum	1.67	-	1.00	1.59	1.00
Geom. mean	1.18	-	0.89	0.81	0.49

Simplifier Total Bytes Allocated					
program	a	b	c	d	e
compress	1.65	1.00	0.30	0.31	0.25
sorting	1.30	1.00	0.73	0.80	0.42
event	1.43	1.00	0.77	1.38	0.42
gen_regexps	2.17	1.00	0.77	0.70	0.44
solid	1.40	1.00	0.77	0.40	0.29
sched	1.21	1.00	0.80	0.74	0.33
pretty	1.42	1.00	0.81	0.82	0.53
parser	1.67	1.00	0.82	0.76	0.31
maillist	1.60	1.00	0.83	0.93	0.38
minimax	1.60	1.00	0.83	0.82	0.51
reptile	1.26	1.00	0.84	0.83	0.52
treejoin	1.54	1.00	0.84	0.83	0.23
gg	1.38	1.00	0.85	1.08	0.44
listcompr	1.17	1.00	0.86	0.93	0.63
listcopy	1.16	1.00	0.87	0.94	0.66
prolog	1.52	1.00	0.87	0.82	0.44
fluid	1.35	1.00	0.88	0.94	0.40
hpg	1.47	1.00	0.88	1.01	0.40
transform	1.42	1.00	0.88	0.54	0.53
rsa	1.23	1.00	0.90	0.87	0.83
wang	1.14	1.00	0.90	0.55	0.49
lift	1.31	1.00	0.91	0.95	0.53
primetest	1.21	1.00	0.91	0.89	0.85
veritas	1.24	1.00	0.91	0.88	0.69
clausify	1.26	1.00	0.92	0.67	0.43
ida	1.12	1.00	0.92	0.58	0.40
parstof	2.55	1.00	0.92	0.31	0.29
boyer	1.19	1.00	0.94	0.81	0.44
multiplier	1.18	1.00	0.94	0.93	0.55
rewrite	1.44	1.00	0.94	0.80	0.38
wave4main	1.89	1.00	0.94	1.45	0.23
boyer2	2.07	1.00	0.96	0.75	0.32
cichelli	1.62	1.00	0.96	0.62	0.28
hidden	1.46	1.00	0.97	3.15	0.32
knights	1.44	1.00	0.97	0.58	0.09
fft	1.19	1.00	0.98	0.93	0.70
infer	2.07	1.00	0.98	0.92	0.20
mandel	1.40	1.00	0.98	1.43	0.46
fft2	1.12	1.00	0.99	0.91	0.19
exp3_8	1.00	1.00	1.00	1.00	1.00
genfft	1.19	1.00	1.00	0.91	0.66
mandel2	1.18	1.00	1.00	0.89	0.15
primes	1.37	1.00	1.00	0.67	0.11
queens	1.38	1.00	1.00	0.59	0.06
typecheck	1.69	1.00	1.01	1.00	0.48
comp_lab_z	1.63	1.00	1.05	0.81	0.67
Minimum	1.00	-	0.30	0.31	0.06
Maximum	2.55	-	1.05	3.15	1.00
Geom. mean	1.41	-	0.88	0.82	0.38

Table 4.1 Simplifier: Instructions executed and bytes allocated

The poor results of two programs using strictness analysis in column (d) are due to the absence of the floating inwards transformation, which is presented in Chapter 5. We can see that these poor results disappear in column (e).

It is clear that the transformations themselves account for an improvement of at least 10% in both instructions executed and heap allocated. But as we mentioned before, they interact quite heavily with other transformations in the compiler, and we believe that their actual overall effect is greater than that.

4.3 Conclusions

We have presented details of the implementation of the simplifier pass in the Glasgow Haskell Compiler, and measured the effects of using that set of local transformations in the `nofib` benchmark suite.

The results show that the transformations presented, although small and simple, can have major effects in the performance of real programs, mostly due to the way they interact to achieve the effect of more complicated transformations.

Chapter 5

Let Floating

This chapter presents a collection of transformations that we call "`let` floating" transformations, because they concern the exact placement of `let` or indings. It was a big surprise to us that `let` floating can make a very substantial difference to a program's performance.

We distinguish between three forms of `let` floating: The first two are "long-distance" transformations (in that we may move the bindings very far from their original positions), while the third is a local one:

- The *floating inwards* moves bindings as far inwards as possible (Section 5.1).
- The *full laziness transformation* floats selected bindings out of enclosing lambda abstractions (Section 5.2)
- *Local floating* "fine-tunes" the location of bindings. The issues concerning local transformations (local floating of `let`s) were discussed in Section 3.4; therefore, in this chapter, we will only discuss local floating when it seems to conflict with other transformations, namely the *floating inwards* transformation.

We will also briefly discuss the floating of `case`s outside enclosing lambda abstractions (Section 5.3), which is related to the full laziness transformation.

5.1 Floating `let`s inwards

The floating-inward transformation is based on the following observation: *other things being equal, the further inward a binding can be moved, the better.* For example, consider the expression:

```
let x = y+1
in case z of
        []     -> x*x
        (p:ps) -> 1
```

Here, the binding for `x` is used in only one branch of the `case`, so it can be moved into that branch:

```
case z of
  [] -> let x = y+1
         in x*x
  (p:ps) -> 1
```

This code is better than the original for two reasons:

- Whenever `z` turns out to be of the form `(p:ps)` the closure for `x` is not allocated. Before the transformation a thunk (updatable closure) for `x` would be allocated regardless of the value of `z`.
- At the new position, the binding for `x` is guaranteed to be demanded (evaluated) in its body, since it is now used strictly in its body. This enables the `let` to be transformed into a `case` using the `let` to `case` transformation (Section 3.6.1), thereby allocating no thunks at all.

We have suggested that a binding can be floated inward "as far as possible", that is, to a point where it can be floated no further while still keeping all the occurrences of its bound variable in scope. There is an important exception to this rule: *it is highly dangerous to float a binding into a lambda abstraction.* The problem is that if a `let` is moved into a lambda it will be allocated *every* time the lambda is entered! As we cannot usually anticipate how many times a lambda will be entered during the execution of a program, we must not take the risk of increasing the allocation by an unknown factor. This is a consequence of the fact that our evaluation strategy is not fully lazy, as is often the case in recent implementations of functional languages. If our evaluation strategy was based on SK combinators [Tur79], for example, which are fully lazy, we would not need this restriction.

We are not aware of any work which suggests this transformation in the context of lazy functional languages, especially for improving strictness analysis. Like many of the transformations in this thesis, it was suggested by inspecting the actual code generated by the Glasgow Haskell Compiler.

5.1.1 Benefits of floating inwards

Let us see which benefits this transformation is trying to achieve. Some closures may initially be defined in a scope much larger than needed. This is particularly harmful in cases where, if they were defined in a more localised context, one could:

- ✓ *Reduce allocation* by moving bindings into a single `case` branch:

```
let a = E                          case x of
in case x of                         alt1 -> let a = E in a + a
     alt1 -> a + a          ==>      alt2 -> b
     alt2 -> b
```

  Before the transformation, the closure for `a` would be allocated regardless of which branch was taken, although it would only be needed if `alt1` was the branch taken. After the transformation it is only allocated if the `alt1` branch is taken.

  The same might happen when floating inwards into a `let` right hand side:

```
let a = E                        let b = let a = E
in let b = a + a         ==>             in a + a
in (b,b)                         in (b,b)
```

  The details on the advantages and disadvantages of each of the two forms (for `let`s) are discussed in Section 3.4.2, where exactly the opposite transformation is presented and discussed. We discuss the apparent incompatibility between these two transformation in Section 5.1.4.

- ✓ *Increase opportunities for the* `let` *to* `case` *transformation* (Section 3.6.1), by moving the closure to a local context in which the closure may be used strictly (demanded). Using the same example above regarding `case`s, after being floated into the branch, `a` is guaranteed to be demanded in its new context (as `+` is strict in its arguments). Before, it would not be demanded if `alt2` was the branch taken. As it is guaranteed to be demanded, we can use the `let` to `case` transformation:

```
case x of                             case x of
  alt1 -> let a = E in a + a   ==>      alt1 -> case E of a -> a + a
  alt2 -> b                             alt2 -> b
```

The same might happen when floating inwards into a `let` right hand side:

```
let b = let a = E                let b = case E of a -> a + a
        in a + a             ==> in (b,b)
in (b,b)
```

✓ *Increase opportunities for the* `case` *reduction transformation.* Another example of optimisations that can happen after floating `let`s inwards is the `case` reduction transformation (Section 3.3.1):

```
let x = case y of (a,b) -> a
in case y of
     (p,q) -> E
```

If the binding for `x` is moved into the `case` branch, we get:

```
case y of
 (p,q) -> let x = case y of (a,b) -> a
          in E
```

Now the compiler can spot that the inner `case` for `y` is in a branch of an enclosing `case` which also scrutinises `y`. We can therefore eliminate the inner `case` (and then inline `x`):

```
case y of                    ==> case y of
 (p,q) -> let x = p in E           (p,q) -> E[p/x]
```

5.1.2 Risks of floating inwards

Possible disadvantages of floating inwards are:

□ It may increase (or decrease) closure sizes (due to the change in the number of free variables of closures after floating) and consequently increase (or decrease) the total heap allocation. The "moving" `let` is unaffected, but the `let` into which it is moving will have the free variables of the "moving" `let` as new free variables (if they weren't already free variables) less one, which is the variable bound by the "moving" `let` itself. We expect that it will often increase the number of free variables, since it will only reduce the number of free variables if the "moving" `let` has no free variables itself, or if its free variables are already free variables of the `let` right hand side it is being moved into.

- □ it may increase (or decrease) the number of heap checks, since `let`s that would originally be allocated together (and therefore perform a single heap check) may now be split into separate groups demanding extra heap checks. But it is also possible that a `let` (due to floating) joins another group of `let`s, and therefore the number of heap checks is reduced.
- □ it may hide (or present) opportunities for other transformations. The issues related to this are discussed in Section 5.4.

5.1.3 Implementing floating inwards

The algorithm we use for floating `let`s inwards is presented in Figure 5.1.3.

The floating inward function ($\mathcal{FI}[\![\,]\!]$) takes as arguments an expression and a list of bindings that are to be pushed into that expression, and returns a new expression with the bindings pushed into its subexpressions as far as possible. The algorithm keeps the following invariant:

$\mathcal{FI}[\![E]\!]_{\rho} \;=\; \texttt{let}\ \rho\ \texttt{in}\ E$

The algorithm works by collecting the bindings in an environment and, for each expression:

- Drop the bindings just outside the expression, if the expression does not contain subexpressions (a, b, c, d in Figure 5.1.3) or is a lambda expression (e in Figure 5.1.3).
- Try to push at least some of the bindings into the subexpressions (which we will call branches or "drop points") of the expression. This is done by the *sepByDropPoint* function, which checks which binders are used in only one of the branches (f, g, h, i, j, k in Figure 5.1.3). To know which bindings are used in each of the branches we use the *fv* function, that returns the free variables of an expression.

  For `case`s the possible "drop points" for the bindings are:

  - in the `case` scrutinee;
  - in the `case` alternatives.

  For `let`s, the possible "drop points" for the bindings are:

  - in the body,

$$\mathcal{FI}[\![\ ]\!] \ :: \ Expr \ \rightarrow \ [Binding] \ \rightarrow \ Expr$$

$$(a) \quad \mathcal{FI}[\![k]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ k]\!]$$

$$(b) \quad \mathcal{FI}[\![v]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ v]\!]$$

$$(c) \quad \mathcal{FI}[\![C\ v_1 \ldots v_n]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ C\ v_1 \ldots v_n]\!]$$

$$(d) \quad \mathcal{FI}[\![op\ v_1 \ldots v_n]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ op\ v_1 \ldots v_n]\!]$$

$$(e) \quad \mathcal{FI}[\![\lambda v_1 \ldots v_n.E]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ \lambda v_1 \ldots v_n.\mathcal{FI}[\![E]\!]_{\emptyset}]\!]$$

$$(f) \quad \mathcal{FI}[\![\Lambda t_1 \ldots t_n.E]\!]_{\rho} \quad = \quad [\![\Lambda t_1 \ldots t_n.\mathcal{FI}[\![E]\!]_{\rho}]\!]$$

$$(g) \quad \mathcal{FI}[\![E\ T]\!]_{\rho} \quad = \quad [\![\mathcal{FI}[\![E]\!]_{\rho}\ T]\!]$$

$$(h) \quad \mathcal{FI}[\![E\ v_1 \ldots v_n]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho'\ \rho_1 \ldots \rho_n\ \texttt{in}\ \mathcal{FI}[\![E]\!]_{\rho_0}\ v_1 \ldots v_n]\!]$$
$$\textit{where}$$
$$[\rho_0, \ldots, \rho_n, \rho'] = \textit{sepByDropPoint}\ [\textit{fv}\ E, \{v_1\}, \ldots, \{v_n\}]\ \rho$$

$$(i) \quad \mathcal{FI}[\![\texttt{case}\ E_0\ \texttt{of}\ \{alt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!]_{\rho}$$
$$= \quad [\![\texttt{let}\ \rho'\ \texttt{in}\ \texttt{case}\ (\mathcal{FI}[\![E_0]\!]_{\rho_0})\ \texttt{of}\ \{alt_i\ \texttt{->}\ \mathcal{FI}[\![E_i]\!]_{\rho_i}\}_{i=1}^{n}]\!]$$
$$\textit{where}$$
$$[\rho_0, \ldots, \rho_n, \rho'] = \textit{sepByDropPoint}\ [\textit{fv}\ E_0, \ldots, \textit{fv}\ E_n]\ \rho$$

$$(j) \quad \mathcal{FI}[\![\texttt{let nonrec}\ v\ \texttt{=}\ E_v\ \texttt{in}\ E_0]\!]_{\rho}$$
$$= \quad [\![\mathcal{FI}[\![E_0]\!]_{\rho_0 +\!\!+ [\texttt{nonrec}[(v, \mathcal{FI}[\![E_v]\!]_{\rho_v})]] +\!\!+ \rho'}]\!]$$
$$\textit{where}$$
$$[\rho_0, \rho_v, \rho'] = \textit{sepByDropPoint}\ [\textit{fv}\ E_0, \textit{fv}\ E_v]\ \rho$$

$$(k) \quad \mathcal{FI}[\![\texttt{let}\ b\ \{v_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E_0]\!]_{\rho}$$
$$= \quad [\![\mathcal{FI}[\![E_0]\!]_{\rho_0 +\!\!+\ [b[(v_1, \mathcal{FI}[\![E_1]\!]_{\rho_1}), \ldots, (v_n, \mathcal{FI}[\![E_n]\!]_{\rho_n})]] +\!\!+ \rho'}]\!]$$
$$\textit{where}$$
$$[\rho_0, \ldots, \rho_n, \rho'] = \textit{sepByDropPoint}\ [\textit{fv}\ E_0, \ldots, \textit{fv}\ E_n]\ \rho$$

Figure 5.1 Algorithm for floating inwards

- in the right hand side of a non-recursive binding,
- in each of the right hand sides of a

- Drop the bindings that are used in more than one branch just outside the expression.

Notice what we do with let-bindings: consider:

```
let w = ...
in let v = ... w ...
   in ... w ...
```

Look at the inner let. As w is used in both the right hand side and the body of the inner let, we could panic and leave w's binding where it is. But if v is floatable into its body then w will *also* be floatable into the body.

So rather than drop w's binding here, we add it onto the list of things to drop and let the decision of where to drop it to be made later.

It is important to keep the list of bindings to be dropped in a specific order, as this will help us during the partitioning of the list by drop points. Earlier bindings in the list may use (i.e. refer to) later bindings in the list, but not the reverse.

Improving the let rec rule

A possible improvement in the rule for recursive lets would be to break recursive groups, by introducing local recursion if one or more binders are only used in one of the bindings. For example:

```
let rec a = ...b...          let rec a = let rec b = ...a...
        b = ...a...     ==>              in ...b...
in ...a...                   in ...a...
```

since b is only used in a's right hand side. We do not perform this optimisation, as we believe this is a rare case, and it introduces extra complexity into the algorithm.

Separating the bindings by drop points

The function that separates the bindings by drop points (*sepByDropPoint*) is the crucial function. The idea is: we have a list of bindings that we would like to distribute inside a collection of *drop points*; inside the alternatives of a `case` would be one example of some drop points; the right hand sides and body of a `let`-binding would be another example.

The algorithm proceeds as follows: we are given a list of sets of free variables, one per drop point, and a list of floating-inwards bindings. Then we have three possibilities:

(*a*) A binding is not used in any of the drop points: it is therefore dead code, and we can remove it from the list.

(*b*) If a binding can go into *only one* drop point, in it goes. *But now its free variables are also free variables of that drop point*, therefore we should use a new version of the list of sets of free variables when looking for a drop point for other bindings, updated to take this fact into account.

(*c*) if a binding is used inside *multiple* drop points, then it has to go in a "you must drop it above all these drop points" point. This also means that its free variables cannot go into a single drop point either, so we update the list of sets of free variables to take this fact into account. A simple way of doing it is by making its free variables part of the sets that contain that binding (or part of all sets, as we do in Figure 5.2)

Maintaining the order on the bindings' lists (with the ones occurring first having references to the ones occurring later on the list) allows us to process the list in one traversal.

5.1.4 Relation to local `let` floating

Since this transformation and the ones that float `let`s outwards (`let` floating from `let` in Section 3.4.2, `let` floating from application in Section 3.4.1, `let` floating from `case` scrutinee in Section 3.4.3) do opposite things, let us see how they can be used together.

Although initially one transformation seems to undo what the other ones do, in fact the objectives that we are trying to achieve by floating inwards are not affected

$$
\begin{array}{ll}
\mathit{sepByDropPoint} :: [[\mathit{Var}]] \rightarrow [\mathit{Binding}] \rightarrow [[\mathit{Binding}]] & \\
\mathit{sepByDropPoint}\ [p_1, \ldots, p_n]\ [\,] & \\
\quad = [[\,]_1, \ldots, [\,]_n, [\,]] & \\
\mathit{sepByDropPoint}\ [p_1, \ldots, p_n]\ (\mathit{bind} : \mathit{binds}) & \\
\quad \mid \mathit{bind} \notin \bigcup_n^{i=1} p_i & (a) \\
\quad = \mathit{sepByDropPoint}\ [p_1, \ldots, p_n]\ \mathit{binds} & \\
\quad \mid \exists! i. \mathit{bind} \in p_i & (b) \\
\quad = [d_1, \ldots, \mathit{bind}\texttt{++}d_i, \ldots, d_n, \mathit{multd}] & \\
\quad \mathit{where} & \\
\qquad [d_1, \ldots, d_n, \mathit{multd}] & \\
\qquad\qquad = \mathit{sepByDropPoint}\ [p_1, \ldots, (p_i \cup \mathit{fvs}\ \mathit{bind}), \ldots, p_n]\ \mathit{binds} & \\
\quad \mid \mathit{otherwise} & (c) \\
\quad = [d_1, \ldots, d_n, \mathit{bind}\texttt{++}\mathit{multd}] & \\
\quad \mathit{where} & \\
\qquad [d_1, \ldots, d_n, \mathit{multd}] & \\
\qquad\qquad = \mathit{sepByDropPoint}\ [(p_1 \cup \mathit{fvs}\ \mathit{bind}), \ldots, (p_n \cup \mathit{fvs}\ \mathit{bind})]\ \mathit{binds} &
\end{array}
$$

Figure 5.2 *sepByDropPoint* function

by later floating outwards transformations, as we will see. Therefore we *first* float inwards and then float outwards, usually performing other transformations that take advantage of the new contexts exposed by floating inwards in between (e.g. strictness analysis and the `let` to `case` transformation). *Then* we allow local floating to decide where to place the remaining `let`s into other `let`s' right hand sides.

Let us see why we are not losing the benefits of the floating inwards transformation by later floating outwards:

- When `let`s are being pushed into other `let`s right hand sides, we are trying, by use of local strictness information, to increase the chances that the `let` being pushed will be transformed into a `case` by the `let` to `case` transformation (3.6.1). We are therefore assuming that the strictness analyser and the transformations related to it will be applied *before* we do any floating outwards. If the `let` to `case` transformation does not happen, we leave the decision of where to place the `let` to the transformations that float `let`s out, as discussed in Section 3.4.2.
- When `let`s are being pushed into `case` branches we expect:
  1. To increase the chances that the `let` being pushed will be transformed into a `case` by the `let` to `case` transformation, just as discussed above.

$$\mathcal{FI}[\![E_0\ v_1 \ldots v_n]\!]_{\rho} \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ \mathcal{FI}[\![E_0]\!]_{\emptyset}\ v_1 \ldots v_n]\!]$$

$$\begin{aligned}&\mathcal{FI}[\![\texttt{case}\ E_0\ \texttt{of}\ \{alt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!]_{\rho}\\ &\qquad = [\![\texttt{let}\ \rho'\ \rho_0\ \texttt{in case}\ (\mathcal{FI}[\![E_0]\!]_{\emptyset})\ \texttt{of}\ \{alt_i\ \texttt{->}\ \mathcal{FI}[\![E_i]\!]_{\rho_i}\}_{i=1}^{n}]\!]\\ &\qquad \textit{where}\\ &\qquad\quad [\rho_0, \ldots, \rho_n, \rho'] = \textit{sepByDropPoint}\ [\textit{fv}\ E_0, \textit{fv}\ E_1, \ldots, \textit{fv}\ E_n]\ \rho\end{aligned}$$

Figure 5.3 Algorithm for floating inwards – Modified Rules

If this transformation takes place we will no longer have a `let`-binding to (possibly) float out again.

2. To avoid closures being allocated regardless of which branch will be taken, therefore saving allocations. When performing local `let` floating outwards we do not float `let`s out of multi-branch `case`s exactly to avoid creating this problem, therefore the transformation will not be undone. Actually the full laziness transformation (Section 5.2) may still decide to float `let`s out of multi-branch cases if the `let` is going to be floated past a lambda. In this case, *some* of the `let`s may be floated outwards again, hoping that the increased possibility that it will be shared is more important than avoiding the allocation when some branches are taken, that is, we prioritise full laziness over floating inwards. We return to this point in Section 5.2.5.
3. To expose more opportunities for the `case` reduction transformation. If this happens the `let` will not be able to move out of the `case` branch anymore.

The other transformations in the algorithm, namely floating inwards for applications and `case` scrutinees, do not have any effects unless this leads to floating into `let`s and/or `case` branches. In particular, if the floating `let`s out of `case` scrutinees (Section 3.4.3), floating `let`s out of applications (Section 3.4.1), floating `case` out of applications (Section 3.5.1) and `case` of `case` (Section 3.5.2) transformations have already being applied to the code, there will not be `case`s or `let`s in `case` scrutinees or in the function position in applications. These rules could therefore be simplified to drop all bindings immediately when finding an application and to drop the bindings that would be floated into the `case` scrutinee outside the `case`. This would lead to the two rules being modified as presented in Figure 5.3.

5.1.5 Improvements to the algorithm

There were some improvements to the algorithm that were not obvious when we first implemented the algorithm. They were often suggested by looking at a few programs that were actually getting worse after performing the transformation. These improvements are described below.

Dropping `let`s outside type lambdas

Dropping `let`s outside type lambdas (as we do for normal lambdas) if the `let` would otherwise be dropped just in between type lambdas and normal lambdas, e.g.

```
let y = ...                      =/=> let f = /\t -> let y = ...
in let f = /\t -> \x ->...y...                       in \x ->...y...

let y = ...                      ===> let f = let y = ...
in let f = /\t -> \x ->...y...                in /\t -> \x ->...y...
```

Both resulting expressions above are identical in performance (as the type lambdas will eventually be removed), and both offer the same opportunities for transformations after `y` is floated into `f`'s right hand side. The reason for preferring the second form occurs when nothing happens with the floated `let`, in which case we might want to move it back to the original position using local `let` floating (and therefore turning `f` back into a weak head normal form, that is, into a non-updatable closure). It is not easy to move the binding for `y` out again due to the type lambda, as floating `let`s out of type lambdas (and out of lambdas, as we will see in Section 5.2) is a much more complicated (global) transformation: we need to check that the type variable `t` is not used in `y`'s right hand side to allow it to float out of the type lambda. Although this is certainly true in this case, we avoid the complication of (maybe) relying in another global transformation to fix this problem by dropping the `let` outside type lambdas. This is achieved by introducing the extra rule in Figure 5.4.

This problem caused one of the programs (`cichelli`) to execute 10% more instructions with floating inwards enabled.

Duplicating `let`s into `case` branches

We may want to duplicate `let`s which are used in more than one branch of a `case`, although this may generate some code duplication. By doing this there is no risk of

$$\mathcal{FI}[\![\Lambda t_1 \ldots t_n.\lambda v_1 \ldots v_n.E]\!]_\rho \quad = \quad [\![\texttt{let}\ \rho\ \texttt{in}\ \Lambda t_1 \ldots t_n.\lambda v_1 \ldots v_n.\mathcal{FI}[\![E]\!]_\emptyset]\!]$$

Figure 5.4 Algorithm for floating inwards – extra rule for type lambdas

duplicating work, and we may actually end up saving the allocation of the `let` if it is only used in *some* of the `case` branches, e.g. if it is used in two out of three branches we would end up not allocating the `let` whenever the third branch is taken.

We actually only perform this more aggressive version of floating in a specific circumstance: when the `let` is binding a constructor. This has very minor effects in our usual optimisation setup as can be seen in the table below (there was no effect on the other programs):

program	object size	instructions executed	allocation
`parser`	0.99	0.99	1.00
`boyer2`	1.00	1.00	0.98
`comp_lab_zift`	1.00	1.00	0.98
`treejoin`	1.00	1.00	0.99
`ida`	1.00	1.00	1.01
`prolog`	1.00	1.00	1.02
`fluid`	1.01	1.00	1.00
`gg`	1.01	1.00	1.00
`veritas`	1.01	1.00	1.00
`rewrite`	1.02	1.00	1.00

But when we tried more aggressive inlining strategies (Chapter 6) we found that some *reboxing* constructors created by the worker-wrapper transformation [PP93] were causing a lot more allocation, since they now had less opportunities to be floated inwards (due to the amount of code duplication caused by inlining). Let us look at one simple example:

```
f x y = let pair = (x,y)
        in let g = ...pair...
        in case x of
             0 -> ...g...
             1 -> ...g...
             y -> ...
```

suppose g is a "join point" (therefore it does not cost anything in terms of allocation) and is not mentioned in the third case alternative. If g is not inlined we will push pair into g's right hand side. But if g is inlined then pair would have to be left out (as it will be used in two case alternatives) and we will always allocate it. This caused sometimes an increase of up to 11% in instructions executed and 42% in heap allocated. The improvement from the use of this more aggressive floating inwards strategy in programs compiled with an aggressive inlining strategy is shown in the table below.

program	Normal Floating In object size	instr. exec.	alloc.	Aggressive Floating In obj. size	instr. exec.	alloc.
`treejoin`	1.00	1.00	1.00	1.00	0.89	0.79
`wave4main`	1.00	1.00	1.00	1.00	0.90	0.58
`maillist`	1.00	1.00	1.00	1.00	0.97	0.86
`fft`	1.00	1.00	1.00	1.00	0.99	0.96
`comp_lab_zift`	1.00	1.00	1.00	1.00	1.00	0.98
`fluid`	1.00	1.00	1.00	1.00	1.00	0.98
`hpg`	1.00	1.00	1.00	1.00	1.00	0.98
`prolog`	1.00	1.00	1.00	1.00	1.00	0.98
`listcompr`	1.00	1.00	1.00	0.98	1.00	1.00
`veritas`	1.00	1.00	1.00	0.99	1.00	1.00
`ida`	1.00	1.00	1.00	1.00	1.00	1.01
`primetest`	1.00	1.00	1.00	1.00	1.00	1.03
`rsa`	1.00	1.00	1.00	1.00	1.00	1.03
33 other progs.	1.00	1.00	1.00	1.00	1.00	1.00
Minimum	-	-	-	0.98	0.89	0.58
Maximum	-	-	-	1.00	1.00	1.03
Geometric Mean	-	-	-	1.00	0.99	0.98

This change in the strategy for floating lets into case branches (when they are bound to constructors) is certainly worthwhile, and had no major impact in the program size.

Another possible solution would be only to push lets that occur in *some* branches. We tried this option, but sometimes, although used in *all* branches at an outer level, a let is used only in *some* branches of an inner case (in one or more of the outer case's branches). This was the case in one of our worst performing examples. Since, as we have shown, the overhead of *always* floating these lets is very small we decided always to float them.

5.1.6 Results

In this section we present the effect of the floating inwards transformation (including the modifications presented in the previous section) on programs in the **nofib** benchmark suite. We will be looking at whether the transformation actually achieves the effects we presented in Section 5.1.1 and also if it is affected by any of the possible drawbacks presented in Section 5.1.2.

One of the first questions to be answered is *how much allocation is saved as a result of floating inwards?* Moving the bindings inwards may increase or decrease the closure size (due to the change in the free variables of the closure), therefore it would be possible for example to reduce the number of objects allocated but nevertheless to increase the heap allocation! Therefore we compare the total heap allocated when programs are run with and without floating inwards enabled, in Figure 5.1. Both runs are with strictness analysis turned off, so that we measure only the benefits from floating inwards, and not the additional benefits of `let` to `case` transformations exposed by floating inwards. The improvement in total heap allocated, although being as high as 45% for one of the programs was on average of about 3%. A few programs had their allocation increased, which is probably due to some closures having their size increased and the program not taking any of the benefits of floating inwards. Other measurements comparing these two setups gave the following results:

- there was no effect (on average) on the average closure size, although the maximum effects were an increase of 6% and a decrease of 7%.
- heap checks were also unaffected on average, but there were extremes with up to 51% more heap checks and 27% less.
- on instructions executed the maximum improvement was of up to 29%, although on average the improvement was only of 1%.
- There were on average 3% more instances of the `case` reduction transformation.

But this is not the only benefit we are trying to get from floating inwards. Our next question is *how much is the strictness analyser helped by floating inwards?* This time we start by comparing the number of `let` to `case` and `case` floating from `let` transformations that occurred during compilation, since these are the main transformations we will enable by strictness analysis. We got on average:

- 10% more `let` to `case`s;

Float In - No Strictness Analysis Total Instructions Executed		
	Float In	
program	off	on
`sched`	1.00	0.71
`boyer`	1.00	0.92
`boyer2`	1.00	0.97
`primes`	1.00	0.97
`hpg`	1.00	0.99
`maillist`	1.00	0.99
`mandel`	1.00	0.99
`parser`	1.00	0.99
`parstof`	1.00	0.99
`cichelli`	1.00	1.01
`typecheck`	1.00	1.01
`mandel2`	1.00	1.03
`rewrite`	1.00	1.05
`treejoin`	1.00	1.06
32 other programs	1.00	1.00
Minimum	1.00	0.71
Maximum	1.00	1.06
Geometric mean	1.00	0.99

Float In - No Strictness Total Heap Allocated		
	Float In	
program	off	on
`sched`	1.00	0.55
`boyer`	1.00	0.75
`parstof`	1.00	0.78
`boyer2`	1.00	0.86
`parser`	1.00	0.88
`cichelli`	1.00	0.93
`mandel2`	1.00	0.95
`comp_lab_zift`	1.00	0.97
`maillist`	1.00	0.98
`ida`	1.00	0.99
`rewrite`	1.00	0.99
`sorting`	1.00	1.01
`treejoin`	1.00	1.01
`solid`	1.00	1.02
`wang`	1.00	1.03
31 other programs	1.00	1.00
Minimum	-	0.55
Maximum	-	1.03
Geometric mean	-	0.97

Table 5.1 Float In - No Strictness: instructions executed and bytes allocated

- 5% more `case` floating from `lets`;
- 2% more `case` reductions;
- 3% less heap checks.

The overall effect (this time including strictness analysis) on heap allocated and instructions executed due to floating inwards is presented in Figure 5.2. More programs are affected than before, and we get even better results, with a peak of 56% reduction in heap allocation and an average improvement of 6%. Again a few programs get a slightly higher allocation, up to 4%. The average effect on instructions executed is relatively small (2%), but some programs improved as much as 38%.

Other important effects of the transformation were:

- its reduction in the total number of updates performed, which is reduced by as much as 78% in one program (`mandel2`), but was on average of 6%.
- on average no change in the average closure size, although it was increased by up to 16% and reduced by up to 22%.

Float In Total Instructions Executed		
	Float In	
program	off	on
sched	1.00	0.62
mandel2	1.00	0.70
wave4main	1.00	0.91
ida	1.00	0.96
treejoin	1.00	0.97
maillist	1.00	0.98
boyer2	1.00	0.99
cichelli	1.00	0.99
comp_lab_zift	1.00	0.99
compress	1.00	0.99
event	1.00	0.99
fft	1.00	0.99
fluid	1.00	0.99
parser	1.00	0.99
parstof	1.00	0.99
solid	1.00	1.01
wang	1.00	1.01
29 other programs	1.00	1.00
Minimum	-	0.62
Maximum	-	1.01
Geometric mean	-	0.98

Float In Total Heap Allocated		
	Float In	
program	off	on
sched	1.00	0.44
wave4main	1.00	0.50
mandel2	1.00	0.71
parstof	1.00	0.79
treejoin	1.00	0.79
maillist	1.00	0.89
cichelli	1.00	0.90
parser	1.00	0.90
boyer	1.00	0.93
comp_lab_zift	1.00	0.94
ida	1.00	0.95
fft	1.00	0.97
knights	1.00	0.97
boyer2	1.00	0.98
clausify	1.00	0.98
event	1.00	0.98
fluid	1.00	0.98
hpg	1.00	0.98
prolog	1.00	0.98
compress	1.00	0.99
gg	1.00	0.99
lift	1.00	0.99
rewrite	1.00	0.99
solid	1.00	1.03
wang	1.00	1.04
21 other programs	1.00	1.00
Minimum	-	0.44
Maximum	-	1.04
Geometric mean	-	0.94

Table 5.2 Float In: instructions executed and bytes allocated

5.1.7 Related work

The idea of pushing `lets` into (the equivalent of) `case` branches is used by Appel in his SML compiler [App92], where it is called *hoisting down*. As SML is a strict language, this actually *always* saves the *evaluation* of the `let` if other branches are taken, while we are only guaranteed to save the closure allocation (although we may benefit from the new local context exposing other transformations). He does not push `lets` into `lets`, as there is no benefit from doing that, because SML is a strict language and therefore the `let` would be evaluated anyway. The presence of side effects in the language restrict the class of `lets` suitable for this transformation to "purely functional" ones (side-effect free).

He also uses hoisting (up or down) to group `lets` together, in order to (possibly) share closures and therefore reduce allocation and code size. We do not have this optimisation, and the only possible benefit of grouping `lets` would be to reduce heap checks, as we discussed in Section 3.4.2.

He implemented the transformation as a local transformation, and that seems to be suitable in his case because he does not push `lets` into `lets`, which (to be done in the best way and in a single pass) accounts for our more complex algorithm.

He did not get good results from this transformation (his best results were improvements of about 4%), but the experiments presented in [App92] are based on only six programs.

5.1.8 Conclusion

The majority of the programs are unaffected by the transformation, which is reflected in the low average improvement. On the other hand the effect is reasonably big for the programs it hits. Since there is very little overhead during the compilation process to perform this transformation and it has no major negative effects, we believe this is a useful optimising transformation for a compiler.

Another interesting fact is that, although some of the opportunities for applying this transformation are introduced by desugaring or by other transformations, many of them are already present in the original program, as programmers tend to group all local definitions in a single `let` or `where` clause, instead of defining them in a nested way or closer to the place where they are used (e.g. in the `case` branch where they are used).

Also, by knowing this transformation will be performed, the programmer may write his code in a more readable style, without worrying about doing this sort of transformation by hand.

5.2 Full laziness

In the previous section we concluded that floating a binding inwards is generally a good thing. But, if a binding can be floated *out of an enclosing lambda abstraction*, then its evaluation will become shared among all the applications of that abstraction, and even larger gains may accrue. For example, consider:

```
f = \xs -> let rec g = \y -> let n = length xs
                             in ...g...n...
           in ...g...
```

Here, `length xs` will be allocated and recomputed on each recursive call to `g`. This recomputation can be avoided by floating the binding for `n` outside the `\y`-abstraction:

```
f = \xs -> let n = length xs
           in let rec g = \y -> ...g...n...
              in ...g...
```

This transformation is called *full laziness*. It was originally invented by Hughes ([Hug83],[Pey87]) who presented it as a variant of the supercombinator lambda-lifting algorithm. [PL91b] subsequently showed how to decouple full laziness from lambda lifting by regarding it as an exercise in floating `let (rec)` bindings outwards. [Tak88] also presents full laziness as a separate transformation.

The need for a full laziness transformation is due to our evaluation strategy not being fully lazy. It is possible to implement fully lazy evaluation strategies (e.g. [Tur79], based on combinators) but these implementations are much less efficient than current implementations based on graph reduction [Pey87].

Despite being around for so long, the full laziness transformation has not made it into any functional language compiler we know of. One possible reason for this is the risk of introducing space leaks, e.g. by sharing a big data structure that would be rebuilt in the original code. We will return to this point in Sections 5.2.2 and 5.2.3.

Our contribution here is that we present results from actual use of the full laziness transformation, showing not only that its use may improve program performance quite substantially, but also that the risk of creating space leaks, although present, may not happen very often in real programs. We also present ways of reducing the risk of space leaks, while retaining some of the benefits of full laziness and improve the algorithm presented in [PL91b] by preventing some unnecessary floating from taking place (which could have some performance implications).

5.2.1 Benefits of full laziness

The full laziness transformation (as we will see in Section 5.2.4) can be regarded as floating `let`s out of lambdas, since the algorithm will `let`-bind any other expressions that can be floated out (and shared). Based on this, when discussing it we will often regard it as just floating `let`s out of lambdas.

Full laziness has the following benefits:

✓ The full laziness transformation can save a great deal of repeated work. One might object that in practice programmers don't write such programs, but it sometimes applies in non-obvious situations. One example we came across in practice is part of a program which performed the Fast Fourier Transform (FFT). The programmer wrote a list comprehension similar to the following:

```
[xs_dot (map (do_cos k) (thetas n)) | k<-[0 .. n-1]]
```

What he did not realise is that the expression `(thetas n)` was recomputed for each value of `k`! The list comprehension syntactic sugar was translated into the Core language, where the `(thetas n)` appeared inside a function body. The full laziness transformation lifted `(thetas n)` out past the lambda, so that it was only allocated and computed once (this example was only discovered because the programmer was trying to find the reason for the widely differing performance of his program using different Haskell compilers).

It should now be clear why we remarked in previous sections that we should beware of floating a binding inside a lambda abstraction: doing so is the exact reverse of the full laziness transformation, and can duplicate an arbitrary amount of work.

Lifting things out from inside a lambda is particularly significant for *loops*, i.e. for recursive functions. Consider:

```
f = \ x -> let y = fib 20
           in case x of
```

```
0 -> y
n -> let z = n - 1 ;
         v = f z
     in y + v
```

Floating `y` out of `f` will mean that it is allocated and computed once rather than at every call to `f`, saving an unbounded number of recomputations and allocations.

However it is not enough to consider only recursive functions: even if `f` is non-recursive, it might be called from another recursive function, or passed as an argument to a higher-order function; hence even non-recursive functions may be called an unbounded number of times.

Notice that the objective achieved is a generalisation of the idea of removing an invariant from a loop in an imperative language [ASU87].

✓ If the right hand side of the `let` being floated is a weak head normal form, no recomputation work is saved by sharing it among many invocations of the same function, but some allocation may nevertheless be saved by avoiding the reallocation of the object multiple times.

✓ The full laziness transformation also interacts with other transformations, exposing opportunities for their use. Some examples are

- inlining:

  ```
  f a = let g x = x + 1     ==>     g x = x + 1
        in h g a                    f a = h g a
  ```

  `f`'s right hand side becomes very simple, which allows `f` to be inlined.

- eta expansion: using the same example, supposing we would not inline `f`, if `h` has arity 3, after `g` is floated `f` can be eta abstracted, but not before, as laziness would be lost. The expression then becomes

  ```
  g x = x + 1
  f a b = h g a b
  ```

5.2.2 Risks of full laziness

The risks of performing full laziness are:

× If the lambda from which a `let` is being floated is never entered, we risk allocating an object (the `let`) that would never be allocated in the original program.

□ Full laziness gives no gain at all if the lambda abstraction is applied no more than once. There are program analyses which detect when a lambda abstraction is applied only once, but we do not make use of such analyses, since they have just recently become available [Mar93, MTW95].

□ The full laziness transformation may modify the number of free variables of closures, thereby increasing or decreasing their size. The `let` being floated itself is not modified, but other bindings' free variables may be decreased due to the floating (since the "floating" binding's free variables may no longer be free variables of the binding it is being moved from, although the variable bound by that binding will now be a free variable) or increased (if the only effect in the free variables is the extra free variable for the binding being floated). This has the effect of increasing or decreasing closure sizes and therefore increasing or decreasing heap allocation.

× There is a final disadvantage to the full laziness which is much more difficult to quantify: it may cause a space leak. Consider the expression:

```
f = \x -> let a = [1..n] in <body>
```

where `[1..n]` returns the list of integers between `1` and `n`. Is it a good idea to float the binding for `a` outside the `\x`-abstraction? Doing so would avoid recomputing `a` on each call to `f`. On the other hand, `a` is reasonably cheap to recompute and, if `n` is large, the list might take up a lot of heap, which will be alive as long as `f` is alive. It might even turn a constant-space algorithm into a linear-space one, or even worse. We discuss how to deal with this problem next, in Section 5.2.3.

Full laziness and parallelism

When compiling parallel code, full laziness might be an undesirable transformation due to the fact that it increases the sharing of the code.

The following fragment of code was found to run 6 times slower on a 10 processor GRIP ([PCSH87]) parallel machine than the code without full laziness being performed:

```
gen n board = gen' n board
  where gen' :: Int -> Board -> Board
        gen' 0 board = []
        gen' r board = new `par` rest `seq` new : rest
          where new = row' (n-r)
                rest = gen' (r-1)

        row' :: Int -> Row
        row' r = forcelist (row ((shift (copy n 0) board) !! r))
```

The problem in this case was that the expression `(shift (copy n 0) board)` was being floated out of the `row'` function, as it did not depend on `r`, and being shared for all calls to `row'`. But for the parallelism to be fully exploited, each call to `row'` (i.e. each thread) should create its own private copy of the data generated by `(shift (copy n 0) board)`. By sharing the expression a bottleneck is created, as all the processes will depend on a single copy of it.

5.2.3 Reducing the risk of space leaks

Up to now we have discussed `let` floating uniformly, without distinguishing top-level and local `let`s. But this is a very important distinction, because one of the major sources of space leaks when floating `let`s is precisely when we float them to the top level, more specifically when we float constant expressions to the top level. These top level constants are called Constant Applicative Forms (CAFs).

Depending on the strategy of the garbage collector, CAFs may be garbage collectable or not. If they are garbage collectable (as in the Chalmers LML/HBC Compiler) there is no greater risk of floating to the top level than with local `let` floating. But if the garbage collector does not collect CAFs (as is the case in the Glasgow Haskell Compiler), we might not want to float some expressions to the top level, as they may create large data structures that will be kept in the heap during the entire execution of the program, even after it is not needed anymore.

Possible solutions are:

- ✓ Garbage collect CAFs properly, just like "normal" local closures. This is the ideal solution.
- × *Let the garbage collector reverse the updatability of CAFs.* Allow the garbage collector revert CAFs to their unevaluated form, if they start taking up too much

space. This raises problems such as which CAFs to revert to their unevaluated form (some CAFs might be expensive to recompute and should not be reverted).

× *CAFs floated to the top level are set as non-updatable.* This is an extreme version of the previous solution, but again the cost of recomputing the CAF might be too large to make discarding it a reasonable approach.

× *Do not float to top level.* This would stop the floating just before letting the potential CAFs float to the top level. This is too conservative, as could keep some trivial closures (e.g. constants) being reallocated and reevaluated multiple times when they could be allocated once and be shared. We will discuss this option again later in this chapter.

✓ *Only float to top level CAFs that cannot generate a space leak.* This is similar to the previous one, but selects based on static information about closures which are safe to be floated. This includes, for example, expression of types that can only use a (small) bounded amount of heap:

  - data types that are not recursive and do not themselves contain recursive data types as subcomponents, including for example fixed size arrays and all basic types, e.g. `Int`, `Float` or `Char`.
  - Literal constants, e.g. (small) constant lists/strings and all 0-arity constructors (like `[]` (`Nil`)).

  This is the approach currently used in the Glasgow Haskell Compiler.

This same solution can be used to select `let`s to be floated in general, and therefore avoid any risk of space leaks, even when not floating to the top level. We currently only use this strategy for `let`s being floated to the top level.

Although for implementation reasons (which will become clear later) we perform this decision in the full laziness pass, one may argue that this is actually an issue related to local `let` floating. Indeed it is important to notice that the problems of generating CAFs are not restricted to when one floats past lambdas (full laziness), but even with simple `let` floating the problems may arise, e.g.

```
v = let w = [1..100000]
    in last w
```

If we float `w` out it will become a CAF.

5.2.4 Implementing the full laziness transformation

The algorithm we present in this section is an extended version of the algorithm presented in [PL91b]. The extensions deal essentially with the issues of floating to the top level and selection of which maximal free expressions are worth floating. This version also handles type abstraction and application.

The full laziness transformation is done in two passes:

1. the first pass annotates each `let (rec)` binder with its "level number". In general, a level number is the number of lambdas that will enclose the expression after being floated.

2. the second pass uses the level numbers on `let(rec)`s to float each binding outward to *just outside* the lambda which has a level number one greater than that on the binding. We in fact don't leave it *just outside* the lambda when it can be floated to the top level, or when it can go past some type lambdas just outside that lambda, for similar reasons to the ones that lead us to stop floating `let`s inwards sometimes when we hit a type lambda as discussed in Section 5.1.4.

The "set level" algorithm

The basic algorithm for tagging the `let`s with their level numbers ($\mathcal{SL}$, for *Set Level*) is presented in Figure 5.5.

The function $\mathcal{SL}$ is given an expression, the current level (a tuple containing a lambda level and a type-lambda/`case` level[1], initially set to $(0, 0)$) and an environment mapping variables (including type variables) to their level number. The need for the minor level numbers will become clear in Section 5.2.4, but it is essentially related to two issues:

1. floating `let`s to the top level: `case` alternatives can introduce binders, and in expressions of the form

```
f = case E of
      (w:ws) -> E1
      _      -> E2
```

[1]We also sometimes refer to the lambda level as major level number and the type-lambda/`case` level as minor level number.

$$\mathcal{SL}\llbracket\ \rrbracket \ :: Expr \rightarrow Level \rightarrow Env \rightarrow Expr$$

$$(a)\quad \mathcal{SL}\llbracket k\rrbracket\ l\ \rho \quad = \quad \llbracket k\rrbracket$$

$$(b)\quad \mathcal{SL}\llbracket v\rrbracket\ l\ \rho \quad = \quad \llbracket v\rrbracket$$

$$(c)\quad \mathcal{SL}\llbracket C\ v_1\ldots v_n\rrbracket\ l\ \rho \quad = \quad \llbracket C\ v_1\ldots v_n\rrbracket$$

$$(d)\quad \mathcal{SL}\llbracket op\ v_1\ldots v_n\rrbracket\ l\ \rho \quad = \quad \llbracket op\ v_1\ldots v_n\rrbracket$$

$$\begin{aligned}(e)\quad \mathcal{SL}\llbracket \lambda v_1\ldots v_n.E\rrbracket\ (l,t)\ \rho \quad &= \quad \llbracket \lambda v_1\ldots v_n.\mathcal{SL}'\llbracket E\rrbracket\ (l',0)\ \rho'\rrbracket \\ &\quad\ \ \textit{where} \\ &\qquad\ \ l' = l+1 \\ &\qquad\ \ \rho' = \rho\ \oplus\ \{v_i \mapsto (l',0)\}_{i=1}^{n}\end{aligned}$$

$$\begin{aligned}(f)\quad \mathcal{SL}\llbracket \Lambda t_1\ldots t_n.E\rrbracket\ (l,t)\ \rho \quad &= \quad \llbracket \Lambda t_1\ldots t_n.\mathcal{SL}\llbracket E\rrbracket\ (l,t')\ \rho\rrbracket \\ &\quad\ \ \textit{where} \\ &\qquad\ \ t' = t+1 \\ &\qquad\ \ \rho' = \rho\ \oplus\ \{t_i \mapsto (l,t')\}_{i=1}^{n}\end{aligned}$$

$$(g)\quad \mathcal{SL}\llbracket E\ v_1\ldots v_n\rrbracket\ l\ \rho \quad = \quad \llbracket (\mathcal{SL}\llbracket E\rrbracket\ l\ \rho)\ v_1\ldots v_n\rrbracket$$

$$(h)\quad \mathcal{SL}\llbracket E\ T\rrbracket\ l\ \rho \quad = \quad \llbracket (\mathcal{SL}\llbracket E\rrbracket\ l\ \rho)\ T\rrbracket$$

$$\begin{aligned}(i)\quad &\mathcal{SL}\llbracket \texttt{case}\ E\ \texttt{of}\ \{alt_i\ \texttt{->}\ E_i\}_{i=1}^{n}\rrbracket\ (l,t)\ \rho \\ &\quad = \left\llbracket \begin{array}{l}\texttt{case}\ (\mathcal{SL}'\llbracket E\rrbracket\ (l,t)\ \rho)\ \texttt{of} \\ \quad \{alt_i\ \texttt{->}\ \mathcal{SL}'\llbracket E_i\rrbracket\ (l,t')\ \rho_i\}_{i=1}^{n}\end{array}\right\rrbracket \\ &\quad\ \ \textit{where} \\ &\qquad\ \ \rho_i = \rho\ \oplus\ \{v_i \mapsto (l,t') | v_i \leftarrow vars\ alt_i\} \\ &\qquad\ \ t' = t+1\end{aligned}$$

$$\begin{aligned}(j)\quad &\mathcal{SL}\llbracket \texttt{let}\ \{v_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E\rrbracket\ (l,t)\ \rho \\ &\quad = \left\llbracket \begin{array}{l}\texttt{let}\ \{v_{i(l',t')} = \mathcal{SL}\llbracket E_i\rrbracket\ (l',t')\ \rho'\}_{i=1}^{n} \\ \texttt{in}\ \mathcal{SL}'\llbracket E\rrbracket\ (l,t)\ \rho'\end{array}\right\rrbracket \\ &\quad\ \ \textit{where} \\ &\qquad\ \ (l',t') = maxLvl\ \rho\ ((\bigcup\{allfvs\ E_i\}_{i=1}^{n}) \setminus \{v_i\}_{i=1}^{n}) \\ &\qquad\ \ \rho' = \rho\ \oplus\ \{v_i \mapsto (l',t')\}_{i=1}^{n}\end{aligned}$$

$$\begin{aligned}\mathcal{SL}'\llbracket E\rrbracket\ (l,t)\ \rho \quad &= \quad \textit{if}\ (maxMajorLvl\ \rho\ (allfvs\ E) < l) \\ &\qquad\quad \textit{and}\ (not\ isWHNF\ E) \\ &\qquad\ \ \textit{then}\ \llbracket \texttt{let}\ v^{\star}_{(l',t')} = \mathcal{SL}\llbracket E\rrbracket\ (l',t')\ \rho\ \texttt{in}\ v^{\star}\rrbracket \\ &\qquad\ \ \textit{else}\ \llbracket \mathcal{SL}\llbracket E\rrbracket\ (l,t)\ \rho\rrbracket \\ &\quad\ \ \textit{where} \\ &\qquad\ \ (l',t') = maxLvl\ \rho\ (fvars\ E)\end{aligned}$$

Figure 5.5 Algorithm for Setting Levels (Full Laziness) we might want to let `let`s coming from `E1` to be floated to the top level if they can. Therefore we need a way of knowing which of those `let`s can float past the binders introduced by the case alternative, and which ones can't.

2. floating past type abstractions: type abstractions (and type applications) are removed in later stages of the compilation process, as they are used just to keep type information correct during transformations. We do not want, therefore, to stop `let`s from being floated due to type abstractions, and one way of knowing when this might happen is by keeping the level numbers for lambda variables and type variables separate.

We use a single minor number for these two purposes, but one could as well have separate level numbers for type variables and `case` alternative binders.

The important rules in the Set Level algorithm are:

- (e) The lambda level number is incremented at each set of lambdas, and that is the level of those lambda variables.
- (f) The type-lambda/`case` level number is incremented at each set of type lambdas, and that is the level of those type lambda variables.
- (g, h) We could actually try to `let`-bind partial applications if they could be floated further than the full application, but we avoid that as more often than not sharing partial applications is not worthwhile. We discuss this issue again later in this section.
- (i) The type-lambda/`case` level number is incremented at each `case`, and that is the level of variables bound by `case` alternatives.
- (j) The level number of a `let` is the maximum level number of the free variables (including type variables) of the `let` (excluding the variables bound by that `let` itself). The level with which the `let` right hand side is analysed is *the same level* assigned to the `let`. This differs from the algorithm presented in [PL91b], in which the "current" level is used to analyse the `let` right hand side. This could cause some unnecessary floating to occur as the levels of binders in the right hand side would be bigger than they needed to be[2]:

[2] For simplicity we assume we assign different level numbers for each variable in a lambda.

$$
\begin{array}{l}
f \texttt{ = } \lambda a_{(1,0)}\ b_{(2,0)}\ \texttt{.let } v_{(1,0)} \texttt{ =case } a \texttt{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (c\!:\!d) \texttt{ ->let } w_{(1,0)} \texttt{ = } a \texttt{ + } a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{in } w \texttt{ + } c \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{[] -> 5} \\
\qquad\qquad\quad \texttt{in } v \texttt{ + } v
\end{array}
$$

> The question here is what the level numbers should be for the `c` and `d` variables in the first `case` alternative. If we analyse `v`'s right hand side using `v`'s level number, we will assign them level (1,1). When we try to float the binding for `w` out, we will leave it where it is, as we know it will not go past any lambdas if we float it out of the `case` alternative (since its major level number (1) is not less than the `case` alternative binder major level number (1)). But if we analysed the right hand side using the "current" level number, `c` and `d` would be assigned level numbers (2,1), and this would make us think that `w` should be floated out of the `case` alternative, when in fact it would not eventually go past a lambda by doing that.

The ordering used to compute *maxLvl* is as follows:

$$
\begin{array}{ll}
(maj, min) \leq (maj', min')\ \textit{if} & maj \leq maj' \\
& \textit{or}\ (maj = maj'\ \textit{and}\ min \leq min')
\end{array}
$$

The $\mathcal{SL}'$ function is needed so that expressions that can be floated out of a lambda but are not `let` bound are floated. Let us look at an example:

```
f = \x -> case x of
            []     -> g y
            (p:ps) -> ...
```

Here, the subexpression `(g y)` is free in the `\x`-abstraction, and might be an expensive computation which could potentially be shared among all applications of `f`. It is simple enough, in principle, to address this shortcoming, by simply `let`-binding `(g y)` thus:

```
f = \x -> case x of
            []     -> let a = g y
                      in a
            (p:ps) -> ...
```

Now the binding for `a` can by floated out like any other binding. Therefore the $\mathcal{SL}'$ function checks if the expression's lambda level number indicates that it can be floated and, if so, `let`-binds the expression. We actually also avoid `let`-binding if the right hand side is already a weak head normal form expression. If we had used $\mathcal{SL}$ instead, we would only be able to float expressions that were already `let` bound in the original program. A possible alternative would be to always `let`-bind expressions scrutinised by `case`s, `let` bodies and `case` alternative right hand sides, so that every potentially floatable expression would be `let`-bound. But the use of the $\mathcal{SL}'$ function will be useful when we discuss possible variations of this algorithm.

Variations of the algorithm

A few things can be improved in the above algorithm:

- *Abstracting type variables.* Type variables sometimes get in the way and prevent some floating from taking place. For example, suppose `f` and `k` are bound outside the `\x`-abstraction:

  ```
  \x -> /\t -> let v = f t k in ...
  ```

  We would like to float out the `v = f t k`, but we can't, because the type variable `t` would be out of scope. The rules we presented above give `t` the same major level number as `x` which will ensure that the binding isn't floated out of `x`'s scope. Still, there are other particularly painful cases, notably pattern-matching failure bindings, such as:

  ```
  ... let fail = error ty "Pattern fail" in ...
  ```

  We really would like this to get lifted to the top level, despite its free type variable `ty`. There are two approaches: ignore the problem of out-of-scope type variables, or fix it up somehow. We take the latter approach, using the following procedure. If a binding `v = e` has free type variables whose maximum level number is *strictly greater* than the one we would get by using only the ordinary variables (that is without the type variables), then we abstract over the offending type variables that are keeping it from floating further, `t1..tn`, thus:

  ```
  v = let v' = /\t1..tn -> e
      in v' t1 ... tn
  ```

  Now `v` is given the usual level number (taking type variables into account), while `v'` is given the maximum level number of the ordinary free variables only (since the type variables `t1..tn` are not free in `v'`). So `v'` can be floated, leaving a type application in its place.

This modification is achieved by modifying $\mathcal{SL}'$ and the (j) rule to detect the conditions and abstract the type variables when necessary.

The reason our strategy is not perfect is that some subsequent binding might mention `v`. In theory it too could be floated out, but it will get pinned inside the binding for `v` (it is the binding for `v'` which floats). But we believe our strategy catches the common cases.

- *Split lambdas.* Another possible variation of the algorithm is to assign different level numbers to each lambda. This would allow `lets` to be floated to places in between lambdas that were originally together, e.g.

```
f = \a -> \b -> let v = a + 1 in b + v
```

would become

```
f = \a -> let v = a + 1 in \b -> in b + v
```

This would be helpful only in the cases where `f` was partially applied to one argument *and* the partial application was shared in applications to a second argument. If this was the case, `v` would be allocated and evaluated only once and shared by the other applications. But if this was not the case, then the second form will be less efficient as it will trigger multiple argument satisfaction checks, among other reasons. This modification can be done by a simple modification to the (e) rule. We have measured the effect of splitting lambdas in our benchmark programs and indeed it has major negative effect on performance when compared with our non-splitting version, as shown in Table 5.3, therefore we do not split lambdas.

- `let`-*bind partial applications to be floated out.* This would allow partial applications to be `let` bound if they can be floated out, e.g.:

```
f = \a -> let v = a + 1 in \b -> g v b
```

would become

```
f = \a -> let v = a + 1; w = g v in \b -> w b
```

Full Laziness Total Instructions Executed		
program	normal	split λ's
`multiplier`	1.00	0.93
`gg`	1.00	0.95
`boyer`	1.00	1.01
`clausify`	1.00	1.01
`event`	1.00	1.01
`veritas`	1.00	1.01
`ida`	1.00	1.02
`transform`	1.00	1.02
`fluid`	1.00	1.03
`mandel2`	1.00	1.03
`lift`	1.00	1.04
`maillist`	1.00	1.04
`mandel`	1.00	1.04
`treejoin`	1.00	1.04
`fft`	1.00	1.05
`prolog`	1.00	1.05
`comp_lab_zift`	1.00	1.06
`rewrite`	1.00	1.06
`hpg`	1.00	1.08
`solid`	1.00	1.08
`genfft`	1.00	1.10
`hidden`	1.00	1.11
`typecheck`	1.00	1.11
`parser`	1.00	1.13
`reptile`	1.00	1.13
`primes`	1.00	1.14
`sched`	1.00	1.17
`boyer2`	1.00	1.19
`wave4main`	1.00	1.31
17 other programs	1.00	1.00
Minimum	-	0.93
Maximum	-	1.31
Geometric mean	-	1.04

Full Laziness Total Heap Allocated		
program	normal	split λ's
`multiplier`	1.00	0.93
`minimax`	1.00	0.99
`boyer`	1.00	1.01
`fft2`	1.00	1.01
`gg`	1.00	1.01
`listcompr`	1.00	1.01
`listcopy`	1.00	1.01
`veritas`	1.00	1.01
`clausify`	1.00	1.02
`infer`	1.00	1.02
`pretty`	1.00	1.02
`event`	1.00	1.03
`mandel`	1.00	1.03
`parstof`	1.00	1.03
`ida`	1.00	1.05
`primetest`	1.00	1.06
`transform`	1.00	1.06
`knights`	1.00	1.07
`rsa`	1.00	1.08
`fft`	1.00	1.09
`mandel2`	1.00	1.10
`prolog`	1.00	1.11
`comp_lab_zift`	1.00	1.12
`rewrite`	1.00	1.15
`fluid`	1.00	1.17
`maillist`	1.00	1.18
`genfft`	1.00	1.19
`lift`	1.00	1.23
`hpg`	1.00	1.27
`reptile`	1.00	1.28
`hidden`	1.00	1.36
`solid`	1.00	1.43
`treejoin`	1.00	1.49
`sched`	1.00	1.56
`typecheck`	1.00	1.57
`primes`	1.00	1.65
`boyer2`	1.00	2.30
`parser`	1.00	2.71
`wave4main`	1.00	2.86
7 other programs	1.00	1.00
Minimum	-	0.93
Maximum	-	2.86
Geometric mean	-	1.18

Table 5.3 Full Laziness (splitting λ's): instructions executed and bytes allocated

This is only useful if `g` actually will perform any work when given one argument *and* `f` is partially applied *and* this partial application is shared in applications to a second argument. Again the second form is less efficient if these conditions are not met. We believe this is often the case, therefore we do not implement this modification. It could be implemented by modifying the (g) rule to use $\mathcal{SL}'$ on partial applications of the expression to some of the arguments.

- We are very careful about giving bindings a level number $(0, x)$, because that will mean they will be floated out of all enclosing lambdas, and possibly create a space leak, even if they don't get to the top level, e.g.

```
f = \ g -> let x = [1..1000]     ==> f = let x = [1..1000]
           in map g x                      in \ g -> map g x
```

is just as bad as floating `x` to the top level (assuming `f` is a top level function) as it will never be garbage collected. Options of what we could do with `let`s that would normally get a level number $(0, x)$ are directly related to the problems (and solutions) we discussed in Section 5.2.3. We proceed with the following algorithm for such `let`s:

  - if the `let` cannot create a space leak (according to the criteria presented in Section 5.2.3), we will give it a level number $(0, x)$, allowing it to be floated past all lambdas (x might be greater than 0 due to `case` alternative variables for example). If the level is actually $(0, 0)$ we will allow it to be floated to the top level. This can be achieved in the algorithm we presented by giving it a level `Top`, which is defined as lower than $(0, 0)$. This will allow the binding to go past top level `let`s.
  - if it may create a space leak we choose one of the following options, which are ordered from the safest to the most risky:
    - (a) we give it a level number $(1, 0)$, so that it won't go past the outermost lambda, unless of course its current major level is already less than 1, in which case we leave it where it is.
    - (b) we leave it with its $(0, x)$ level.
    - (c) we treat it just like the non-leaky `let`s, that is, we allow it to be floated to the top level if its level is $(0, 0)$.

We performed measurements with the three options above, as shown in Tables 5.4 and 5.5. Option (c) indeed caused one major space leak, and therefore we currently use option (b) in the Glasgow Haskell Compiler.

Full Laziness Strategy Total Instructions Executed			
program	(a)	(b)	(c)
`solid`	1.00	0.85	0.85
`minimax`	1.00	0.94	0.92
`cichelli`	1.00	0.95	0.94
`lift`	1.00	0.98	0.97
`boyer`	1.00	1.00	1.01
`parstof`	1.00	1.00	0.88
`rewrite`	1.00	1.00	0.95
`parser`	1.00	1.03	0.90
`gen_regexps`	1.00	1.56	1.00
37 other programs	1.00	1.00	1.00
Minimum	-	0.85	0.85
Maximum	-	1.56	1.01
Geometric mean	-	1.00	0.99

Full Laziness Strategy Total Heap Allocated			
program	(a)	(b)	(c)
`cichelli`	1.00	0.82	0.82
`solid`	1.00	0.83	0.83
`minimax`	1.00	0.90	0.89
`knights`	1.00	0.96	0.96
`lift`	1.00	0.96	0.96
`rewrite`	1.00	0.98	0.98
`gg`	1.00	0.99	1.00
`veritas`	1.00	1.00	0.99
`parser`	1.00	1.03	0.97
`gen_regexps`	1.00	1.32	0.89
36 other programs	1.00	1.00	1.00
Minimum	-	0.82	0.82
Maximum	-	1.32	1.00
Geometric mean	-	0.99	0.98

Table 5.4 Full laziness strategy: instructions executed and bytes allocated

Notice that these numbers are relative to a program that already had full laziness applied to it (a), therefore even `gen_regexps` in the (b) column (which is far worse than with the other setups) still shows an improvement over the version without full laziness, as we will see later in this chapter. The reason for the odd behaviour of `gen_regexps` is due to a `let`-binding being left just inside a function definition, and thus avoiding other transformations (e.g. worker-wrapper transformations) from occurring:

(a)
```
f = \x -> let v = [1..100]
          in ...
```

(b)
```
f = let v = [1..100]
    in \x -> ...
```

(c)
```
v = [1..100]
f = \x -> ...
```

The float out algorithm

The float out algorithm is presented in Figure 5.6.

It receives an annotated expression and a level number l and returns an expression paired with a list of bindings that are being floated outwards. We then drop them just outside the lambda with their level number (unless it can go past enclosing type lambdas just outside that lambda, in which case we allow it to float past them). The

Full Laziness Strategy Residency									
	(a)			(b)			(c)		
program	GCs	bytes	ratio	GCs	bytes	ratio	GCs	bytes	ratio
`cichelli`	37	1,345,424	1.00	30	1,333,408	0.99	30	1,334,880	0.99
`clausify`	20	39,952	1.00	20	39,952	1.00	20	40,516	1.01
`event`	44	4,010,772	1.00	44	4,010,772	1.00	44	4,155,820	1.04
`exp3_8`	98	26,660	1.00	98	26,660	1.00	98	51,088	1.92
`fft2`	25	871,364	1.00	25	874,764	1.00	25	967,456	1.11
`mandel2`	10	468	1.00	10	468	1.00	10	79,056	168.92
`parstof`	47	562,276	1.00	47	562,276	1.00	47	554,648	0.99
`sched`	21	2,204	1.00	21	2,204	1.00	21	2,180	0.99
`typecheck`	131	10,596	1.00	131	10,596	1.00	131	15,940	1.50
`genfft`	21	3,464	1.00	21	3,496	1.01	21	3,440	0.99
`mandel`	220	12,648	1.00	220	12,820	1.01	220	12,624	1.00
`parser`	11	866,040	1.00	12	872,480	1.01	11	857,132	0.99
`boyer`	21	95,512	1.00	21	97,240	1.02	21	97,096	1.02
`infer`	10	1,978,136	1.00	10	2,010,228	1.02	10	2,012,396	1.02
`ida`	51	380,468	1.00	51	391,356	1.03	51	391,316	1.03
`rewrite`	21	17,208	1.00	21	17,700	1.03	21	31,000	1.80
`gg`	7	355,160	1.00	7	375,264	1.06	7	375,400	1.06
`hpg`	61	569,444	1.00	61	610,432	1.07	61	610,624	1.07
`multiplier`	85	1,662,412	1.00	85	1,813,728	1.09	85	1,866,436	1.12
`rsa`	31	3,676	1.00	31	4,148	1.13	31	3,744	1.02
14 other progs.	-	-	1.00	-	-	1.00	-	-	1.00
Minimum	-	-	1.00	-	-	0.99	-	-	0.99
Maximum	-	-	1.00	-	-	1.13	-	-	168.92
Geom. mean	-	-	1.00	-	-	1.01	-	-	1.24

Table 5.5 Full Laziness Strategy: Residency

$$
\begin{array}{lll}
& \mathcal{FL}[\![\ ]\!] \ :: Expr \rightarrow Level \rightarrow (Expr, [Binding]) & \\
(a) & \mathcal{FL}[\![k]\!] \ l & = ([\![k]\!], \emptyset) \\
(b) & \mathcal{FL}[\![v]\!] \ l & = ([\![v]\!], \emptyset) \\
(c) & \mathcal{FL}[\![C \ v_1 \ldots v_n]\!] \ l & = ([\![C \ v_1 \ldots v_n]\!], \emptyset) \\
(d) & \mathcal{FL}[\![op \ v_1 \ldots v_n]\!] \ l & = ([\![op \ v_1 \ldots v_n]\!], \emptyset) \\
(e) & \mathcal{FL}[\![\lambda v.E]\!] \ (l,t) & = ([\![\lambda v.\texttt{let} \ \rho_h \ \texttt{in} \ E']\!], \rho') \\
& & \quad \textit{where} \\
& & \quad\quad (E', \rho) = \mathcal{FL}[\![E]\!] \ (l', 0) \\
& & \quad\quad l' = l + 1 \\
& & \quad\quad (\rho_h, \rho') = \textit{partitionByMajorLvl} \ \rho \ (l', 0) \\
(f) & \mathcal{FL}[\![\Lambda t_1 \ldots t_n.E]\!] \ (l,t) & = ([\![\Lambda t_1 \ldots t_n.\texttt{let} \ \rho_h \ \texttt{in} \ E']\!], \rho') \\
& & \quad \textit{where} \\
& & \quad\quad (E', \rho) = \mathcal{FL}[\![E]\!] \ (l, t') \\
& & \quad\quad t' = t + 1 \\
& & \quad\quad (\rho_h, \rho') = \textit{partitionByMinorLvl} \ \rho \ (l, t') \\
(g) & \mathcal{FL}[\![E \ v_1 \ldots v_n]\!] \ l & = ([\![E' \ v_1 \ldots v_n]\!], \rho) \\
& & \quad \textit{where} \\
& & \quad\quad (E', \rho) = \mathcal{FL}[\![E]\!] \ l \\
(h) & \mathcal{FL}[\![E \ T]\!] \ l & = ([\![E' \ T]\!], \rho) \\
& & \quad \textit{where} \\
& & \quad\quad (E', \rho) = \mathcal{FL}[\![E]\!] \ l \\
(i) & \mathcal{FL}[\![\texttt{case} \ E \ \texttt{of} \ \{alt_i \ \texttt{->} \ E_i\}_{i=1}^n]\!] \ (l,t) & \\
& & = ([\![\texttt{case} \ E' \ \texttt{of} \ \{alt_i \ \texttt{->} \ \texttt{let} \ \rho_i'' \ \texttt{in} \ E_i\}_{i=1}^n]\!], \rho) \\
& & \quad \textit{where} \\
& & \quad\quad (E', \rho') = \mathcal{FL}[\![E]\!] \ (l, t) \\
& & \quad\quad (E_i, \rho_i) = \mathcal{FL}[\![E_i]\!] \ (l, t') \\
& & \quad\quad (\rho_i'', \rho_i') = \textit{partitionByMajorLvl} \ \rho_i \ (l, t') \\
& & \quad\quad t' = t + 1 \\
& & \quad\quad \rho = \rho' \cup \bigcup\{\rho_i'\}_{i=1}^n
\end{array}
$$

Figure 5.6 Algorithm for Floating `lets` Out (Full Laziness)

$$
\begin{array}{l}
(j)\quad \mathcal{FL}[\![\texttt{let}\ \{v_{i(l',t')} = E_i\}_{i=1}^{n}\ \texttt{in}\ E]\!]\ (l,t)\ \rho \\
\qquad = \ \ \mathit{if}\ l' < l \\
\qquad\qquad \mathit{then}\ ([\![E']\!], \rho' \cup \{v_i \mapsto \texttt{let}\ \rho_i''\ \texttt{in}\ E_i\}_{i=1}^{n} \cup \bigcup\{\rho_i'\}_{i=1}^{n}) \\
\qquad\qquad \mathit{else}\ ([\![\texttt{let}\ \{v_i = \texttt{let}\ \rho_i''\ \texttt{in}\ E_i'\}_{i=1}^{n}\ \texttt{in}\ E']\!], \rho' \cup \bigcup\{\rho_i'\}_{i=1}^{n}) \\
\qquad\qquad \mathit{where} \\
\qquad\qquad\quad (E', \rho') = \mathcal{FL}[\![E]\!]\ (l,t) \\
\qquad\qquad\quad (E_i', \rho_i) = \mathcal{FL}[\![E_i]\!]\ (l',t') \\
\qquad\qquad\quad (\rho_i'', \rho_i') = \mathit{partitionByMajorLvl}\ \rho_i\ (l',t')
\end{array}
$$

Figure 5.7 Algorithm for Floating `lets` Out (Full Laziness)

function *partitionByLvl* splits the list of bindings in two sets, one for bindings that should be dropped immediately (cannot go any further) and the other for bindings that are to be floated out further.

A binding is floated out just far enough to escape all the lambdas which it can escape, and no further. This is consistent with the idea that bindings should be as far in as possible (floating inwards, Section 5.1). In the actual implementation there is one exception to this: bindings with level `Top` are floated right to the top level. This is also a difference between this algorithm and the one presented in [PL91b], which implicitly *always* floats `lets` out of `lets`, since it does no partitioning when a `let` is reached.

A binding is not moved at all unless it will definitely escape a lambda.

5.2.5 Floating inwards and full laziness

One might think that if we know we will perform the full laziness transformation after floating inwards, we could relax the restriction of not floating inwards past a lambda, since the expression could be floated outwards again by the full laziness transformation.

This is not true, and actually very risky! After the expression is inside the lambda, it may be simplified and then become impossible to be pulled out again. Let us follow an example to show how this happens:

```
let v = case w of
         I w# -> fib w#
in let f = \x -> case v of
                  I v# -> case x of
```

```
                              I x# -> case v# +# x# of
                                         r# -> r#
   in (f,f)
```

If `v` is floated into `f` (and its lambda) it will be used strictly, and therefore a `let` to `case` and other transformations can take place. After that the code will become

```
let f = \x -> case w of
               I# w# -> case fib w# of
                          v -> case v of
                                I# v# -> case x of
                                          I# x# -> case v# +# x# of
                                                     r# -> r#
in (f,f)
```

Now the expression `fib w#` cannot be floated out of the lambda because `w#` is bound inside the lambda. Therefore `fib w#` will be computed as many time as the lambda expression is entered!

Actually, as we will see in Section 5.3, the `case` scrutinising `w` could be floated out of the lambda, but there are cases when this is not possible (e.g. if there was a multi-branch case between the lambda and the `case` we want to float out).

This same discussion applies to inlining inside lambdas, that is, we cannot inline arbitrary expressions inside lambdas (as discussed in Chapter 6) relying on full laziness to undo the work (if necessary), because expressions might become (due to other transformations) impossible to be taken back out of the lambda.

5.2.6 Results

We can measure the overall effect of full laziness in many different ways, but we will concentrate on its effect on total heap allocated, number of updates, residency and number of instructions executed. We expect the heap allocated, instructions executed and number of updates to show improvements due to the increase in sharing, and we hope that the residency is not increased significantly.

In Figure 5.6 we have the overall effect on the total heap allocated and instructions executed. The first column has full laziness turned off, the second one only floats bindings that we select as "non-leaky" (not only to the top level, but also in a local context), and the third column presents our normal full laziness setup, which only floats to the top level "non-leaky" bindings. We can see that some of the programs are significantly affected by full laziness, sometimes allocating 3 times more heap and running in twice the time if full laziness is turned off. The average improvement was 13% on heap allocation and 8% on instructions executed, which is a surprisingly good result.

The effect on heap residency (Figure 5.7) has some mixed results, with some significant increase on some of the programs, although some of the programs have a very small residency and therefore can easily be affected by any transformation.

It was quite surprising to find that the residency can actually be reduced by the full laziness transformation. This can be explained by the following example:

```
let l = [1..100000]
in let f = \a -> let n = length l
                 in a + n
   in ...
```

after full laziness becomes

```
let l = [1..100000]
in let n = length l
   in let f = \a -> a + n
      in ...
```

In the first expression `l` is alive until `f` can be garbage collected, while in the second one it can be garbage collected after `f` is evaluated for the first time (that is, after `n` is evaluated).

5.2.7 Conclusion

Again many programs are unaffected by the transformation, but the few that are affected show a significant improvement. The impact on compile time is again negligible.

We believe these results justify the presence of full laziness in optimising compilers in at least two forms:

- always performed when the `let`s cannot create a space leak;

Full Laziness Total Instructions Executed			
program	off	safe	on
mandel2	1.00	0.48	0.48
fft2	1.00	0.88	0.50
queens	1.00	0.99	0.56
hidden	1.00	0.58	0.58
sched	1.00	0.77	0.77
solid	1.00	1.01	0.86
minimax	1.00	0.98	0.92
boyer	1.00	0.94	0.94
cichelli	1.00	0.99	0.94
fft	1.00	0.94	0.94
gen_regexps	1.00	1.00	0.94
parser	1.00	0.92	0.95
mandel	1.00	0.98	0.96
clausify	1.00	0.97	0.97
genfft	1.00	0.97	0.97
maillist	1.00	0.97	0.97
reptile	1.00	0.97	0.97
hpg	1.00	0.99	0.98
lift	1.00	1.00	0.98
typecheck	1.00	0.98	0.98
fluid	1.00	1.00	0.99
gg	1.00	0.99	0.99
infer	1.00	0.99	0.99
knights	1.00	1.00	0.99
rewrite	1.00	0.99	0.99
wang	1.00	0.99	0.99
wave4main	1.00	0.99	0.99
comp_lab_zift	1.00	1.01	1.01
event	1.00	1.01	1.01
pretty	1.00	1.01	1.01
16 other programs	1.00	1.00	1.00
Minimum	-	0.48	0.48
Maximum	-	1.01	1.01
Geometric mean	-	0.96	0.92

Full Laziness Total Heap Allocated			
program	off	safe	on
fft2	1.00	0.73	0.28
mandel2	1.00	0.30	0.30
queens	1.00	1.01	0.39
hidden	1.00	0.57	0.60
boyer	1.00	0.67	0.67
sched	1.00	0.78	0.78
cichelli	1.00	1.00	0.81
solid	1.00	0.98	0.81
gen_regexps	1.00	1.00	0.82
knights	1.00	0.90	0.87
minimax	1.00	0.98	0.88
fft	1.00	0.89	0.89
gg	1.00	0.91	0.89
maillist	1.00	0.91	0.91
lift	1.00	0.96	0.92
rewrite	1.00	0.95	0.93
hpg	1.00	0.96	0.94
parser	1.00	0.91	0.94
fluid	1.00	0.95	0.95
typecheck	1.00	0.95	0.95
reptile	1.00	0.96	0.96
boyer2	1.00	0.97	0.97
clausify	1.00	0.97	0.97
genfft	1.00	0.97	0.97
listcompr	1.00	0.97	0.97
listcopy	1.00	0.97	0.97
wave4main	1.00	0.97	0.97
mandel	1.00	0.99	0.98
parstof	1.00	0.98	0.98
multiplier	1.00	0.99	0.99
pretty	1.00	0.99	0.99
prolog	1.00	0.99	0.99
veritas	1.00	0.99	0.99
comp_lab_zift	1.00	1.03	1.03
transform	1.00	1.03	1.03
11 other programs	1.00	1.00	1.00
Minimum	-	0.30	0.28
Maximum	-	1.03	1.03
Geometric mean	-	0.92	0.87

Table 5.6 Full Laziness: instructions executed and bytes allocated

Full Laziness Residency									
	off			safe			on		
program	GCs	bytes	ratio	GCs	bytes	ratio	GCs	bytes	ratio
clausify	21	106,956	1.00	20	39,952	0.37	20	39,952	0.37
boyer	32	162,524	1.00	21	93,232	0.57	21	97,240	0.60
sched	26	2,932	1.00	21	2,204	0.75	21	2,204	0.75
mandel2	34	608	1.00	10	468	0.77	10	468	0.77
queens	23	1,204	1.00	23	1,280	1.06	9	992	0.82
typecheck	138	11,592	1.00	131	10,596	0.91	131	10,596	0.91
fft	41	1,868,252	1.00	36	1,722,508	0.92	36	1,722,536	0.92
fft2	86	909,320	1.00	63	898,872	0.99	25	874,764	0.96
parser	13	900,264	1.00	11	866,040	0.96	12	872,480	0.97
transform	201	146,744	1.00	206	142,792	0.97	206	142,856	0.97
compress	146	169,288	1.00	146	167,084	0.99	146	167,084	0.99
listcompr	74	7,505,292	1.00	71	7,434,184	0.99	71	7,434,184	0.99
cichelli	37	1,333,724	1.00	37	1,345,424	1.01	30	1,333,408	1.00
genfft	22	3,508	1.00	21	3,464	0.99	21	3,496	1.00
parstof	48	555,436	1.00	47	562,276	1.01	47	562,276	1.01
comp_lab_zift	107	1,208,156	1.00	111	1,228,620	1.02	111	1,228,664	1.02
ida	51	379,380	1.00	51	380,372	1.00	51	391,356	1.03
infer	10	1,959,180	1.00	10	1,978,836	1.01	10	2,010,228	1.03
hpg	65	578,816	1.00	62	569,060	0.98	61	610,432	1.05
rewrite	23	16,864	1.00	21	17,216	1.02	21	17,700	1.05
gg	8	347,044	1.00	7	354,496	1.02	7	375,264	1.08
multiplier	86	1,603,752	1.00	85	1,662,420	1.04	85	1,813,728	1.13
rsa	31	3,172	1.00	31	3,744	1.18	31	4,148	1.31
mandel	224	6,104	1.00	222	6,220	1.02	220	12,820	2.10
10 other progs.	-	-	1.00	-	-	1.00	-	-	1.00
Minimum	-	-	-	-	-	0.37	-	-	0.37
Maximum	-	-	-	-	-	1.18	-	-	2.10
Geom. Mean	-	-	-	-	-	0.94	-	-	0.97

Table 5.7 Full Laziness: Residency

- as an (optional) compiler optimisation for possibly leaky `let`s.

As we presented, even the second option is reasonably safe, specially when floating `let`s out in a local context, i.e. not to the top level.

5.3 Floating cases out of lambdas

Suppose we have the following function definition:

```
f = \x -> \y -> case z of
                  1 -> (x,y)
                  _ -> case y of
                         1 -> (y,x)
                         _ -> f (x+y) (y-1)
```

where `z` is a free variable. Since the `case` is scrutinising a variable that is not bound by the enclosing lambdas it could possibly be floated out past the lambdas, and we would get the following definition:

```
f = case z of
      1 -> \x -> \y -> (x,y)
      _ -> \x -> \y -> case y of
                         1 -> (y,x)
                         _ -> f (x+y) (y-1)
```

this particular change has the following effect:

× `f` is now an updatable closure – previously it was not, as it was a weak head normal form.

× as the lambdas are further down the expression we miss some optimisations that are based on the arity information, e.g. worker-wrapper transformations.

✓ `z` will only be scrutinised once, while in the original definition it was evaluated every time f was entered.

We are actually interested in the benefit of not rescrutinising `z` after the transformation. If `f` is entered many times this might save quite a lot, even taking into account the two disadvantages.

Floating `case`s out of lambdas achieves a similar effect to full laziness, by allowing the possibility of sharing the evaluation of the scrutinee.

One should notice that the expression has a slightly different semantics after the transformation. If the value of the scrutinee is bottom, before the transformation the expression is isomorphic to $\lambda x.\bot$ and after it is isomorphic to $\bot$. This difference is not relevant for practical or even theoretical purposes as it will only affect the semantic value of a program that fails. The language itself never distinguishes between $\lambda x.\bot$ and $\bot$.

Sometimes there is no disadvantage in performing this transformation: if `f` was a local definition and was used strictly (demanded), we would be able now to float the case further out (using the `case` floating from `let` transformation) (Section 3.5.3), and therefore eliminate the disadvantages above. In the following example the resulting expression is certainly better than the original one:

```
f = \ a -> let rec g = \ c -> case a of
                                (e,f) -> let v = c - 1 ;
                                             w = g v
                                         in e + w
            in g 100
```

$\Longrightarrow$

```
f = \ a -> case a of
             (e,f) -> let rec g = \ c -> let v = c - 1 ;
                                             w = g v
                                         in e + w
                      in g 100
```

The idea of floating `case`s past a lambda is similar to sharing the evaluation of control constructs presented in [Hol90], though we believe that expressing it as floating the control construct itself (in our case the `case` constructor) is simpler and more elegant.

Although [Hol90] presents examples in which this transformation can provide substantial improvements, in our experiments this transformation did not substantially improve any of the benchmark programs, as shown in Table 5.8. We currently do not perform this transformation in the Glasgow Haskell Compiler.

`case` Floating Total Instructions Executed		
program	off	on
`parser`	1.00	0.99
45 other programs	1.00	1.00
Minimum	-	0.99
Maximum	-	1.00
Geometric mean	-	1.00

`case` Floating Total Heap Allocated		
program	off	on
`boyer2`	1.00	0.98
`prolog`	1.00	1.02
44 other programs	1.00	1.00
Minimum	1.00	0.98
Maximum	1.00	1.02
Geometric mean	1.00	1.00

Table 5.8 `case` floating: instructions executed and bytes allocated

5.4 Ordering the `let` floating transformations

At first one might think that the ordering in which the transformations are applied is irrelevant, as each one of them is achieving different objectives. Actually this is not true, as some transformations may expose opportunities for other transformations, and therefore should be done before them. In other cases they may actually hide these opportunities, and therefore should be done after them.

In this section we present some of the issues that lead us to choose a specific sequence for performing the `let` floating transformations. We cannot be 100% sure this is the best possible order, but it was obtained by close inspection of the code of the benchmark programs.

Basically the ordering of the transformations has to follow a set of constraints, which are described in the next sections.

5.4.1 Float inwards before strictness analysis

Floating inwards moves definitions inwards to a site at which a binding might become strict, as presented in Section 5.1.1.

5.4.2 Full laziness after strictness analysis

When generating worker-wrapper pairs it may be the case that an argument is not used by the worker, e.g. in

```
\z -> let x = f (a,z) in ...
```

it might be the case that f actually only needs a, and therefore after a worker/wrapper pair is generated we get

```
===> (absence analysis + inline wrapper of f)
                \z -> let x = f.wrk a in ...
===> (full laziness)
                let x= f.wrk a in  \z -> ...
```

and as we can see f can now be floated past the enclosing lambda. Therefore strictness analysis (actually absence analysis) may allow something to be floated out which would not otherwise be.

Another possibility is that inlining exposes some extra opportunities for the full laziness transformation, for example:

```
                f = \z -> let x = g z 20 in ...
                g = \a -> \b -> fib b + fib a
===> (inlining)
                f = \z -> let x = fib 20 + fib z in ...
```

At this point we could float fib 20 to the top level.

As we will see there are also reasons to perform full laziness very early in the compilation process. We performed experiments in which we run full laziness twice, first early in the compilation process and later again. Experimental evidence suggests that the cases described above actually do not happen very often, and therefore running full laziness twice does not improve the code in the great majority of the programs, as shown in Table 5.9. The Glasgow Haskell Compiler currently does not run full laziness twice.

5.4.3 Simplify after floating inwards

This is due to the following (that happens with dictionaries):

```
let a1 = case v of (a,b) -> a
in let m1 = \ c -> case c of I# c# -> case c# of 1 -> a1 5
                                                 2 -> 6
in let m2 = \ c -> case c of I# c# ->
                     case c# +# 1# of cc# -> let cc = I# cc#
                                             in m1 cc
   in (m1,m2)
```

Full Laziness Total Instructions Executed		
program	once	twice
`genfft`	1.00	0.99
`lift`	1.00	0.99
`mandel`	1.00	0.99
`parstof`	1.00	0.99
`sorting`	1.00	0.99
`hpg`	1.00	1.01
`mandel2`	1.00	1.01
39 other programs	1.00	1.00
Minimum	-	0.99
Maximum	-	1.01
Geometric mean	-	1.00

Full Laziness Total Heap Allocated		
program	once	twice
`genfft`	1.00	0.98
`boyer`	1.00	0.99
`lift`	1.00	0.99
`fluid`	1.00	1.01
`gg`	1.00	1.01
`hpg`	1.00	1.02
`parser`	1.00	1.02
`mandel2`	1.00	1.04
38 other programs	1.00	1.00
Minimum	1.00	0.98
Maximum	1.00	1.04
Geometric mean	1.00	1.00

Table 5.9 Full Laziness twice: instructions executed and bytes allocated

floating inwards will push the definition of `a1` into `m1` (supposing it is only used there):

```
in let m1 = let a1 = case v of (a,b) -> a
            in \ c -> case c of I# c# -> case c# of 1 -> a1 5
                                                    2 -> 6
in let m2 = \ c -> case c of I# c# ->
                     case c# +# 1# of cc# -> let cc = I# cc#
                                              in m1 cc
   in (m1,m2)
```

if we do strictness analysis now we will not get a worker-wrapper for `m1`, because of the `let` for `a1`.

Not having this worker wrapper might be very bad, because it might mean that we will have to rebox arguments to `m1` if they are already unboxed, generating extra allocations, as occurs when it is called from `m2` (`cc`) above.

To solve this problem we run the simplifier after floating inwards, so that `let`s whose body is a weak head normal form are floated out, undoing the floating inwards transformation in these cases. We are then back to the original code, which would have a worker-wrapper for `m1` after strictness analysis and would avoid the extra `let` in `m2`.

What we lose in this case are the opportunities for `let` to `case` (or `case` floating) that could be presented if, for example, `a1` would be demanded (strict) after the floating inwards.

The only way of having the best of both is if we make the worker-wrapper pass explicit, and then we could do with:

1 – float-in
2 – strictness analysis
3 – simplify
4 – strictness analysis
5 – worker-wrapper generation

as we would:

- be able to detect the strictness of `a1` after the first call to the strictness analyser, and exploit it with the simplifier (in case it was strict);
- after the call to the simplifier (if `a1` was not demanded) it would be floated out just like we currently do, before strictness analysis II and worker-wrapperisation.

We currently simplify after floating inwards.

5.4.4 Float inwards again after strictness analysis

When workers are generated after strictness analysis (worker-wrapper), we generate them with "reboxing" lets, that simply rebox the unboxed arguments, as it may be the case that the worker will need the original boxed value:

```
f x y = case x of
          (a,b) -> case y of
                     (c,d) -> case a == c of
                                True -> (x,x)
                                False -> ((1,1),(2,2))

==> (worker/wrapper)

f x y = case x of
          (a,b) -> case y of
                     (c,d) -> f.wrk a b c d

f.wrk a b c d = let x = (a,b)
                    y = (c,d)
                in case a == c of
```

```
                    True -> (x,x)
                    False -> ((1,1),(2,2))
```

in this case the simplifier will remove the binding for `y`, since it is not used (we expected this to happen very often, but we do not know how many "reboxers" are eventually removed and how many are kept), and will keep the binding for `x`. But `x` is only used in *one* of the branches in the `case`, but is always being allocated! The floating inwards pass would push its definition into the `True` branch. A similar benefit occurs if it is only used inside a `let` definition. These are basically the advantages of floating inwards, but they are only exposed after the Strictness Analysis/worker-wrapperisation of the code! As we also have reasons to float inwards before Strictness Analysis, we have to run it twice.

Another compelling example of the need to float inwards again after strictness analysis is the following:

```
f = \ a -> let x = case a of
                     (c,d) -> c ;
               y = case a of
                     (w,z) -> z
           in case y of
                0 -> (x,y)
                n -> (y,x)
==>
f = \ a -> let x = case a of
                     (c,d) -> c
           in case a of
                (w,z) -> case z of
                           0 -> (x,z)
                           n -> (z,x)
```

`y` is demanded, therefore we can float the `case` out and do other simplifications. But we are still left with the closure for `x` (if the order of `x` and `y`'s definition was swapped we would not have this problem!). But if we now float `x` definition into the first `case` it will be simplified by the `case` reduction transformation, as we would expect.

In Table 5.10 we see the effect that floating inwards twice, as opposed to floating inwards once (early), has on our benchmark programs.

Float In twice Total Instructions Executed		
	Float In	
program	once	twice
`wave4main`	1.00	0.91
`mandel2`	1.00	0.97
`treejoin`	1.00	0.97
`fft`	1.00	0.99
`fluid`	1.00	0.99
`maillist`	1.00	0.99
40 other programs	1.00	1.00
Minimum	-	0.91
Maximum	-	1.00
Geometric mean	-	1.00

Float In twice Total Heap Allocated		
	Float In	
program	once	twice
`wave4main`	1.00	0.50
`treejoin`	1.00	0.79
`maillist`	1.00	0.91
`fft`	1.00	0.97
`knights`	1.00	0.97
`clausify`	1.00	0.98
`fluid`	1.00	0.98
`hpg`	1.00	0.98
`compress`	1.00	0.99
`prolog`	1.00	0.99
Minimum	-	0.50
-Maximum	-	1.00
Geometric mean	-	0.97

Table 5.10 Float In twice: instructions executed and bytes allocated

5.4.5 Full laziness before any inlining

When experimenting with more aggressive inlining strategies (Chapter 6), we found that sometimes if inlining is performed before full laziness some opportunities for full laziness may be lost. This is related to the same issues we discussed in Section 5.2.5, in which we explain that we cannot rely on full laziness to float `let`s out again if we allow `let`s to be floated into lambdas. The same may happen due to inlining, as some expressions may end up with unboxed types, which we cannot `let`-bind and float out:

```
f x = case (fromIntegral Int Float dict1 dict2 m) of
        F# v -> ...
```

Without inlining `fromIntegral` nothing happens and eventually we float the `case` scrutinee to the top level. But if `fromIntegral` is inlined we eventually get

```
f x = case int2Integer m of
        J# u1# u2# u3# -> case encodeFloat#! u1# u2# u3# 0# of
                            v -> ...
```

and as `encodeFloat#!` returns an unboxed float we cannot `let`-bind it and float it to the top level. The only thing we get to float is `(int2Integer m)`, and therefore we end up evaluating `encodeFloat` over and over again. This caused a program to run 50% slower with more aggressive inlining!

Another good reason to have the full laziness transformation early during the compilation is that it avoids some possible bad interactions with the "join points" we use for the `case` of `case` and `case` floating from `let` transformations (Sections 3.5.2 and 3.5.3). As we know, join points are a special kind of `let`, which the compiler can later optimise to a jump, therefore incurring no cost for its "allocation". The problem is that as we abstract some variables during the creation of the join point, some expressions might be spotted as being suitable for full laziness. But join points are *linear*, in the sense that they will not be entered multiple times, therefore there is no advantage in moving expressions out of them. Actually that may introduce extra `let`s that will be allocated unnecessarily. Let us look at an example:

```
let v = case E1 of                let j = \v -> E4
          C1 a b -> E2     ===>   in case E1 of
          C2 a b -> E3                C1 a b -> let v = E2 in j v
in E4                                 C2 a b -> let v = E3 in j v
```

In this case, the full laziness transformation may float some subexpression of `E4` which does not depend on `v` from the right hand side of `j`, creating a new `let`-binding unnecessarily.

5.4.6 The ordering we use

The following ordering obeys all the constraints above, except 5.4.2.

1 – full laziness
2 – float-in
3 – simplify
4 – strictness
5 – worker-wrapper generation
6 – simplify
7 – float-in
8 – simplify

5.5 Conclusions

We have presented the `let` floating inwards transformation which produced good results for some programs. This transformation was suggested by inspecting the intermediate code generated by the compiler.

Code inspection again suggested the use of the full laziness transformation, which is often regarded as too dangerous (due to the risks of space leaks) to be integrated into compilers. We have suggested improvements to reduce and/or eliminate the risk of space leaks, and we advocate that the transformation should be available at least as an option in optimising compilers, since the actual creation of a space leak by the transformation in real programs might occur far less often than is generally believed.

Chapter 6

Inlining

Procedure inlining is an optimisation often used in imperative languages' compilers [ASU87]. It consists of heuristically selecting some (usually small) procedures to be inlined, that is, every call to the procedure is replaced by the actual code of the procedure. Inlining aims to save time by eliminating the overhead of these procedure calls and increasing the opportunity for other optimisations, since the procedure code is now exposed to local context information and therefore to more optimisations. But inlining must be done carefully, since excessive inlining can easily lead to a large increase in code size as one is in fact duplicating code. In imperative languages' compilers inlining has been reported to improve programs' execution time by 18% [RG89], 12% [DH88] and 10% [Cho83]. [DH92] presents a comprehensive analysis on the effect of inlining in imperative languages.

In the functional framework, function definitions can also be inlined at their call sites. There is the same risk of code explosion due to excessive code duplication, but, done in a controlled way, similar benefits can be obtained, since opportunities for local optimisation often appear.

6.1 Inlining and lazy functional languages

In lazy functional languages it is always safe to substitute equals for equals, i.e. one cannot change the semantics of a program by inlining.

The process of inlining in a functional language can be described as:

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ body \Longrightarrow \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ body[e/x]$$

which means we are replacing some (or all) occurrences of `x` by the expression `e`. If eventually all occurrences of `x` are inlined one can apply the dead code removal transformation to eliminate the `let`-binding.

Notice that the `let`-binding that we are inlining may be binding a function or simply an expression: many of the issues involved in deciding what `let`s to inline apply equally well to both functions and non-functions, therefore we make no distinction between them at this point. This way we also separate the inlining of functions from the beta-reduction that usually immediately follows it, although these two transformations in conjunction are closer to the concept of inlining in imperative languages.

The main advantages that come from inlining are:

- ✓ the definition is available in the place of use, allowing some transformations like β-reduction (section 3.1) to occur;
- ✓ more things may be evaluated at the call site, allowing transformations like `case` reduction (section 3.3.1) to occur.

But it also has the following risks:

- × code duplication, if expressions are inlined when they occur multiple times;
- × work duplication if the inlining is not carefully done (redex copying).

An example of work duplication due to a bad inlining decision would occur if we decide to inline the variable `v` in the following expression:

```
let v = fib 20    ==> fib 20 + fib 20
in v + v
```

Although the two expressions are semantically equivalent (both give the same result), the latter is much more expensive to evaluate, as `fib 20` is evaluated twice instead of once.

Therefore although the semantics of lazy functional languages allows us to inline without major concerns, efficiency issues impose some restrictions on what we can inline without increasing the costs of evaluation. These basic restrictions are related to *updatable* closures, that can have their evaluation shared, i.e. these restrictions do not apply to non-updatable closures (notably functions and constructors). These restrictions are:

- *do not inline updatable closures if they occur more than once.* This is the case in the example above. If the closure is non-updatable (e.g. a function) there is no risk of work duplication by inlining it multiple times:

  ```
  let f = \ x -> fib 20 + x
  in f 5 + f 6
  ==> (\ x -> fib 20 + x) 5 + (\ x -> fib 20 + x) 6
  ==> fib 20 + 5 + fib 20 + 6
  ```

  This restriction can be relaxed a bit more if the multiple occurrences are (single occurrences) in different `case` branches. Since only one of the branches can be taken at a particular time, the expression cannot be evaluated more than once, although code has been duplicated:

  ```
  let v = fib 20                case e of
  in case e of          ==>       True  -> fib 20 + 2
       True  -> v + 2             False -> fib 20 * 2
       False -> v * 2
  ```

- *do not inline updatable closures past lambda abstractions.* The problem here is that if the expression is inlined past the lambda abstraction it will be evaluated as many times as the lambda abstraction is entered, and not only once as it was before:

  ```
  let v = fib 20                  let f = \ x -> x + fib 20
  in let f = \ x -> x + v  ==>    in f 3 + f 4
     in f 3 + f 4
  ```

  The evaluation of `fib 20` is shared in the first expression, and therefore occurs only once, while in the second one it occurs twice, one for each call to `f`.

Notice that the tagging of closures as being update or not is supplying us with information about which closures, if inlined indiscriminately, may cause work duplication (updatable closures) and which won't (non-updatable closures). The compiler can be regarded as performing an extremely simple form of "update analysis", by tagging functions and other weak head normal forms as non-updatable closures, and all other closures as updatable. More elaborate update analysis techniques can improve this by finding out some of the updatable closures that do not actually need to be updated [LGH+92, Mar93, MTW95, MOTW95]. Some of these analyses will detect lambda abstractions which are guaranteed to be entered only once, and therefore inlining non weak head normal forms into them cannot duplicate work, allowing some of those closures to be tagged as non-updatable.

These analyses have only recently become available [MTW95], and we do not make use of them in the work presented in this thesis.

In summary, the major advantage of inlining comes from increasing the possibility of other transformations being applied. But due to its possible code duplication (whenever the expression to be inlined occurs more than once) the decision to inline should be done only when there is a good chance that the transformations will actually occur.

In the next sections we will discuss some methods for taking this decision.

6.2 Basic inlining

According to the restrictions for inlining we discussed in the previous section, one can see that there are a few basic cases in which inlining can be done safely, depending on the form of the right hand sides of the bindings:

- *variables:*

  $$\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ body \implies body[v/x]$$

  The transformation basically removes one level of indirection to the variable (v in this case):

  ✓ saves the allocation of the closure for x, as we will not have to keep the original definition.

  ✓ saves the update to x if it is ever entered.

  ✓ saves one enter, since if x was entered it would then enter the variable v, but after inlining the variable is entered directly.

  One may also refer to this form of inlining as "copy propagation".

- *constructors:*

  $$\texttt{let}\ x\ \texttt{= MkInt 5 in}\ body \implies \texttt{let}\ x\ \texttt{= MkInt 5 in}\ body[(\texttt{MkInt 5})/x]$$

  The transformation basically removes one level of indirection to the constructor (`MkInt 5` in this case). It also saves the allocation of the closure for x, if all occurrences are inlined.

- *expressions that occur only once (not inside a lambda), or functions that occur only once*:

  $$\texttt{let}\ x\ \texttt{=}\ \mathit{fib}\ 20\ \texttt{in}\ x \implies \texttt{let}\ x\ \texttt{=}\ \mathit{fib}\ 20\ \texttt{in}\ \mathit{fib}\ 20$$

  This

  ✓ saves the allocation of the closure for x;

  ✓ saves the update to x if it is ever entered;

  ✓ may expose transformations, due to the new local context.

  Due to the syntax of the Core language, one can only inline these `let`s if they occur in a position where an expression is allowed, that is, we cannot inline if the variable occurs as an argument to a function, a constructor or a primitive operator:

```
h y = let f = fib 20
      in g f
```

  `f` cannot be inlined, as the Core language does not allow for an expression to occur as an argument. This does not cause any problems, since the only possible advantage of such an inlining would be to avoid allocating the closure too early, e.g. if it was used in only one branch of a `case` and was being allocated outside the `case`:

```
h y = let f = fib 20
      in case y of
           1 -> g f
           n -> let v = y - 1
                in h v
```

  `f` would be *allocated* regardless of which branch is taken, although it would only (possibly) be needed if the first branch was taken. We already deal with this possibility with the floating inwards transformation in Chapter 5, therefore there is no disadvantage in losing these possible inlining opportunities.

Most compilers perform these simple forms of inlining, e.g. [KKR+86, Aug87, App92].

6.3 Inlining strategy

By inlining function definitions with multiple occurrences we do not risk duplicating work, but we do risk duplicating code. Inlining functions often exposes not only beta-reductions, but many other transformations, especially `case` reduction (Section 3.3.1). On the other hand we do not want to inline big functions many times just to find out that we only did a few beta-reductions, and therefore we are only saving the costs of the function call.

Every compiler uses its own method to assess which functions are worth inlining [App92], often based in some notion of "size" of the function being inlined, which amounts to a way of counting the language constructs in the function. Then a notion of "discount" is introduced, which gives discounts to the size of the function on a occurrence by occurrence basis [App92], according to a relation between its size and number of occurrences [Bee93] or other criteria. Finally if the discounted size is smaller than a given threshold, the function is inlined.

The Glasgow Haskell Compiler, during occurrence analysis, records information about the right hand side of functions regarding its size and whether the function scrutinises any of its arguments (using a `case`, in its right hand side), and if so which ones. This supplies us with a rough estimate of how many `case` reductions we will get *if* in the place the function is used it is given a constructor (or variable known to be bound to a constructor) as an argument:

```
f x y z = case x of
            (a,b) -> case z of
                       [] -> g b a y
                       (v:vs) -> g vs b y
```

In this case we will record that we will get `case` reductions if, when `f` is inlined, we know the constructor form of its first and third argument.

Supplied with this information, the compiler then chooses, *for each occurrence* of the function in the program, whether inlining the function in that place will be worthwhile. This is done, given the size of the function, by discounting its size for each argument that is a constructor (or is known to be bound to a constructor) and which is scrutinised by the function. We then decide to inline it *at that site* if the "discounted" size is smaller than a given threshold. Therefore the cost of a particular inlining is calculated using the following formula:

$$cost = size\ of\ function\ body - discounts\ from\ call\ site$$

We will first describe how we calculate the cost of a function, in Figure 6.1.

The only unusual definitions are the ones for `case`s:

- if a `case` is scrutinising a variable we do not charge for that variable ((i) and (k)); this allows us to keep low the cost of such `case`s, which are particularly likely to benefit from any information we may get on that variable if we inline the expression.
- primitive `case`s add no extra costs, just add the cost of its subexpressions ((i) and (j));
- algebraic `case`s add to the cost of its subexpressions a cost equal to the number of constructors of the data type (*confamilysize*) it is scrutinising ((i) and (j)), e.g. it would add 1 for tuples and 2 for lists. This decision will become clear after we describe the "discounting" system below.

We then, *at each call site*, apply the following discounts:

- number of arguments: we discount 1 for each argument the function is applied to; this accounts for the beta-reductions we will get due to inlining.
- we discount *confamilysize*, for each argument we know is bound to a constructor *and* we know we scrutinise; this accounts for the `case` reductions we know we are going to get from inlining the function, and the more `case` alternatives we eliminate by such a reduction the better, therefore the use of the *confamilysize*.

We then proceed to compare the discounted size against a given *inlining threshold*, which can be set by a command line flag. If the discounted size is smaller than the threshold we inline the function.

The use of *confamilysize* when dealing with `case`s allows us to make the following sort of distinction:

```
f x y z = case x of
            (a,b) -> case z of
                       alt_1 -> E_1
                       ...
                       alt_10 -> E_10
```

$$
\begin{array}{lll}
 & \mathcal{C}[\![\ ]\!] & :: Expr \rightarrow Int \\
(a) & \mathcal{C}[\![k]\!] & = 1 \\
(b) & \mathcal{C}[\![v]\!] & = 1 \\
(c) & \mathcal{C}[\![C\ v_1 \ldots v_n]\!] & = 1 + n \\
(d) & \mathcal{C}[\![op\ v_1 \ldots v_n]\!] & = 1 + n \\
(e) & \mathcal{C}[\![\lambda v_1 \ldots v_n . E]\!] & = n + \mathcal{C}[\![E]\!] \\
(f) & \mathcal{C}[\![\Lambda t_1 \ldots t_n . E]\!] & = \mathcal{C}[\![E]\!] \\
(g) & \mathcal{C}[\![E\ v_1 \ldots v_n]\!] & = n + \mathcal{C}[\![E]\!] \\
(h) & \mathcal{C}[\![E\ T]\!] & = \mathcal{C}[\![E]\!] \\
(i) & \mathcal{C}[\![\texttt{case}\ v\ \texttt{of}\ \{primalt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!] & \\
 & & = \sum_{i=1}^{n} \mathcal{C}[\![E_i]\!] \\
(j) & \mathcal{C}[\![\texttt{case}\ E\ \texttt{of}\ \{primalt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!] & \\
 & & = \sum_{i=1}^{n} \mathcal{C}[\![E_i]\!] + \mathcal{C}[\![E]\!] \\
(k) & \mathcal{C}[\![\texttt{case}\ v\ \texttt{of}\ \{algalt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!] & \\
 & & = \sum_{i=1}^{n} \mathcal{C}[\![E_i]\!] + confamilysize \\
(l) & \mathcal{C}[\![\texttt{case}\ E\ \texttt{of}\ \{algalt_i\ \texttt{->}\ E_i\}_{i=1}^{n}]\!] & \\
 & & = \sum_{i=1}^{n} \mathcal{C}[\![E_i]\!] + \mathcal{C}[\![E]\!] + confamilysize \\
(m) & \mathcal{C}[\![\texttt{let}\ \{v_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E]\!] & \\
 & & = 1 + \sum_{i=1}^{n} \mathcal{C}[\![E_i]\!] + \mathcal{C}[\![E]\!]
\end{array}
$$

Figure 6.1 Inlining: cost of an expression

At a given call site, we would like to give a bigger discount if we have information about the constructor for the third argument (`z`) than if we know information about the first argument (`x`). The fact that when calculating the cost of a `case` expression we also add *confamilysize* allows us to be sure that with a inlining threshold 0 we will not get any inlining happening due to the inlining strategy[1].

These notions of "size" and "discounts" are rather arbitrary, but by varying the *inlining threshold* we can effectively increase the aggressiveness of the inlining strategy.

The main advantage of our criteria as opposed to the usual strategy of deciding to "inline all functions of up to size n" is that this sort of criteria does not take into account the possible increase of opportunities for other transformations, therefore one often ends up with inlinings that only save the function call overhead, but nothing else. We also make our decision *for each occurrence*, rather than having one decision for all occurrences made beforehand.

Our approach is more similar to the one used in [App92], where he also uses the notions of cost of a function and discounts at each occurrence. His costs and discount functions, though, are much more elaborate than ours.

The Glasgow Haskell Compiler also allows "inlinings" to be exported across modules, by including (pre-processed) function definitions in interface files. This means that we are not limited to inlining within a module boundary. To avoid gratuitous exporting of all exported functions in a module, we impose the following limit on the discounted size of a function being exported: supposing it gets all the discounts it can (i.e. it is applied to all arguments it expects *and* we know the constructors of all arguments), if it still has size greater than 3 we do not export it. This limit was set based on the default inlining threshold in the Glasgow Haskell Compiler, which is 3, and it probably should vary together with the supplied inlining threshold, although we have not experimented with varying it.

6.4 Inlining recursive `lets`

Selecting recursive lets for inlining must be done even more carefully, since we may risk non-termination by inlining them. For this reason, none of the implementations we know of inline recursive `lets`. The optimisation we try to get from recursive lets is to reduce the number of bindings, by combining mutually recursive functions

[1]The basic forms of inlining described in Section 6.2 may still happen, as the decision for their inlining is not dependent on the notion of costs described in this section.

whenever possible. The idea is that if a particular binding does not occur in its own right hand side (that is, it is not self-recursive) it can be safely inlined, e.g.:

```
   let rec odd  x = case x of
                      1 -> True
                      _ -> even (x-1)
           even y = case y of
                      1 -> False
                      _ -> odd (y-1)
   in ... odd ...
===>
   let rec odd  x = case x of
                      1 -> True
                      _ -> case (x-1) of
                             1 -> False
                             _ -> odd (y-1)
   in ... odd ...
```

The criteria for selecting a particular function to be inlined, after knowing that it is not self-recursive, may be similar to the one adopted for non-recursive bindings, which we presented in previous sections.

Notice that even without explicit recursion it is possible to obtain an infinite sequence of possible inlinings. Look at this example:

```
data T = C (T -> T)

f x = let g x = case x of
                  C f -> f x
      in g (C g)
```

Here `g` (which is not recursive) can be inlined, but after inlining, beta-reduction and `case` reduction give back the original expression! This is an example that indeed causes the simplifier to keep iterating (since we are always performing some simplifications), and the only way of guaranteeing termination is by having a fixed maximum number of iterations.

6.5 Interaction with other transformations

We did not expect initially that increasing the inlining threshold would have any negative effect in any program, except for increasing the code size.

Actually, when increasing the threshold we were surprised twice by the inlining interacting with other transformations:

- *Floating inwards.* This interaction was described in Chapter 5 (Section 5.1.5). Due to code duplication that occurs when inlining, some `let`s which originally were being floated into other `let`s are now being left in an outer context (because they now occur in more places).
- *Full laziness.* This interaction was described in Chapter 5 (Section 5.4.5). The inlining and simplifications meant that some expressions that were being computed only once and shared were now being left inside lambdas, due to being simplified to unboxed expressions.

6.6 Results

In the following tables we have measured the effects of increasing the inlining threshold when compiling our benchmark programs.

Table 6.1 shows the effect on instructions executed as the inlining threshold is varied. The column labelled "off" has inlining of functions turned off, although inlining of variables and constructors still happens (but not expressions or functions that occur only once). The column labelled "one occ." has the results for inlining expressions or functions that occur only once. It is clear that we start having diminishing returns for thresholds greater than 4, and that inlining expressions or functions that occur only once is very important. The same is true for total heap allocation, presented in Table 6.2. Only for threshold 32 we have used our more aggressive floating inwards strategy, and this was the threshold that had the major increase in heap allocation described in Chapter 5 (Section 5.1.5).

Table 6.3 presents the effect of inlining on the object code sizes, which quite surprisingly does not go up significantly with larger inlining thresholds. Compilation time (Table 6.6), on the other hand, is heavily affect by increasing the inlining threshold.

The number of functions inlined (Table 6.4) and the number of `case` reductions (Table 6.5) are also presented. When comparing the number of functions inlined and the number of `case` reductions the base column (from which the other ones are relative to) is shown with the actual number of occurrences of each transformation. Notice that inlining of constructors and variables are not included in these numbers.

The results obtained by [App92] (from 6 programs) were of about 25% improvement from what we call "basic inlining", without a major increase in the object code size. After that he still gets up to 9% improvement, but with up to 25% increase in the object code size. He does not mention the effect on compilation time.

Our results are quite similar for "basic inlining", in which we get about 20% improvement, but we get another 20% with our inlining strategy, without major effects on the object code size.

6.7 Conclusions

To our surprise we did not get code explosion when we incremented the inlining threshold. On the other hand it seems that there is not much to be gained from inlining large functions, as can be seen by the increase in the number of functions inlined with larger thresholds not being reflected in a reduction in the number of instructions executed. Even though it is clear that there is some improvement (by looking at the number of `case` reductions that occur), it is not enough to cause a major effect in the overall number of instructions executed. The increase in compilation time seems to be too high to make it worthwhile to increase the threshold much more than we did.

Currently the inlining threshold used by the Glasgow Haskell Compiler is 3, and it seems that this is a good compromise between compilation time and reduction in instructions executed.

It would be interesting in the future to investigate the effect of having the inter-module inlining limit set to the same level as the inlining threshold. This would allow more inter-module inlining, although the possible increase in the interface files' size could possibly have major effects in compilation time.

Inlining – Total Instructions Executed											
	off	one	threshold								
program		occ.	0	1	2	3	4	8	16	32
mandel2	1.00	0.88	0.76	0.27	0.24	0.22	0.19	0.19	0.19	0.19
queens	1.00	0.67	0.51	0.44	0.25	0.25	0.25	0.25	0.25	0.25
primes	1.00	0.48	0.34	0.31	0.31	0.31	0.30	0.30	0.30	0.30
wave4main	1.00	0.46	0.43	0.34	0.33	0.31	0.30	0.29	0.28	0.26
treejoin	1.00	0.61	0.47	0.41	0.39	0.37	0.36	0.36	0.35	0.32
ida	1.00	0.79	0.63	0.48	0.44	0.43	0.44	0.42	0.37	0.39
parstof	1.00	0.70	0.56	0.56	0.47	0.44	0.44	0.44	0.42	0.42
sched	1.00	0.92	0.72	0.70	0.70	0.49	0.49	0.49	0.48	0.48
maillist	1.00	0.58	0.54	0.53	0.53	0.50	0.50	0.50	0.48	0.46
solid	1.00	0.95	0.85	0.50	0.55	0.52	0.51	0.50	0.51	0.50
hpg	1.00	0.69	0.60	0.57	0.55	0.54	0.52	0.52	0.51	0.50
mandel	1.00	0.81	0.73	0.60	0.56	0.54	0.54	0.53	0.51	0.49
event	1.00	0.85	0.73	0.56	0.55	0.55	0.55	0.55	0.55	0.55
fluid	1.00	0.74	0.63	0.58	0.56	0.55	0.54	0.54	0.53	0.50
reptile	1.00	0.69	0.65	0.63	0.59	0.58	0.58	0.58	0.58	0.58
rewrite	1.00	0.75	0.64	0.63	0.63	0.58	0.58	0.53	0.52	0.52
hidden	1.00	0.83	0.64	0.62	0.60	0.59	0.59	0.54	0.54	0.54
sorting	1.00	0.73	0.63	0.63	0.59	0.59	0.59	0.59	0.59	0.59
genfft	1.00	0.83	0.80	0.66	0.63	0.61	0.61	0.61	0.61	0.61
prolog	1.00	0.75	0.64	0.62	0.61	0.61	0.60	0.60	0.60	0.60
compress	1.00	0.75	0.72	0.65	0.63	0.63	0.63	0.63	0.62	0.63
fft	1.00	0.86	0.85	0.68	0.65	0.63	0.61	0.60	0.60	0.57
gen_regexps	1.00	0.73	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.63
wang	1.00	0.94	0.91	0.82	0.65	0.63	0.63	0.63	0.63	0.63
infer	1.00	0.74	0.65	0.65	0.65	0.65	0.65	0.65	0.65	0.65
lift	1.00	0.71	0.68	0.66	0.66	0.66	0.66	0.66	0.66	0.65
cichelli	1.00	0.85	0.74	0.68	0.68	0.68	0.68	0.67	0.67	0.67
gg	1.00	0.81	0.76	0.73	0.69	0.68	0.68	0.67	0.66	0.65
knights	1.00	0.72	0.78	0.69	0.68	0.68	0.57	0.57	0.56	0.56
transform	1.00	0.97	0.84	0.77	0.70	0.69	0.69	0.69	0.69	0.68
pretty	1.00	0.77	0.77	0.71	0.70	0.70	0.70	0.70	0.70	0.70
boyer2	1.00	0.80	0.71	0.71	0.71	0.71	0.70	0.66	0.66	0.66
typecheck	1.00	0.80	0.74	0.73	0.71	0.71	0.71	0.71	0.68	0.68
comp_lab_zift	1.00	0.91	0.81	0.75	0.74	0.73	0.74	0.74	0.73	0.73
parser	1.00	0.80	0.75	0.75	0.73	0.73	0.73	0.71	0.71	0.71
multiplier	1.00	0.89	0.84	0.81	0.74	0.74	0.74	0.74	0.73	0.73
clausify	1.00	0.82	0.82	0.76	0.76	0.76	0.76	0.71	0.71	0.71
fft2	1.00	0.92	0.91	0.87	0.78	0.77	0.76	0.76	0.76	0.76
minimax	1.00	0.90	0.88	0.87	0.78	0.78	0.78	0.78	0.78	0.78
listcompr	1.00	0.84	0.83	0.83	0.82	0.82	0.82	0.82	0.82	0.82
listcopy	1.00	0.85	0.84	0.84	0.83	0.83	0.83	0.83	0.83	0.83
veritas	1.00	0.86	0.85	0.84	0.84	0.84	0.84	0.84	0.83	0.83
boyer	1.00	0.96	0.96	0.96	0.96	0.96	0.96	0.92	0.92	0.92
rsa	1.00	0.99	0.99	0.98	0.98	0.98	0.98	0.98	0.98	0.98
primetest	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
exp3_8	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Minimum	-	0.46	0.34	0.27	0.24	0.22	0.19	0.19	0.19	0.19
Maximum	-	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Geometric mean	-	0.79	0.72	0.65	0.62	0.60	0.60	0.59	0.58	0.58

Table 6.1 Inlining: instructions executed

Inlining – Total Heap Allocated											
	off	one	threshold								
program		occ.	0	1	2	3	4	8	16	32	
`queens`	1.00	0.32	0.15	0.15	0.12	0.12	0.12	0.12	0.12	0.12	
`wave4main`	1.00	0.23	0.23	0.13	0.13	0.13	0.13	0.13	0.13	0.21	
`primes`	1.00	0.23	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	
`knights`	1.00	0.18	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.15	
`mandel2`	1.00	1.02	0.88	0.25	0.23	0.23	0.19	0.19	0.20	0.20	
`treejoin`	1.00	0.37	0.34	0.28	0.27	0.27	0.28	0.28	0.28	0.34	
`solid`	1.00	0.94	0.82	0.39	0.34	0.34	0.34	0.34	0.34	0.34	
`ida`	1.00	0.79	0.73	0.46	0.36	0.35	0.35	0.35	0.35	0.36	
`infer`	1.00	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.35	
`maillist`	1.00	0.44	0.44	0.43	0.43	0.43	0.43	0.43	0.42	0.43	
`cichelli`	1.00	0.57	0.57	0.45	0.45	0.45	0.45	0.45	0.45	0.42	
`parstof`	1.00	0.97	0.44	0.43	0.50	0.50	0.50	0.49	0.38	0.38	
`compress`	1.00	0.55	0.55	0.55	0.51	0.51	0.51	0.51	0.50	0.51	
`hpg`	1.00	0.61	0.56	0.53	0.52	0.51	0.50	0.51	0.51	0.50	
`event`	1.00	0.75	0.70	0.52	0.52	0.52	0.52	0.52	0.52	0.52	
`fluid`	1.00	0.62	0.61	0.56	0.54	0.52	0.51	0.52	0.54	0.54	
`sorting`	1.00	0.64	0.64	0.64	0.52	0.52	0.52	0.52	0.52	0.52	
`boyer2`	1.00	0.51	0.56	0.56	0.56	0.56	0.56	0.45	0.45	0.44	
`gg`	1.00	0.73	0.69	0.66	0.62	0.58	0.58	0.60	0.63	0.63	
`reptile`	1.00	0.59	0.61	0.61	0.56	0.58	0.58	0.58	0.59	0.59	
`rewrite`	1.00	0.71	0.70	0.70	0.69	0.60	0.60	0.51	0.50	0.50	
`wang`	1.00	0.89	0.88	0.78	0.60	0.60	0.60	0.60	0.60	0.60	
`parser`	1.00	0.63	0.65	0.66	0.61	0.61	0.61	0.61	0.61	0.61	
`hidden`	1.00	0.78	0.66	0.65	0.64	0.63	0.63	0.60	0.60	0.60	
`prolog`	1.00	0.65	0.64	0.63	0.62	0.63	0.63	0.62	0.61	0.63	
`lift`	1.00	0.67	0.66	0.65	0.65	0.65	0.64	0.64	0.64	0.65	
`typecheck`	1.00	0.64	0.64	0.62	0.61	0.65	0.64	0.64	0.63	0.63	
`fft2`	1.00	0.88	0.87	0.83	0.66	0.66	0.66	0.66	0.66	0.66	
`genfft`	1.00	0.79	0.81	0.66	0.63	0.66	0.66	0.66	0.66	0.65	
`sched`	1.00	0.90	0.89	0.88	0.87	0.66	0.66	0.66	0.66	0.66	
`mandel`	1.00	0.82	0.78	0.70	0.68	0.68	0.68	0.68	0.68	0.66	
`pretty`	1.00	0.70	0.71	0.69	0.68	0.68	0.68	0.68	0.68	0.68	
`clausify`	1.00	0.59	0.59	0.71	0.71	0.71	0.71	0.71	0.71	0.71	
`minimax`	1.00	0.84	0.84	0.84	0.73	0.73	0.73	0.73	0.73	0.73	
`gen_regexps`	1.00	0.56	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.74	
`listcompr`	1.00	0.74	0.73	0.73	0.73	0.74	0.73	0.73	0.73	0.73	
`multiplier`	1.00	0.84	0.82	0.79	0.74	0.74	0.74	0.74	0.71	0.71	
`comp_lab_zift`	1.00	0.89	0.80	0.76	0.74	0.75	0.75	0.75	0.75	0.73	
`fft`	1.00	0.86	0.87	0.78	0.76	0.75	0.74	0.74	0.74	0.74	
`listcopy`	1.00	0.76	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	
`transform`	1.00	0.96	0.91	0.74	0.75	0.75	0.75	0.75	0.75	0.75	
`rsa`	1.00	0.79	0.78	0.82	0.81	0.78	0.78	0.78	0.77	0.80	
`veritas`	1.00	0.80	0.81	0.81	0.80	0.80	0.80	0.80	0.80	0.80	
`primetest`	1.00	0.79	0.79	0.84	0.83	0.81	0.81	0.81	0.81	0.83	
`boyer`	1.00	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	
`exp3_8`	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
Minimum	-	0.18	0.14	0.13	0.12	0.12	0.12	0.12	0.12	0.12	
Maximum	-	1.02	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
Geometric mean	-	0.65	0.61	0.55	0.53	0.52	0.52	0.51	0.51	0.52	

Table 6.2 Inlining: heap allocated

Inlining – Binary Size										
	off	one	threshold							
program		occ.	0	1	2	3	4	8	16	32
`compress`	1.00	0.90	0.83	0.81	0.81	0.81	0.81	0.81	0.81	0.83
`fluid`	1.00	0.95	0.91	0.86	0.84	0.84	0.85	0.87	0.91	0.94
`wave4main`	1.00	0.94	0.89	0.84	0.84	0.84	0.84	0.84	0.87	0.90
`mandel2`	1.00	0.63	0.60	0.86	0.85	0.85	0.58	0.59	0.59	0.62
`pretty`	1.00	0.93	0.88	0.85	0.85	0.85	0.85	0.87	0.90	0.93
`wang`	1.00	0.93	0.88	0.85	0.85	0.85	0.85	0.87	0.88	0.91
`boyer2`	1.00	0.91	0.88	0.88	0.88	0.86	0.86	0.88	0.88	0.89
`comp_lab_zift`	1.00	0.90	0.88	0.88	0.88	0.86	0.86	0.88	0.88	0.92
`fft`	1.00	0.94	0.89	0.86	0.86	0.86	0.86	0.88	0.89	0.94
`fft2`	1.00	0.94	0.89	0.86	0.86	0.86	0.86	0.87	0.89	0.91
`knights`	1.00	0.90	0.88	0.86	0.86	0.86	0.86	0.86	0.86	0.90
`mandel`	1.00	0.94	0.88	0.86	0.86	0.86	0.86	0.87	0.88	0.91
`rewrite`	1.00	0.91	0.88	0.88	0.88	0.86	0.88	0.88	0.89	0.95
`clausify`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.89	0.89	0.91
`event`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.89	0.89	0.91
`exp3_8`	1.00	0.91	0.89	0.89	0.87	0.87	0.87	0.87	0.87	0.89
`gg`	1.00	0.95	0.90	0.88	0.87	0.87	0.88	0.89	0.93	0.98
`hidden`	1.00	0.94	0.90	0.87	0.87	0.87	0.87	0.89	0.93	0.95
`ida`	1.00	0.91	0.89	0.87	0.87	0.87	0.87	0.87	0.87	0.91
`listcompr`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.87	0.87	0.89
`listcopy`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.87	0.87	0.91
`minimax`	1.00	0.92	0.90	0.90	0.87	0.87	0.87	0.90	0.90	0.92
`primes`	1.00	0.91	0.89	0.89	0.87	0.87	0.87	0.87	0.87	0.89
`queens`	1.00	0.89	0.89	0.87	0.87	0.87	0.87	0.87	0.87	0.89
`sched`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.87	0.87	0.89
`solid`	1.00	0.95	0.91	0.87	0.87	0.87	0.87	0.88	0.89	0.92
`treejoin`	1.00	0.91	0.89	0.89	0.89	0.87	0.87	0.89	0.89	0.91
`cichelli`	1.00	0.92	0.90	0.88	0.88	0.88	0.88	0.88	0.88	0.92
`genfft`	1.00	0.90	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.92
`infer`	1.00	0.91	0.89	0.89	0.89	0.88	0.88	0.89	0.89	0.93
`maillist`	1.00	0.92	0.90	0.88	0.88	0.88	0.88	0.88	0.88	0.90
`multiplier`	1.00	0.92	0.90	0.88	0.88	0.88	0.88	0.90	0.90	0.94
`primetest`	1.00	0.90	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.92
`prolog`	1.00	0.92	0.90	0.90	0.88	0.88	0.88	0.88	0.90	0.92
`typecheck`	1.00	0.90	0.90	0.88	0.88	0.88	0.88	0.88	0.88	0.92
`reptile`	1.00	0.94	0.91	0.89	0.89	0.89	0.89	0.89	0.90	0.97
`sorting`	1.00	0.91	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.91
`boyer`	1.00	0.92	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.92
`gen_regexps`	1.00	0.92	0.90	0.90	0.90	0.90	0.88	0.90	0.90	0.92
`hpg`	1.00	0.98	0.92	0.90	0.90	0.90	0.90	0.91	0.94	0.98
`lift`	1.00	0.93	0.91	0.90	0.90	0.90	0.88	0.90	0.91	0.95
`rsa`	1.00	0.92	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.94
`transform`	1.00	0.94	0.92	0.92	0.91	0.91	0.91	0.91	0.92	0.97
`parser`	1.00	0.94	0.93	0.93	0.93	0.93	0.93	0.94	0.94	0.96
`parstof`	1.00	0.96	0.95	0.93	0.93	0.93	0.92	0.93	0.97	1.01
`veritas`	1.00	0.96	0.94	0.93	0.93	0.93	0.93	0.95	0.96	1.03
Minimum	-	0.63	0.60	0.81	0.81	0.81	0.58	0.59	0.59	0.62
Maximum	-	0.98	0.95	0.93	0.93	0.93	0.93	0.95	0.97	1.03
Geometric mean	-	0.92	0.89	0.88	0.88	0.87	0.87	0.88	0.88	0.92

Table 6.3 Inlining: binary size

Inlining – Total Functions Inlined										
	off	one	threshold							
program		occ.	0	1	2	3	4	8	16	32
boyer	0.00	17	1.00	1.06	1.06	1.06	1.12	2.18	2.41	2.41
prolog	0.00	157	1.09	1.19	1.23	1.24	1.27	1.27	1.30	1.34
boyer2	0.00	83	1.24	1.36	1.36	1.36	1.40	1.89	1.94	2.00
parstof	0.00	309	1.07	1.22	1.36	1.36	1.37	1.38	1.97	2.35
hidden	0.00	390	1.10	1.31	1.34	1.39	1.40	1.48	1.54	1.56
clausify	0.00	36	1.28	1.44	1.44	1.44	1.44	1.50	1.50	1.97
infer	0.00	210	1.19	1.31	1.40	1.49	1.52	1.55	1.84	1.87
minimax	0.00	49	1.27	1.45	1.45	1.49	1.49	1.90	2.00	1.96
sorting	0.00	41	1.12	1.46	1.49	1.49	1.49	1.49	1.49	1.68
hpg	0.00	484	1.14	1.25	1.39	1.51	1.62	1.65	1.71	1.78
veritas	0.00	943	1.16	1.31	1.43	1.51	1.56	1.69	1.79	1.79
fft2	0.00	81	1.12	1.28	1.48	1.54	1.57	1.62	1.64	1.69
lift	0.00	135	1.18	1.41	1.48	1.56	1.68	1.74	1.87	1.89
ida	0.00	105	1.25	1.43	1.51	1.63	1.75	1.80	1.97	2.58
event	0.00	41	1.22	1.56	1.59	1.66	1.66	1.66	1.76	1.90
comp_lab_zift	0.00	129	1.22	1.38	1.50	1.67	1.68	1.78	2.09	2.24
genfft	0.00	69	0.96	1.23	1.43	1.68	1.68	1.72	2.01	2.30
typecheck	0.00	48	1.00	1.40	1.65	1.69	2.02	2.25	3.08	3.25
cichelli	0.00	89	1.24	1.52	1.60	1.71	1.72	1.82	1.83	1.94
treejoin	0.00	45	1.27	1.53	1.73	1.73	1.73	1.73	1.73	1.73
queens	0.00	12	1.17	1.50	1.75	1.75	1.75	1.75	1.75	1.75
gen_regexps	0.00	22	1.14	1.32	1.55	1.77	1.86	1.86	1.86	1.86
rewrite	0.00	133	1.31	1.53	1.58	1.78	2.02	2.57	2.84	3.10
mandel	0.00	70	1.13	1.49	1.57	1.79	1.81	1.83	1.89	1.90
multiplier	0.00	96	1.31	1.50	1.76	1.81	2.09	2.17	2.40	2.48
fft	0.00	99	1.10	1.26	1.51	1.84	2.23	2.40	2.48	3.06
listcompr	0.00	18	1.28	1.56	2.00	1.89	2.17	2.17	2.17	2.17
listcopy	0.00	18	1.28	1.56	2.00	1.89	2.17	2.17	2.17	2.17
gg	0.00	433	1.28	1.55	1.70	1.94	2.00	2.07	2.16	2.19
transform	0.00	184	1.32	1.46	1.87	1.96	2.01	2.03	2.17	3.05
parser	0.00	351	1.62	1.79	1.94	1.98	2.04	2.10	2.16	2.18
knights	0.00	129	1.36	1.64	1.90	1.99	2.06	2.08	2.13	2.28
reptile	0.00	325	1.15	1.50	1.86	2.04	2.04	2.21	2.33	2.62
rsa	0.00	46	1.33	1.37	1.63	2.17	2.54	2.54	2.54	2.65
primes	0.00	3	1.67	2.00	2.33	2.33	2.67	2.67	2.67	2.67
mandel2	0.00	58	1.22	1.57	2.12	2.38	2.90	2.86	3.19	3.07
primetest	0.00	79	1.43	1.65	1.95	2.54	2.70	2.71	2.71	3.01
fluid	0.00	507	1.40	1.85	2.27	2.55	2.63	2.82	3.00	3.07
wave4main	0.00	134	1.08	1.75	2.24	2.59	2.74	2.78	3.14	2.88
sched	0.00	36	1.86	1.86	2.08	2.78	2.83	2.83	3.08	3.17
pretty	0.00	37	1.14	1.92	2.89	3.16	3.16	3.19	3.24	3.49
exp3_8	0.00	6	2.17	2.50	3.00	3.67	4.00	4.33	4.33	4.00
wang	0.00	29	1.21	1.48	2.48	4.07	4.48	4.52	4.97	4.97
solid	0.00	61	2.33	3.05	3.69	4.44	4.92	5.13	5.16	6.08
maillist	0.00	18	4.17	4.44	4.50	4.56	4.56	4.56	4.56	4.56
compress	0.00	36	6.31	6.72	6.67	6.81	6.81	6.83	6.92	6.97
Minimum	0.00	-	0.96	1.06	1.06	1.06	1.12	1.27	1.30	1.34
Maximum	0.00	-	6.31	6.72	6.67	6.81	6.81	6.83	6.92	6.97
Geometric mean	-	-	1.34	1.60	1.82	1.98	2.09	2.21	2.34	2.47

Table 6.4 Inlining: Total Functions Inlined

Inlining – case reductions										
	off	one	threshold							
program		occ.	0	1	2	3	4	8	16	32
boyer	0.67	6	1.00	1.00	1.00	1.00	1.00	2.00	2.00	2.00
sorting	0.76	25	1.04	1.00	1.00	1.00	1.00	1.16	1.16	1.56
clausify	0.36	14	1.21	1.21	1.21	1.21	1.21	1.79	2.14	5.50
hidden	0.37	248	1.07	1.27	1.29	1.26	1.25	1.33	1.66	1.80
lift	0.47	77	1.23	1.21	1.26	1.26	1.16	1.21	1.23	1.18
minimax	0.47	34	1.29	1.35	1.35	1.35	1.35	2.59	2.94	3.44
infer	0.51	41	1.29	1.17	1.20	1.39	1.39	1.44	2.07	2.10
typecheck	0.59	27	1.00	1.41	1.44	1.44	1.44	1.44	1.63	1.63
boyer2	0.17	53	1.58	1.58	1.58	1.58	1.70	2.45	2.45	2.79
rewrite	0.51	81	1.47	1.48	1.48	1.58	1.58	1.86	1.93	2.35
ida	0.37	78	1.21	1.47	1.59	1.62	1.68	1.71	1.94	3.21
cichelli	0.33	30	1.43	1.90	1.97	1.70	1.70	1.67	1.87	1.90
prolog	0.43	42	1.38	1.50	1.79	1.76	1.79	1.79	1.93	1.95
parstof	0.13	138	1.28	1.42	1.91	1.91	1.91	1.91	3.14	3.37
veritas	0.38	395	1.51	1.75	1.93	1.93	1.95	2.24	2.57	3.70
primes	0.00	3	1.33	2.00	2.00	2.00	2.00	2.00	2.00	2.00
event	0.27	30	1.23	2.07	2.13	2.13	2.13	2.13	2.17	2.50
fft	0.22	67	1.27	1.88	2.25	2.15	2.39	2.69	2.75	3.88
listcompr	0.41	17	1.24	1.06	2.24	2.18	2.18	2.18	2.18	2.18
listcopy	0.41	17	1.24	1.06	2.24	2.18	2.18	2.18	2.18	2.18
fluid	0.30	419	1.41	1.76	2.26	2.20	2.25	2.29	2.43	2.61
comp_lab_zift	0.57	63	1.49	1.75	2.13	2.22	2.24	2.17	2.17	3.22
treejoin	0.47	17	1.47	1.94	2.24	2.24	2.29	2.29	2.29	2.29
multiplier	0.58	81	1.43	1.83	2.26	2.26	2.28	2.28	2.42	2.54
gg	0.41	192	1.52	2.13	2.33	2.29	2.42	2.67	2.95	3.20
mandel2	0.15	86	1.23	1.63	2.37	2.35	2.57	2.57	2.71	2.85
hpg	0.34	100	1.19	1.38	2.14	2.39	2.58	3.01	3.23	3.42
knights	0.28	89	1.62	2.16	2.42	2.40	2.42	2.42	2.42	2.92
gen_regexps	0.73	11	1.55	2.18	2.82	2.55	2.55	2.55	2.55	2.55
wave4main	0.10	142	1.13	2.62	2.85	2.63	2.67	2.79	2.93	2.87
fft2	0.00	19	1.89	1.89	3.00	2.74	2.84	3.11	3.11	3.16
parser	0.25	112	2.12	2.41	2.79	2.77	2.85	3.12	3.43	3.79
pretty	0.73	15	1.40	2.40	2.93	2.87	2.87	2.87	2.87	3.27
primetest	0.26	61	1.59	2.36	2.89	3.11	3.00	3.00	3.00	3.52
sched	0.50	42	2.26	2.55	2.67	3.29	3.29	3.29	3.24	3.88
wang	0.58	19	1.16	2.26	4.32	3.32	3.32	3.32	3.16	3.16
exp3_8	0.71	7	1.71	2.14	2.29	3.43	4.00	4.57	4.57	5.71
mandel	0.14	7	1.43	3.00	4.14	3.43	3.43	3.43	3.86	4.86
transform	0.51	53	2.25	2.53	3.40	3.43	3.43	3.47	3.47	11.72
solid	0.59	63	1.41	3.02	3.71	3.49	3.67	4.43	4.37	5.79
reptile	0.27	165	1.46	2.95	3.72	3.52	3.52	3.78	4.10	4.65
compress	0.57	7	2.43	3.71	3.43	3.57	3.57	3.57	6.29	5.14
queens	0.25	4	1.75	2.00	3.75	3.75	3.75	3.75	3.75	3.75
genfft	0.10	21	1.67	3.76	4.33	4.14	4.14	4.24	4.29	5.05
rsa	0.21	14	2.21	2.43	4.00	4.71	4.14	4.14	4.14	4.71
maillist	0.00	4	3.75	6.75	6.50	6.75	6.75	6.75	6.75	6.75
Minimum	0.00	-	1.00	1.00	1.00	1.00	1.00	1.16	1.16	1.18
Maximum	0.76	-	3.75	6.75	6.50	6.75	6.75	6.75	6.75	11.72
Geometric mean	-	-	1.46	1.89	2.28	2.29	2.31	2.51	2.69	3.12

Table 6.5 Inlining: case reductions

Inlining – Compilation time										
	off	one	threshold							
program		occ.	0	1	2	3	4	8	16	32
`sched`	1.00	0.86	0.77	0.75	0.77	0.80	0.77	0.76	0.79	0.80
`ida`	1.00	0.92	0.87	0.80	0.84	0.86	0.91	0.92	1.02	2.34
`comp_lab_zift`	1.00	0.83	0.77	0.78	0.75	0.87	0.88	0.89	1.14	1.83
`wave4main`	1.00	0.89	0.89	0.76	0.75	0.87	0.81	0.86	1.04	1.24
`compress`	1.00	0.91	0.76	0.83	0.83	0.89	0.90	0.92	0.92	0.96
`solid`	1.00	1.00	0.91	0.88	0.91	0.91	0.92	0.94	0.92	0.93
`maillist`	1.00	0.90	0.85	0.85	0.86	0.93	0.86	0.89	1.03	1.02
`fft`	1.00	0.95	0.93	0.91	0.93	0.96	0.93	0.95	0.98	1.76
`genfft`	1.00	1.01	0.95	0.98	1.00	0.97	0.97	1.05	1.13	1.52
`listcompr`	1.00	0.91	0.91	0.93	0.94	1.00	0.95	0.97	0.99	1.02
`clausify`	1.00	0.99	0.97	1.02	0.99	1.01	1.01	1.05	1.16	1.40
`mandel2`	1.00	0.96	0.95	0.96	0.92	1.01	1.04	1.15	1.35	1.41
`event`	1.00	1.02	0.98	0.96	0.98	1.02	1.02	1.02	1.05	1.26
`multiplier`	1.00	1.08	1.03	1.11	1.02	1.03	1.12	1.16	1.25	1.34
`typecheck`	1.00	0.93	1.02	1.06	0.94	1.03	1.03	1.15	1.20	1.19
`infer`	1.00	0.92	0.95	1.05	1.02	1.04	1.05	1.10	1.21	1.28
`rewrite`	1.00	0.99	0.99	1.08	1.04	1.04	1.08	1.16	1.68	1.89
`fluid`	1.00	0.93	0.97	1.00	0.88	1.05	1.00	1.06	1.01	1.23
`sorting`	1.00	0.89	0.94	1.07	1.04	1.07	1.04	1.07	1.09	1.23
`treejoin`	1.00	0.91	0.94	1.09	1.06	1.07	1.07	1.07	1.08	1.16
`gg`	1.00	0.94	0.94	1.01	0.99	1.08	1.04	1.11	1.21	1.29
`listcopy`	1.00	0.94	0.94	0.99	1.01	1.08	0.99	0.99	1.01	1.08
`reptile`	1.00	1.04	1.04	1.07	1.06	1.08	1.05	1.11	1.21	1.35
`wang`	1.00	0.99	0.99	1.03	0.99	1.08	1.08	1.18	1.24	1.31
`hidden`	1.00	0.94	0.98	1.08	1.07	1.09	1.08	1.23	1.31	1.36
`hpg`	1.00	0.98	1.00	1.05	1.05	1.09	1.10	1.17	1.22	1.31
`lift`	1.00	0.93	0.96	1.02	1.04	1.09	1.09	1.19	1.27	1.31
`mandel`	1.00	0.98	1.00	1.03	1.02	1.09	1.12	1.15	1.22	1.36
`gen_regexps`	1.00	0.98	1.07	1.05	1.10	1.10	1.10	1.10	1.14	1.49
`knights`	1.00	1.00	1.02	1.09	1.06	1.10	1.10	1.16	1.23	1.37
`prolog`	1.00	1.02	1.00	1.07	1.04	1.10	1.12	1.12	1.26	1.28
`boyer2`	1.00	0.97	1.02	1.09	1.05	1.11	1.08	1.18	1.21	1.29
`boyer`	1.00	0.97	1.04	1.07	1.04	1.12	1.07	1.13	1.16	1.18
`cichelli`	1.00	0.97	0.98	1.06	1.05	1.12	1.10	1.20	1.30	1.42
`pretty`	1.00	0.97	1.04	1.08	1.08	1.12	1.10	1.14	1.27	1.32
`minimax`	1.00	0.94	1.02	1.09	1.12	1.13	1.13	1.20	1.27	1.45
`exp3_8`	1.00	1.08	1.05	1.09	1.09	1.14	1.14	1.27	1.28	1.41
`fft2`	1.00	0.99	1.03	1.16	1.10	1.14	1.13	1.18	1.23	1.47
`queens`	1.00	0.99	1.13	1.08	1.11	1.14	1.14	1.18	1.25	1.37
`primes`	1.00	0.97	0.99	1.13	1.10	1.16	1.24	1.16	1.28	1.43
`primetest`	1.00	0.93	1.01	1.09	1.11	1.16	1.22	1.23	1.29	1.45
`veritas`	1.00	0.99	1.10	1.14	1.14	1.16	1.12	1.14	1.21	1.18
`parstof`	1.00	1.59	1.26	1.24	1.17	1.19	1.17	1.27	1.38	1.34
`rsa`	1.00	1.02	1.01	1.11	1.08	1.22	1.25	1.37	1.42	1.49
`parser`	1.00	1.04	1.19	1.25	1.41	1.36	1.47	1.50	1.81	1.11
`transform`	1.00	1.07	1.43	1.42	1.29	1.61	1.50	1.57	1.55	1.24
Minimum	-	0.83	0.76	0.75	0.75	0.80	0.77	0.76	0.79	0.80
Maximum	-	1.59	1.43	1.42	1.41	1.61	1.50	1.57	1.81	2.34
Geometric mean	-	0.97	0.98	1.02	1.01	1.06	1.06	1.11	1.19	1.31

Table 6.6 Inlining: compilation time

Chapter 7

The static argument transformation and lambda lifting

In this chapter we present two transformations that are almost the inverse of each other:

- the static argument transformation tries to remove redundant arguments to recursive function calls, turning them into free variables in those calls;
- the lambda lifting transformation adds extra arguments to function definitions, i.e. it turns free variables into extra arguments so that the function can then be lifted to the top level.

As we will see, each of them has its advantages and disadvantages, and we will try to get the benefits from both by allowing these two seemingly incompatible transformations to work together.

7.1 The Static argument transformation

Some recursive functions receive arguments that are always passed unchanged in the recursive calls. One example of such a function is `foldr`:

```
foldr f z l = case l of
                []     -> z
                (a:as) -> let v = foldr f z as
                          in f a v
```

The `f` and `z` arguments are used in the recursive call unmodified and in the same position. They are what we call *static arguments*. A simple transformation could modify the above definition to avoid passing the *static arguments* in the recursive call, by defining a local function that does the same recursion with the *static arguments* as free variables:

```
foldr f z l = let foldr' l' = case l' of
                                [] -> z
                                (a:as) -> let v = foldr' as
                                          in f a v
              in foldr' l
```

This version has the following properties:

✓ It reduces the number of arguments passed in the recursive calls. This means that less arguments are pushed in the stack at each recursive function call.

✓ It exposes the possibility of inlining the function, as it is not recursive anymore (although it contains a recursive function in its body).

✓ It decreases the number of free variables of the `v` closure from 3 (`f`, `z` and `as`) to 2 (`foldr'` and `as`). In implementations like the STG machine this decreases the closure size, which is related to the number of free variables. Before the transformation any closure with a recursive call has the static arguments as free variables. After the transformation the static arguments are not free variables of the closure anymore, but the new local recursive function is a new free variable. For one static argument the number of free variables is reduced by one (the static argument) and increased by one (the new recursive function), therefore the number of free variables is unchanged. For two or more static arguments the number of free variables removed (the static arguments) is greater than the number of free variables introduced (always one, the new recursive function). This only applies if the recursive call occurs in a closure, not if it occurs as a tail call.

✓ if we had subexpressions that only referred to `f` and `z` we could, by using the full laziness transformation, lift those subexpressions out of the recursive loop, therefore avoiding recalculating its value at each iteration.

× It introduces an extra closure for the local recursive function.

Actually some abstract machines used for the implementation of functional languages need lambda lifting (Section 7.2), which will undo the static argument transformation, and therefore only the advantage of increasing inlining opportunities would apply. The G machine [Joh83], for example, needs lambda lifting, as it cannot handle local function definitions. We will return to this point later in the chapter, and for the moment we will assume this is not the case (as in the STG machine).

The reduction in the number of free variables in the closures inside the function definition may have a much greater impact in heap usage than one may initially suspect in implementations in which the size of a closure is related to the number of free variables, like the STG. Let us analyse our example in more detail. `v`'s closure in the example has `{f,z,as}` as free variables, but after the transformation it has only `{foldr',as}`. The extra closure after the transformation (`foldr'`), will be allocated for every call to `foldr` (`foldr'`, has `{f,z,foldr'}` as free variables). But in the recursive calls we use less heap, as `v`'s closure is smaller.

Let us compare two different patterns of calls to the `foldr` function:

- If we called `foldr` 100 times with a list of 5000 elements one might think it would use less heap without the local definition, since there was 1 less closure in its definition (`foldr'`). It indeed performs 100 less heap allocations because of that. But as the size of `v`'s closure is 4 bytes bigger (in the Glasgow Haskell Compiler) due to the extra free variable, the original version allocates 2Mb of extra heap (100 calls $\times$ 5000 elements $\times$ 4 bytes), although doing a smaller number of allocations (100 less).
- now suppose we make 500000 calls to `foldr`, but the list happens to be empty for all these calls. In this case we will be paying the cost for the allocation of `foldr'` 500000 times, which gives us 500000 more heap allocations, increasing the heap consumed by 500000 $\times$ size of `foldr'` closure. Since no recursive calls occur (due to the lists being empty), we are paying all this cost and saving nothing. Actually there is even the cost for the extra call to `foldr'`.

The advantages of the transformation as one can notice by the above example are very dubious, as they will vary from program to program. We will discuss this again when we present our measurements of the transformations' effect in section 7.1.2.

A few other important observations are:

- the heap usage change does not happen when we have one static argument (like in map) as the closures will have the same number of free variables (assuming the transformation occurs in the top level):

```
map f bs = case bs of
             (a:as) -> let v = f a
                           w = map f as
                       in v : w
==>
map f bs = let map' bs = case bs of
                           (a:as) -> let v = f a
                                         w = map' as
                                     in v : w
           in map' bs
```

The closure for `w` had `f` and `as` as free variables (as top level definitions like `map` are not counted as free variables) and after the transformation the closure for `w` contains `map'` and `as` as free variables.

If the definition of `map` was a local definition (and therefore `map` *would* be counted as a free variable) we would be already reducing the number of free variables by one.

- The change in the number of free variables only occurs if the recursive call is done inside another closure. For tail calls there is no such a change:

```
f a b = case a of
          0 -> b
          n -> let v = n - 1 in f v b
==>
f a b = let f' a = case a of
                     0 -> b
                     n -> let v = n - 1 in f v
        in f' a
```

Although `b` is static there is no change in the free variables since no closures are built for the recursive call.

7.1.1 The algorithm

The algorithm in the case of a single recursive binding proceeds as follows:

- Record the name of the λ bound variables in the function right hand side.

- For every recursive call of the binder record whether this call repeats any arguments in the same place as they were in the function definition.

- For all arguments which are static (same position) in the recursive calls we may define a local recursive function which uses such arguments as free variables. This definition's right hand side is the original right hand side with calls to the original definition replaced by calls to the new definition with the static arguments removed, and the body of this newly introduced `letrec` is a call to the new recursive function with the same arguments it received less the static ones. At this point the original definition is not recursive anymore.

Example:

$$
\begin{array}{l}
f\ v_1 \ldots \boxed{v_k} \ldots v_n\ \texttt{=}\ \ldots f\ a_1 \ldots \boxed{v_k} \ldots a_m \ldots \\
\Longrightarrow \\
f\ v_1 \ldots \boxed{v_k} \ldots v_n \\
\quad \texttt{= let}\ b\ f'\ v_1 \ldots v_{k-1}\ v_{k+1} \ldots v_n\ \texttt{=}\ \ldots f'\ a_1 \ldots v_{k-1}\ v_{k+1} \ldots a_m \ldots \\
\qquad \texttt{in}\ f'\ v_1 \ldots v_{k-1}\ v_{k+1} \ldots v_n
\end{array}
$$

For partial applications of `f` the same can be applied as far as the static argument is still passed as an argument.

Now the definition of `f` is not recursive, and the criterias used to decide whether to inline or not a non-recursive definition can be applied to it.

For mutually recursive functions the same can be individually applied to each binding, but the functions are still mutually recursive. Advantage can be taken by the fact that the definitions that have been transformed are not *self* recursive[1] anymore, therefore they may be inlined. The transformation should be applied to one binding, then the binding is (possibly) inlined (in the other bindings) and then the transformation applied again. This is because, after inlining, the other functions may become *self* recursive again.

The best way to perform this transformation for sets of mutually recursive functions is probably by doing abstract interpretation to keep track of which arguments are static. In the following example the two last arguments of `g` and `h` are static, but to find that one has to keep track of the names of the arguments in the recursive calls:

```
f a = let g a b c = b + h (a-1) b c
          h d e f = d + g (d-1) e f
      in g 5 a a + h 6 a a
```

[1]By *self recursive* we mean that the binder occurs in its own right hand side. The function still is obviously *recursive* as other functions in the same mutually recursive set call it.

One could end up with a definition like

```
f a = let g a b c = let g a = b + h (a-1)
                        h d = d + g (d-1)
                    in g a
          h d e f = let g a = b + h (a-1)
                        h d = d + g (d-1)
                    in  h d
      in g 5 a a + h 6 a a
```

which introduces a lot of code duplication. For our purposes we believe that the risk of code explosion is not worth the gains from doing the transformation for sets of mutually recursive functions. For other purposes, like improving strictness analysis as discussed in the next section, this might be worthwhile.

7.1.2 Results

For the `nofib` programs we first tried to perform the static argument transformation with any number of static arguments ("*always*" column in Table 7.1). The results were not very promising, because most of the opportunities for the transformation were for functions with one static argument, as we can see in Table 7.2, and in these cases the gain from reducing the number of arguments in the recursive call by one was probably not enough to compensate for the extra closure allocated and the extra call.

We then decided to restrict the static argument transformation to cases were we had two or more static arguments, as in these cases the potential gains are bigger. The results, shown in the column labelled "2+" in the same table, present the improvements in instructions executed, although the results for heap allocated were mixed. We knew this could be the case, since we are indeed always creating an extra closure. Restricting the static argument transformation even more (only doing it if we have three or more static arguments) reduces the improvement, and therefore is too restrictive, as shown by the column labelled "3+" in Table 7.2.

Unfortunately this transformation does not seem to improve many programs. But it was quite a surprise that it could have such a significant effect in any of the benchmark programs at all, as we never expected many instances of it to be present in programs. It is also a very simple and cheap transformation to perform, therefore it might be a good idea to have it available in an optimising compiler.

static argument transformation Total Instructions Executed				
	static argument transf.			
program	never	always	2+	3+
treejoin	1.00	0.90	0.90	1.00
comp_lab_z	1.00	0.96	0.96	1.00
genfft	1.00	1.02	0.98	1.00
mandel2	1.00	0.98	0.98	1.00
listcompr	1.00	1.03	0.99	1.00
listcopy	1.00	1.02	0.99	1.00
wang	1.00	1.01	0.99	1.00
boyer	1.00	1.01	1.00	1.00
cichelli	1.00	0.96	1.00	1.00
clausify	1.00	1.03	1.00	1.00
compress	1.00	1.02	1.00	1.00
event	1.00	1.01	1.00	1.00
exp3_8	1.00	1.04	1.00	1.00
fft2	1.00	1.01	1.00	1.00
fluid	1.00	0.99	1.00	1.00
gen_regexps	1.00	0.98	1.00	1.00
hidden	1.00	1.01	1.00	1.00
ida	1.00	1.01	1.00	1.00
infer	1.00	0.99	1.00	1.00
knights	1.00	0.96	1.00	1.00
maillist	1.00	1.01	1.00	1.00
minimax	1.00	1.03	1.00	1.00
multiplier	1.00	1.01	1.00	1.00
parstof	1.00	0.99	1.00	1.00
primes	1.00	1.01	1.00	1.00
prolog	1.00	1.01	1.00	1.00
queens	1.00	1.05	1.00	1.00
reptile	1.00	1.03	1.00	1.00
rewrite	1.00	1.01	1.00	1.00
sched	1.00	1.02	1.00	1.00
transform	1.00	1.02	1.00	1.00
typecheck	1.00	1.04	1.00	1.00
veritas	1.00	1.03	1.00	1.00
wave4main	1.00	1.01	1.00	1.00
solid	1.00	1.01	1.01	1.01
11 other progs.	1.00	1.00	1.00	1.00
Minimum	-	0.90	0.90	1.00
Maximum	-	1.05	1.01	1.01
Geom. mean	-	1.00	1.00	1.00

static argument transformation Total Heap Allocated				
	static argument transf.			
program	never	always	2+	3+
multiplier	1.00	0.94	0.93	1.00
treejoin	1.00	0.97	0.97	1.00
genfft	1.00	1.03	0.99	1.00
wang	1.00	1.01	0.99	1.00
boyer	1.00	1.03	1.00	1.00
boyer2	1.00	1.00	1.00	1.00
cichelli	1.00	1.15	1.00	1.00
clausify	1.00	1.14	1.00	1.00
compress	1.00	1.00	1.00	1.00
event	1.00	1.05	1.00	1.00
fft2	1.00	1.02	1.00	1.00
fluid	1.00	1.02	1.00	1.00
hidden	1.00	1.03	1.00	1.00
hpg	1.00	1.01	1.00	1.00
ida	1.00	1.02	1.00	1.00
infer	1.00	1.01	1.00	1.00
knights	1.00	1.08	1.00	1.00
lift	1.00	1.02	1.00	1.00
listcompr	1.00	1.05	1.00	1.00
listcopy	1.00	1.04	1.00	1.00
maillist	1.00	0.97	1.00	1.00
minimax	1.00	1.08	1.00	1.00
parser	1.00	1.01	1.00	1.00
pretty	1.00	1.02	1.00	1.00
prolog	1.00	1.07	1.00	1.00
queens	1.00	1.30	1.00	1.00
reptile	1.00	1.05	1.00	1.00
rewrite	1.00	1.07	1.00	1.00
sched	1.00	1.06	1.00	1.00
sorting	1.00	1.01	1.00	1.00
transform	1.00	1.04	1.00	1.00
typecheck	1.00	1.13	1.00	1.00
veritas	1.00	1.03	1.00	1.00
wave4main	1.00	1.04	1.01	1.01
comp_lab_z	1.00	1.06	1.04	1.00
parstof	1.00	1.05	1.05	1.05
solid	1.00	1.07	1.07	1.07
9 other progs.	1.00	1.00	1.00	1.00
Minimum	-	0.94	0.93	1.00
Maximum	-	1.30	1.07	1.07
Geom. mean	-	1.04	1.00	1.00

Table 7.1 static argument transformation: instructions executed and bytes allocated

programs	Static arguments				
	1	2	3	4	5
`exp3_8`	3				
`queens`	1				
`event`	5	1	1		
`fft`	3	1			
`genfft`	3	1			
`ida`	3	3	1		
`listcompr`	1	1			
`listcopy`	1	1			
`parstof`	8	3	1		
`sched`	3				
`solid`	3	2			1
`typecheck`	3	1			
`wang`	1	1			
`comp_lab_zift`	8	4	1		
`transform`	15	1			
`wave4main`		1			
`boyer`	2				
`cichelli`	8				
`clausify`	2	1			
`knights`	4	1			
`mandel`		4			
`mandel2`	1	1			
`minimax`	2				
`multiplier`	10	4	1		
`pretty`	1				
`rewrite`	9	2			
`sorting`	3				
`treejoin`	3	1			
`compress`	1	1			
`fluid`	1				
`gg`	18				
`hidden`	1	1			
`hpg`	3			1	
`infer`	2				
`lift`	6				
`parser`	2	1			
`prolog`	3				
`reptile`	7	4			
`rsa`	1	1			
`veritas`	30	8	2		2
6 other programs	0	0	0	0	0

Table 7.2 Static argument count

7.1.3 Related work

The static argument transformation is similar to the analysis/transformation described in [CD91] for deciding when high order arguments can be effectively removed by transforming function definitions and then specialising the functions. In fact what he describes consists on conditions to decide whether the recursive high order argument could be eliminated by inlining (unfolding) the definition and then folding. As we rely only on inlining, as folding is a rather more complicate and expensive process to automate, we initially apply a transformation on the original function to (possibly) expose opportunities for inlining/specialisation. This achieves similar results, although with slightly more restricted applications.

The way static argument transformation can be used to remove high order arguments from functions is (whenever possible) by transforming the functions with high order arguments using the static argument transformation (the high order arguments are kept in the non-recursive part) and then inlining *all* the non-recursive functions with high order arguments. This can be used by a strictness analyser to reduce high orderness of the code being analysed, and therefore get better analysis results. Although for a compiler we have seen that we cannot remove the recursion (and therefore benefit from) the transformation for sets of mutually recursive function, a strictness analyser does benefit from that.

This procedure is used in [Sew94] to increase the scope of first-order analyses, and thus the number of programs for which a first-order analysis gives useful results.

A transformation that at first sight seems very similar to the static argument transformation is *lambda dropping* [Dan95], but a closer look shows that there are major differences between what we and [Dan95] do. [Dan95] *always* starts with a lambda lifted program, and is concerned about restoring the block structure of programs, therefore he does not introduce definitions that were not in the original program in first place, but only *restores* the original structure.

Appel independently suggested the same transformation for the SML-NJ compiler [App94], with the aim of helping in the inlining of recursive functions. He also points out the advantages for the purpose of removing invariants from a loop (similar to the effect we achieve with full laziness), and presents some other benefits related to closure representations and register allocation in his compiler. He achieves an average of 5% improvement over 10 programs, with a maximum of 11%.

7.2 Lambda lifting

Lambda lifting is a transformation that eliminates free variables from function definitions by passing them as arguments. After this is done, as the functions do not have free variables anymore, they can be "lifted" to the top level [Joh85, Hug82]. Therefore after lambda lifting all the function definitions are in the top level, in the form of supercombinators [Hug82]. This is an essential transformation for some implementations of functional languages in which all function definitions have to be in the top level, like the G-machine [Joh83].

In the STG machine lambda lifting is not needed, but is there any advantage in performing lambda lifting in the context of the STG machine?

The answer to this question, as we will see, is *sometimes yes, to save closure allocation.* A simple example would be:

```
f x z = let g y = y * x
        in case x of
             1 -> g z
```

```
          n -> let x' = x - 1
                   z' = g z
               in f x' z'
```

in this example the closure for the function `g` is allocated at each iteration of `f`. A lambda lifted version would be:

```
$g x y = y * x
f x z = case x of
          1 -> $g x z
          n -> let x' = x - 1
                   z' = $g x z
               in f x' z'
```

in which the free variables of the function `g` (in this case `x`) is abstracted, that is, becomes an argument, and then as `g` has no remaining free variables it can be moved to the top level. All calls to `g` now needs an extra argument for the abstracted variable. In this case we only allocate `g` once, although we have two possible disadvantages to take in consideration:

- □ the free variables in closures that mention the lifted function are modified. Actually we get an extra free variable for each abstracted variable (unless they were already free variables of that closure) less one: the lifted function, which due to the fact that it is now a top level declaration is not counted as a free variable anymore. In our particular example the closure for `z'` now has `x` and `z` as free variables, as opposed to `g` and `z` before.
- × one extra argument has to be passed at each call to `g`.

But there are cases in which the disadvantages far outweight the advantages, and therefore we should not do the transformation:

- × we may have to abstract *too many* variables in which the number of extra arguments and possible increase in the size of closures would often be more costly than the savings form allocating the function closure once.
- × if a variable occurs in an argument position we will not gain anything, since we will still have to create a closure for the partial application, e.g.:

```
f x z = let g y = y * x              $g x y = y * x
        in case x of                 f x z = let g' = $g x
            1 -> (g, g z)                    in case x of
            n -> let           ===>              1 -> (g', $g x z)
                   x' = x - 1                    n -> let
                   z' = g z'                              x' = x - 1
                  in f x' z'                              z' = $g x z
                                                         in f x' z'
```

We had to create a new `let`-binding (`g'`) that is left in place of `g`, therefore we will not save a closure allocation. Actually, since `g'` is a partial application (which is an updatable closure) this will make it even worse in performance.

× a similar problem with partial applications occurs if the function occurs on its own, as is sometimes the case in the STG machine, where we have to `let`-bind all lambda expressions:

```
f x z = ... let g y = y * x       ===>    $g x y = y * x
            in g                          f x z = ... $g x
```

Although we do eliminate the local definition, we end up creating a new partial application (`$g x`), which may cause the function to actually execute slower than before.

When experimenting with lambda lifting we noticed that if we did not restrict the lambda lifting in these three aspects, we were getting worse results with lambda lifted code than with non-lambda lifted code. This leads us to conjecture that implementations that do have to perform lambda lifting, and therefore cannot be selective like we can, may actually pay a heavy penalty in performance.

But we also have another difficulty in performing lambda lifting in our implementation: it does the opposite of the static argument transformation, which we discussed in the previous section and found to be a useful transformation. Therefore one may think that if lambda lifting is to be performed the advantages of the static argument transformation should not apply, as lambda lifting would undo that transformation.

Actually, as we will discuss in Section 7.3, by being even more selective on which functions we lambda lift we can still keep the benefits from the static argument transformation, e.g. by lambda lifting only non-recursive functions. For the rest of this section we will ignore the interaction of the static argument transformation and lambda lifting and analyse the effect of performing lambda lifting on its own.

In summary, our *selective* lambda lifter has the following effect:

× It increases the number of arguments passed in the recursive calls. This means that more arguments are pushed in the stack at each function call.

✓ It increases the possibility of inlining the function (if it is non-recursive), since the size of its body is reduced by removing any local function definitions.

□ It may increase the number of free variables for the closures which contain calls to functions that are being lifted. Actually the increase occurs if the function being lifted had more than one free variable, as if it had only one this would be compensated by the fact that after the lifting the function becomes a top level variable and therefore will not be a free variable anymore. This effect does not occur if the calls are tail calls, as there will not be a closure containing the call.

✓ It removes closures for *all* the local functions that have now moved to the top level. This means that these closures will be created only once (in the top level) and shared. If a function definition (f) occurred inside another function (g) which was called n times, this would save n allocations of the function (f) closure.

Just like in many other transformations, one can easily get examples which behave better or worse with lambda lifting. Therefore only by measurements one might get to the conclusion of whether on average it is a good idea or not to perform lambda lifting.

Algorithms for performing lambda lifting are well known and presented for example in [Joh85], [Pey87], [PL91b], therefore we will not present an algorithm here.

7.2.1 Results

We measured the effect of lambda lifting on the `nofib` benchmark programs with the criteria of abstracting up to 4 variables. The number of instances of lambda lifting opportunities to abstract more than 4 variables is very small, as show in Table 7.3, and we also start getting diminishing returns beyond that point. Table 7.4 presents the effects of lambda lifting, including an "always lambda lift" option, showing the importance of having a selective lambda lifter. The other columns show the effects of our selective lambda lifter, varying then maximum number of abstracted variables allowed.

Selective Lambda Lifting										
	abstracted arguments									
	0	1	2	3	4	5	6	7	8	9
non-recursive functions	1	7	3	-	-	-	2	-	-	1
recursive functions	25	187	100	23	13	7	3	2	2	-

Table 7.3 Selective lambda lifter: count distribution

One can see the disadvantage of always lambda lifting, as we expected. But the effect of our selective lambda lifting was quite disappointing.

7.3 Combining static argument transformation and lambda lifting

Although the static argument transformation and lambda lifting are seemingly incompatible transformations, we have seen that for each of them there are programs that can be improved by applying these transformations. We also know that by applying one after the other will undo its effect, *unless we are selective enough to avoid this interference*, by lambda lifting only those definitions whose performance is improved by so doing. Presumably such definitions were not created by the static argument transformation, since if so the latter transformation would have made things worse.

We performed various experiments in which we perform the static argument transformation and then tried to make the lambda lifter selective enough to avoid undoing the effects of the static argument transformation. Initially we were too selective, disallowing the lambda lifting of recursive functions, for example, which removed virtually all the benefits of lambda lifting in the programs in the `nofib` benchmark, although keeping the benefits of the static argument transformation.

We eventually decided to use our selective lambda lifter with an extra restriction: only lambda lift recursive functions if we are going to abstract only one argument. This way we will not be interfering with the recursive functions introduced by the static argument transformation, which we know are improving the code.

The overall effect was again quite small, but we managed to get some extra benefit from selectively lambda lifting after the static argument transformation, as show in Table 7.5.

lambda lifting Total Instructions Executed						
	lambda lifting					
			selective			
program	off	always	≤ 1	≤ 2	≤ 3	any
`lift`	1.00	1.21	0.99	0.99	0.99	0.98
`prolog`	1.00	1.31	1.00	0.99	0.98	0.98
`hidden`	1.00	1.01	1.00	1.00	1.00	0.99
`infer`	1.00	1.04	1.00	1.00	0.99	0.99
`parser`	1.00	1.26	1.00	1.00	1.00	0.99
`queens`	1.00	0.99	1.00	1.00	0.99	0.99
`typecheck`	1.00	1.18	1.00	0.99	0.99	0.99
`boyer`	1.00	1.01	1.00	1.00	1.00	1.00
`compress`	1.00	1.04	1.00	1.00	1.00	1.00
`fft`	1.00	1.17	1.00	1.00	1.00	1.00
`fft2`	1.00	1.17	1.00	1.00	1.00	1.00
`fluid`	1.00	1.06	1.00	1.00	1.00	1.00
`gen_regexp`	1.00	1.24	1.00	1.00	1.00	1.00
`genfft`	1.00	1.03	1.00	1.00	1.00	1.00
`gg`	1.00	1.11	1.00	1.00	1.00	1.00
`hpg`	1.00	1.07	1.00	1.00	1.00	1.00
`ida`	1.00	1.16	1.00	1.00	1.00	1.00
`knights`	1.00	1.48	1.00	1.00	1.00	1.00
`maillist`	1.00	1.25	1.00	1.00	1.00	1.00
`mandel`	1.00	1.14	1.00	1.00	1.00	1.00
`minimax`	1.00	1.09	1.00	1.00	1.00	1.00
`parstof`	1.00	1.37	1.00	1.00	1.00	1.00
`primes`	1.00	1.32	1.00	1.00	1.00	1.00
`reptile`	1.00	1.02	1.00	1.00	1.00	1.00
`rewrite`	1.00	1.37	1.00	1.00	1.00	1.00
`solid`	1.00	1.04	1.00	1.00	1.00	1.00
`sorting`	1.00	1.09	1.00	1.00	1.00	1.00
`veritas`	1.00	1.06	1.00	1.00	1.00	1.00
`wave4main`	1.00	1.15	1.00	1.00	1.00	1.00
`cichelli`	1.00	1.08	1.00	1.00	1.00	1.01
16 progs.	1.00	1.00	1.00	1.00	1.00	1.00
Minimum	-	0.99	0.99	0.99	0.98	0.98
Maximum	-	1.48	1.00	1.00	1.00	1.01
Geom. mean	-	1.09	1.00	1.00	1.00	1.00

lambda lifting Total Heap Allocated						
	lambda lifting					
			selective			
program	off	always	≤ 1	≤ 2	≤ 3	any
`cichelli`	1.00	0.90	1.00	1.00	1.00	0.89
`prolog`	1.00	1.02	1.00	0.95	0.91	0.93
`typecheck`	1.00	1.06	1.00	0.96	0.96	0.96
`hidden`	1.00	0.98	1.00	0.99	0.99	0.97
`lift`	1.00	1.17	0.98	0.98	0.98	0.97
`queens`	1.00	0.97	1.00	1.00	0.97	0.97
`boyer2`	1.00	0.98	1.00	1.00	1.00	0.98
`fluid`	1.00	1.01	1.00	0.99	0.98	0.98
`infer`	1.00	1.10	1.00	0.99	0.98	0.98
`clausify`	1.00	0.99	1.00	0.99	0.99	0.99
`event`	1.00	0.99	1.00	1.00	0.99	0.99
`knights`	1.00	1.00	1.00	0.99	0.99	0.99
`reptile`	1.00	1.00	1.00	0.99	0.99	0.99
`rewrite`	1.00	1.14	1.00	1.00	0.99	0.99
`transform`	1.00	0.99	1.00	0.99	0.99	0.99
`wave4main`	1.00	1.10	1.00	0.99	0.99	0.99
`boyer`	1.00	1.01	1.00	1.00	1.00	1.00
`compress`	1.00	1.05	1.00	1.00	1.00	1.00
`fft2`	1.00	1.02	1.00	1.00	1.00	1.00
`gen_regexp`	1.00	1.13	1.00	1.00	1.00	1.00
`gg`	1.00	1.03	1.00	1.00	1.00	1.00
`hpg`	1.00	1.05	1.00	1.00	1.00	1.00
`ida`	1.00	1.03	1.00	1.00	1.00	1.00
`mandel`	1.00	1.01	1.00	1.00	1.00	1.00
`minimax`	1.00	1.01	1.00	1.00	1.00	1.00
`parser`	1.00	1.08	1.00	1.00	1.00	1.00
`parstof`	1.00	1.04	1.00	1.00	1.00	1.00
`solid`	1.00	1.03	1.00	1.00	1.00	1.00
`veritas`	1.00	1.04	1.00	1.00	1.00	1.00
`fft`	1.00	1.02	1.00	1.00	1.00	1.01
`maillist`	1.00	1.14	1.00	1.00	1.00	1.01
`sorting`	1.00	1.13	1.00	1.00	1.01	1.01
14 progs.	1.00	1.00	1.00	1.00	1.00	1.00
Minimum	-	0.90	0.98	0.95	0.91	0.89
Maximum	-	1.17	1.00	1.00	1.01	1.01
Geom. mean	-	1.03	1.00	1.00	0.99	0.99

Table 7.4 lambda lifting: instructions executed and bytes allocated

static argument transformation and lambda lifting Total Instructions Executed			
program	SAT off LL off	SAT 2+ LL off	SAT 2+ LL on
`treejoin`	1.00	0.90	0.90
`comp_lab_zift`	1.00	0.96	0.95
`genfft`	1.00	0.98	0.97
`mandel2`	1.00	0.98	0.98
`lift`	1.00	1.00	0.99
`listcompr`	1.00	0.99	0.99
`listcopy`	1.00	0.99	0.99
`parser`	1.00	1.00	0.99
`prolog`	1.00	1.00	0.99
`typecheck`	1.00	1.00	0.99
`wang`	1.00	0.99	0.99
`cichelli`	1.00	1.00	1.01
`solid`	1.00	1.01	1.01
33 other progs.	1.00	1.00	1.00
Minimum	-	0.90	0.90
Maximum	-	1.01	1.01
Geom. mean	-	1.00	0.99

static argument transformation and lambda lifting Total Heap Allocated			
program	SAT off LL off	SAT 2+ LL off	SAT 2+ LL on
`cichelli`	1.00	1.00	0.89
`multiplier`	1.00	0.93	0.93
`prolog`	1.00	1.00	0.96
`typecheck`	1.00	1.00	0.96
`treejoin`	1.00	0.97	0.97
`boyer2`	1.00	1.00	0.98
`genfft`	1.00	0.99	0.98
`lift`	1.00	1.00	0.98
`clausify`	1.00	1.00	0.99
`comp_lab_zift`	1.00	1.04	0.99
`fluid`	1.00	1.00	0.99
`hidden`	1.00	1.00	0.99
`infer`	1.00	1.00	0.99
`knights`	1.00	1.00	0.99
`listcompr`	1.00	1.00	0.99
`listcopy`	1.00	1.00	0.99
`reptile`	1.00	1.00	0.99
`rewrite`	1.00	1.00	0.99
`transform`	1.00	1.00	0.99
`wang`	1.00	0.99	0.99
`wave4main`	1.00	1.01	1.00
`parstof`	1.00	1.05	1.05
`solid`	1.00	1.07	1.07
23 other progs.	1.00	1.00	1.00
Minimum	-	0.93	0.89
Maximum	-	1.07	1.07
Geom. mean	-	1.00	0.99

Table 7.5 static argument transformation and lambda lifting: instructions executed and bytes allocated

7.4 Conclusion

The effects of the two transformations presented in this chapter were quite surprising, as they were exactly the opposite of what we were expecting when we started experimenting with them:

- the static argument transformation, which we initially did not expect to have much impact on the performance of programs, turned out to be quite important for some of the programs in our benchmarks.
- for lambda lifting we already expected that to get some benefits we would need to do it selectively. But we eventually got two unexpected results: first the negative impact of un-selective lambda lifting is much bigger than we first suspected; and second the effect of our selective lambda lifter was quite disappointing.

This leads us to conclude that implementations techniques that depend on lambda lifting are probably paying a heavy penalty for that, and furthermore cannot take any benefit from performing the static argument transformation.

On the other hand, implementations that do not need lambda lifting probably would not benefit much from lambda lifting, even selectively. They could benefit from having the static argument transformation as an optimising transformation.

Chapter 8

Related work

In this chapter we describe how program transformations are typically used in functional languages. We also compare the transformations presented in this thesis with program transformations used in other compilers, including lazy and strict functional languages' compilers, as well as imperative languages' compilers.

8.1 Programmer-assisted program transformation

The term "program transformation" is often used to describe a program development technique, in which one starts from a clear but inefficient specification (or program) and by the use of semantics preserving source-to-source program transformation one gets to a more obscure but fast program. As an intermediate step of the process the program may even become less efficient. The gains obtained by using these techniques are usually big, sometimes even changing the time and/or space complexity of the program.

Much of the work on program transformation in functional languages is based on the work on fold/unfold transformations by Burstall and Darlington [DB76, BD77]. But these (usually semi-automatic) systems are quite dependent on programmer assistance and often need an *eureka* step, that is dependent on the specific program that one is trying to transform.

Although many tools for using these techniques have been developed (e.g. [Fea82, Fir90]), they are intended to be assisted by the programmer, and therefore cannot be regarded as automatic program transformation tools.

As we are only concerned with automatic program transformations in this thesis, we will not discuss these non-automatic methods any further.

8.2 Automatic program transformations

Automatic program transformations, which are the ones we are interested in, can be fully automated and therefore possibly incorporated into compilers. The gains are usually not as big as the ones from non-automatic methods, usually improving the programs by small constant factors. Many of the code optimisation techniques of imperative languages' compilers can be seen as automatic program transformations [ASU87].

Other characteristics that distinguish this approach from the non-automatic one are:

- the transformation process is not "creative": the system can only use the transformation rules it knows about;
- the sequence in which transformations are applied in predefined;
- the improvements are often small in the sense that a transformation rarely changes the complexity of a program, but improves it by a small constant factor;

Some more recent work tries to describe the entire process of compilation by successive program transformations [Kel89, Kra88, FLM91, App92]. The source language is translated to an intermediate language based on the lambda calculus, which is then transformed up to the point where it can be run on the target machine. One of the advantages of this process is that the correctness of the compiler comes from using only simple source-to-source transformations, which can be shown to be correct. Efficient output comes from using many transformations to simplify the program during the compilation process. Actually, some of the work in this area uses the compilation by transformation approach exactly for the purpose of obtaining not only efficient compilation, but also to prove the correctness of the compilation process by proving correct the individual transformations e.g. [Wan82, FLM91].

8.3 Program transformations in functional languages' compilers

Program transformation is often extensively used in the process of compiling functional languages. It is mainly used in the following contexts:

- when compiling functional languages it is a standard technique to transform the source language to a subset of it, which is still a functional language, although much simpler. This subset is often an enriched *lambda calculus* [Chu41, Wad71, Bar84]. This process of simplifying the language is called *desugaring*. It transforms out some of the syntactic constructs of the language that can be expressed in terms of other simpler constructs. The process of desugaring is often described as a source-to-source transformation [HMM86, Pey87]. Examples of such transformations are
  - compilation of pattern matching [Aug85, Wad87]
  - compilation of list comprehensions [Aug87, Pey87]
  - compilation of overloading [PJ93, Aug93].
- another way in which program transformation is used during the compilation process occurs when the program needs to be transformed in order to be compiled using a given implementation technique. Examples of this kind of transformation are
  - the lambda lifting transformation [Hug82, Joh85, PL91b], which is necessary when compiling a program to supercombinator form [Hug82], e.g. when compiling for the G-machine [Joh85].
  - Continuation Passing Style (CPS) translation [Kra88, FLM91, App92].
- other automatic program transformations are used solely to improve the efficiency of functional programs, and are therefore optional to the compilation process. The program transformations we discuss in this thesis fall into this category. Examples of such program transformations are:

  Deforestation. A source of inefficiency of functional languages is that the style of programming it advocates results in the creation and traversal of intermediate data structures during the evaluation of a program. Trying to improve on this, techniques to avoid creating and traversing these intermediate data structures have been researched. One of these techniques is *deforestation*, which is an automatic transformation to eliminate intermediate data structures from a program [Chi90, Wad90, GLP93, Gil95].

Transformations based on strictness information. An important optimisation for lazy functional languages is the transformation of call-by-need (lazy) to call-by-value (strict). This is only possible with the use of *strictness analysis* [Myc81], which gives information on what expressions can be evaluated strictly (more efficient) and still keep the same semantics. The use of the information obtained by strictness analysis can also be presented as program transformations [PP93, HB93].

High order removal. The removal of high-order functions is also another place where source-to-source transformations are used [CD91]. The goal this time is not only to improve efficiency, but also to improve the efficacy of other transformations or analysis techniques, like *strictness analysis*.

Unboxed values. Being able to express unboxed values in the intermediate language [PL91a] makes possible that some optimisations that are usually regarded as code-generation optimisations to be expressed as program transformations.

Full laziness. Full laziness tries to increase the sharing of data in a program, therefore reducing the number of times an expression is re-evaluated. It is described in [Hug82, Hug83, PL91b], and we also discuss it in Chapter 5.

Small local transformations. Apart from these transformations, there are many simple ones that are widely used by various functional languages' compilers. These usually consist of simple identities that allow a less efficient expression to be replaced by a more efficient one. The transformations we describe in Chapter 3 fall in this category.

8.4 Lazy functional languages' compilers

In this section we compare the transformations we use with the ones used in some other lazy functional languages' compilers.

8.4.1 The Chalmers LML/HBC compiler

The optimisations performed by the Chalmers LML/HBC compiler described in [Aug87] are:

- constant folding;
- β-reduction;
- dead code removal;
- `case` reduction;
- inlining of functions occurring just once;
- `case` of `case` (implemented in the code generator).

We perform all of these transformations. We do not know of any analysis on the effects of these transformations in this compiler.

8.4.2 The FAST compiler

Some optimisations used on the FAST (Functional programming on ArrayS of Transputers) compiler (developed at the University of Southampton) are presented in [HGW91]. The optimisations are:

- CAF lifting: this transformation amounts to the full laziness transformations (Section 5.2), but restricted to only float CAFs to the top level, which is precisely one of the things we try to avoid, due to the risk of space leaks (Sections 5.2.2 and 5.2.3).
- Specialisation: creates specialised version of high order functions, trying to improve strictness analysis results. We could achieve the same effect by using the static argument transformation for functions with high order functions, and later inlining these functions.
- Inlining: the criteria used for inlining is not to inline in an argument position, but no details are given on whether there are other criteria based on the size of the expression being inlined, or the number of occurrences.
- Strictness analysis: to save closure allocation/updates.
- Cheap eagerness: also to reduce closure allocation and updates.
- Boxing Analysis: similar to what is done by the worker-wrapper transformation.

The analysis of the effects of the optimisations are based on 5 programs, and concludes that they benefit most from strictness analysis and boxing analysis.

8.4.3 The Stoffel compiler

In [Bee93] the transformations used in the Stoffel [Bee92] compiler are presented and analysed. They include the ones from the FAST compiler described above, plus what we call "case merging" (Section 3.3.3), and he uses a different function for performing inlining decisions.

His benefits come mostly from inlining and CAF lifting.

8.5 Strict functional languages' compilers

In this section we compare the optimisations used in a state-of-the-art strict functional language compiler (SML-NL) [App92] with the ones present in a lazy functional language compiler. The use of the SML/NJ Compiler also allows us to compare our approach to the use of CPS (Continuation Passing Style) for optimisation and code generation, as is the case of SML/NJ.

These transformations are a superset of the ones presented in other works on CPS-based compilation (e.g. [KKR+86, Kra88, Kel89])

8.5.1 Continuation passing style

Continuation-passing style (CPS) is a program notation that makes every aspect of control flow and data flow explicit.

Here is an example of the translation of a program into CPS style, taken from [App92]:

```
prodprimes n = case n of
                 1 -> 1
                 _ -> case isprime n of
                        True -> n * prodprimes (n-1)
                        False -> prodprimes (n-1)
```

This function computes the product of all primes less than or equal to a positive integer `n`. The translation into CPS gives us the following program:

```
prodprimes n c = case n of
                   1 -> c 1
                   _ -> let k b = case b of
```

```
                        True -> let j p = let a = n * p
                                          in c a
                                    m = n - 1
                                in prodprimes m j
                        False -> let h q = c q
                                     i = n - 1
                                 in prodprimes i h
                in isprime n k
```

In this program `c`, `k`, `j` and `h` are *continuation functions*, i.e. they express "what to do next", turning what was the return from a function call into simply another function call.

For more details on CPS one could refer to [App92] where the technique is described as well as how its characteristics are exploited in the SML/NJ compiler.

[FSDF93] shows that the benefits of compilation using CPS can be obtained by using some source-to-source transformations, which he calls *A reductions*, which name intermediate results. He then shows the equivalence of the two compilation strategies and claims that the language of *A-normal forms* is a good intermediate representation for compilers. The Core language we use is very similar to the *A-normal form*, as we always name closures using `let`s. This means that our approach should be able to achieve the same benefits of compilation using CPS.

As we will see, most of the optimisations done in the CPS are similar to the ones we use when optimising by program transformation. The approach is similar to ours in general terms:

- the process iterates up to the point where very few optimisations are performed. This is a consequence of the fact that each transformation may expose more opportunities for other transformations.
- the optimisations are carefully chosen so that their interaction do not incur in non-termination, which is possible if one transformation is followed by another that turns the code back to the way it was.
- the optimisations make extensive use of heuristics, and try to use inexpensive analysis techniques, so that they will not impose much overhead in compilation time.

- the process relies on small optimisations that interact to produce more complex optimisations.

In this section we will compare some of the optimisations described by Appel [App92] with the ones we use in the Glasgow Haskell Compiler.

8.5.2 β-contraction

This consists of inlining functions used only once, therefore exposing opportunities for β-reductions to take place. In the Glasgow Haskell Compiler we do the same as part of our inlining strategy.

8.5.3 `case` reduction

There are two instances of this transformation in [App92]:

- constant folding of `SWITCH` operator, which eliminates `SWITCH`es (`case`s) when it is scrutinising a known value; and
- constant folding of `SELECT`s from known records: whenever a variable which is statically bound to a record is the operand of a selection operation, the expression can be eliminated and directly replaced by the selected field of the record.

These are similar to our `case` reduction transformation:

```
let v = C1 a1 a2
in ... case v of            ==> let v = C1 a1 a2
         C1 v1 v2 -> E1         in ...E1[a1/v1,a2/v2]
         ...
```

8.5.4 Dead variable elimination

Removes unused variables (bindings) from the program. In SML, due to the strictness of the language and its non-functional extensions, this optimisation has to be careful not to remove code that modifies the store or raises an exception, as it would be evaluated even if there were no references to the variable. For lazy languages it is sufficient that there are no references to the variable. This is discussed in section 8.6.3.

8.5.5 Argument flattening

The argument flattening optimisation improves the way arguments are passed to functions. Functions with arguments passed in a tuple are modified so that the tuple constructor itself is not built or scrutinised. This is achieved whenever all calls to the function pass an explicit tuple, that is, explicitly mention the tuple constructor. The effect is similar to what one would get by the following transformation is a lazy language:

```
let f (a,b) = ...f (v1,v2)... ==> let f a b = ...f v1 v2...
in ...f (v3,v4)...                in ...f v3 v4
```

Due to the semantics of lazy pattern matching we cannot always guarantee in a lazy language that the tuple argument will be "unboxed", that is, whether any of its components will be needed. Therefore we cannot remove the tuple constructor and directly pass the arguments. If (due to strictness analysis) we can be sure that the tuple argument will be evaluated we can perform a similar transformation.

This may be regarded, in a restricted way, as similar to avoiding extra boxing and unboxing operations as described in [PL91a] (worker-wrapper transformation).

8.5.6 Dropping unused arguments

This is a slightly more complicated instance of dead variable elimination. It removes function arguments that are not used in the function body from the argument list, and consequently removes the respective arguments in the call sites. The worker-wrapper transformation [PL91a] handles this transformation in the Glasgow Haskell Compiler.

8.5.7 β-expansion

β-expansion is β-contraction of functions called more than once, that is, inlining functions used more than once in their call sites, trying to expose more local optimisations. Due to the (possible) code duplication this is done by heuristically selecting which functions should be expanded (inlined).

Inlining was discussed in Chapter 6, therefore we will not repeat here the issues discussed in that chapter.

8.5.8 η-reduction

The SML compiler performs η-reduction:

$$\texttt{f a b c = g a b c} \quad \Longrightarrow \quad \texttt{f = g}$$

The Glasgow Haskell Compiler does not perform eta reduction explicitly, but for such simple functions our inlining strategy will choose to inline them, achieving in most cases[1]) together with β-reduction the same result.

8.5.9 Uncurrying

The uncurrying transformation tries to transform curried functions into functions that receive tuples as arguments, as SML can treat this more efficiently:

$$\texttt{f a b c = ...} \quad \Longrightarrow \quad \texttt{f (a,b,c) = ...}$$

This can be done whenever all calls to the function passes the number of arguments it requires, therefore there are not partial applications of the function. In a lazy language we have no gain in doing this transformation, as we would in fact introduce an extra constructor (the tuple) to be matched.

This may look exactly the opposite transformation to the one we described in Section 8.5.5. The difference is that in Section 8.5.5 we know that all calls to the function pass a tuple as argument, therefore it is explicit that there are no partial applications. Here we would expect that there could be a partial application of the function, and the implementation of these partial applications using CPS is inefficient. Therefore what is intended is to get to the uncurried version so that flattening 8.5.5 can be eventually applied.

This is a major difference in the two approaches, as in lazy functional languages currying is extensively used and therefore must be supported efficiently by the underlying model of evaluation. In SML it is more efficient to use tuples and therefore the transformation is worthwhile. In lazy functional languages one would rather use the opposite transformation, avoiding the use of the (tuple) constructor whenever possible. This would be a valid transformation whenever the argument was strict (guaranteed to be evaluated), due to the semantics of (lazy) pattern matching.

[1]whenever the call to `f` is saturated, that is, has as many arguments as its arity.

8.5.10 Hoisting

Hoisting tries to move bindings to reduce or expand the scope of individual definitions. The Glasgow Haskell Compiler achieves the same effect with the local `let` floating transformation and with full laziness, where inner lets are floated to increase their scope and (possibly) expose opportunities for other transformations:

```
let f x = let v = ...         let v = ...
          in ...v...     ==>  in let f x = ...v...
in ...f...f...                   in ...f...f...
```

After the transformation `v` will not be evaluated every time `f` is called, as in the original definition. Actually, SML would not float `v` out of a lambda, because then `v` would be evaluated regardless of `f` being entered. This is true only in a strict functional languages, as in a lazy language although we would be allocating a closure for `v`, we would only evaluate it if it is ever used. In SML one may float `let`s out of `let`s and out of applications. It is also possible to float `let`s out of single branch `case`s, but one has to be careful that the `let` being floated does not have any side-effecting expressions, that could affect the program behaviour if performed in a different order.

Another possibility is to hoist downwards, for example, if a definition is used in a single branch it could be floated to that branch only:

```
let v = ...                   if c then let v = ...
in if c then ...v...v...  ==>           in ...v...v...
        else ...                   else ...
```

in a strict context this would avoid the evaluation of `v` when the condition `c` was false. In a lazy context this would only save the allocation of the closure for `v`, since it would only be evaluated if needed.

As SML is a strict language, many opportunities for hoisting can be taken which are only be possible in a lazy functional language in the presence of a strictness information, since they are valid only in a strict context, e.g.:

```
f (case v of (a,b) -> a)    ==> case v of (a,b) -> f a
```

as if `f` did not use its first argument `v` would not be evaluated in the first expression, but would be evaluated in the second one (in a lazy context).

Here again many of the complications of the algorithm for hoisting are due to the impure characteristics of SML, like assignment and exceptions, which introduce the need for extra restrictions when hoisting.

8.5.11 Common subexpression elimination

The risks of common subexpression elimination in a functional language are discussed in the section related to the same optimisation in imperative languages (Section 8.6.1). SML overcomes part of the problem by looking for common subexpressions only when one expression dominates the other, that is, it is inside the scope of the other. This way it would find common subexpressions like

```
let v = [1..1000]
in sum v + sum [-1000..1] + prod [1..1000]
```

$\Longrightarrow$

```
let v = [1..1000]
in sum v + sum [-1000..1] + prod v
```

but would not try to get a common subexpression out of the following code:

```
sum [1..1000] + sum [-1000..1] + prod [1..1000]
```

A space leak may still occur, as before the common subexpression elimination the space used by `v` could be reclaimed after the evaluation of `sum v` and be (possibly) reused when evaluating `sum [-1000..1]`. After the transformation it can only be reclaimed after `prod v` is evaluated.

We have not investigated this transformation, but similarly to what we do for the full laziness transformation, the risk of space leak can be reduced by restricting the types of expressions that are commoned up.

8.5.12 Closure conversion

This transformation turns free variables in closures into arguments. This is identical to lambda lifting [Joh85] in lazy functional languages.

8.5.13 Effect of the transformations

Appel [App92] analysed the effect of the transformations used in the New Jersey SML Compiler, in which he found that the most important ones were the inlining of functions called only once, dead variable elimination and `case` reduction (actually "constant folding of `SELECT`s from known records").

8.6 Imperative languages' compilers

Optimisations for imperative languages can be divided into three categories:

- local transformations. Many of the optimisation techniques in imperative languages take part in the so called *Basic Blocks* and are referred to as Local Transformations, as they use local context information. A Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end.
- global transformations. Global optimisations use data-flow analysis to extend the local optimisations to a global context as well as introduce a few more optimisations. These extra optimisations, as we will see, are mostly related to optimising loops and procedure calls.
- peephole optimisations. If we go closer to code generation we get to a set of very local and specialised transformations, called peephole optimisation. Here again we can easily find similarities with procedures on optimising functional languages.

Some optimisations can be included in more than one of these categories, as many local transformations, for example, can be extended to be applied using global context information. When this is the case we will discuss it only once, commenting on possible differences if used as a local or global transformation.

In this section we will discuss some optimisations in each of these classes. Most of them are extensively described in [ASU87].

8.6.1 Common subexpression elimination

Common subexpression elimination tries to locate places where the same expression is used more than once in a basic block, and eliminate these multiple evaluations.

```
a = b + c            v = b + c
d = b - c    ==>     a = v
e = b + c            d = a - c
f = e + c            e = v
                     f = e + c
```

In the example above `b + c` is a common subexpression as it is assigned to `a` and to `e`. Therefore instead of recomputing it, we can compute it once (and assign it to a new variable `v`) and replace its occurrences directly by that variable. Later, as we will see, this code can be further optimised via copy propagation.

The optimisation is more complicated than it might initially seem, as one has to check whether any of the variables in the common subexpression is modified between the occurrences. When optimising between basic blocks (global optimisation) it is even harder to keep track of whether the two common expressions can be eliminated, due to multiple entry points to the blocks.

On the functional world we do not have the notion of assignment and therefore it is much easier to keep track of whether an expression is a common subexpression or not, as the "values" (bindings) of a variable do not change. In fact assuming we have unique names in the program (no name is used more than once), which is often the case in later stages of the compilation of functional languages, whenever the same syntactical expression occurs it can be regarded as a common subexpression, therefore it becomes much easier to detect one when it occurs.

On the other hand, common subexpression elimination in functional languages might not always be good, as it might drastically change the space behaviour of a program by causing so called "space leaks". These might be introduce whenever a large data structure becomes shared due to common subexpression elimination, and therefore its space which before was reclaimed by garbage collection now cannot be reclaimed until its last reference is used. To illustrate the problem the following program creates three times a list with 1000 elements, which after being used by `prod` (product of a list) and `sum` (sum of a list) has its space immediately reclaimed, therefore it could (possibly) use the same space when creating the three lists.

```
sum [1..1000] + sum [1000..2000] + prod [1..1000]
```

If `[1..1000]` is regarded as a common subexpression one could transform this program to:

```
let l = [1..1000]
in sum l + sum [1000..2000] + prod l
```

and in this case the space allocated for the list will not be available when evaluating `sum [1000..2000]`, but can only be reclaimed after the evaluation of `prod l` is over[2].

[2]assuming left to right evaluation of the sum.

8.6.2 Copy propagation

Copy propagation consists of eliminating the assignment of values in a variable to other variables, by substituting occurrences of the latter by the former, as in the example below:

`v = b + c`		
$\boxed{\texttt{a = v}}$		`v = b + c`
`d = a - c`	$\Longrightarrow$	`d = v - c`
$\boxed{\texttt{e = v}}$		`f = v + c`
`f = e + c`		

The optimisation presents some difficulties to be used in a global framework, that is, between basic blocks. The problems arise due to the fact that if (for example) there was a jump to the `e = v` statement from a different point in the program (it would not be a single basic block anymore) then it could possibly not be true that `f` would get the right value after the transformation. A similar transformation can be used in the functional framework, without the extra difficulties presented above, as there is no notion assignment in (pure) functional languages:

```
let v = b + c          let v = b + c
in let a = v     ==>   v + b
   in a + b
```

8.6.3 Dead code elimination

This tries to locate portions of the code that cannot be accessible during the program execution, and can therefore be removed. For example:

$$\begin{array}{rl}
 & S_1 \\
 & \texttt{goto } L_1 \\
L_2: & S_2 \\
L_1: & S_3
\end{array} \quad\Longrightarrow\quad \begin{array}{rl}
 & S_1 \\
 & \texttt{goto } L_1 \\
L_1: & S_3
\end{array}$$

If there are no jumps to L_2 then it cannot be reached, as the instruction before it is an unconditional jump. Therefore it can be removed.

Similarly in functional languages sometimes a definition is not used at all, therefore it can be removed in the same way:

$$\begin{array}{l}\texttt{let v} = \texttt{E} \\ \texttt{in a} + \texttt{b}\end{array} \Longrightarrow \texttt{a} + \texttt{b}$$

since `v` is not used it can be removed.

8.6.4 Algebraic transformations

Algebraic transformations use algebraic properties of operators to replace more expensive computations by less expensive ones. This includes, for example, the use of identity properties for `+` and `*` operators in expressions like `x + 0` and `x * 1` to replace them by `x`. Also, the optimisation called constant folding falls in this category, and consists of eliminating some run-time computations of operations on constants by their results, e.g. replacing `5 + 3` by the constant `8`.

As the same properties are valid in functional languages, it can be similarly used in the functional framework.

8.6.5 Code motion

Code motion tries to remove invariant computations out of loops, therefore avoiding its recomputation for every iteration. The following transformation presents the basic concept:

`for i := 1 to 10 do begin`
`x :=` $\boxed{\text{fib (20)}}$;
`k := k + x`
`end`
	$\Longrightarrow$	`x :=` $\boxed{\text{fib(20)}}$;
		`for i := 1 to 10 do begin`
		`k := k + x`
		`end`

Clearly computing the value of `x` does not depend on any variable modified in the loop, therefore it need not be computed for every iteration, but can be computed only once before the loop executes, which considerably reduces the overhead for every iteration.

In the functional framework iterations are done using recursion, and a similar transformation that removes invariant computation out of loops is the full laziness transformation (Section 5.2). Similarly it finds out computations which do not depend on the variables used in the recursion and floats these computations out of the loop. One example of a transformation that would be carried out by full laziness transformation is:

`let`
` f v = if v = 11`
`       then 0`
`       else f (v + 1) +` $\boxed{\text{fib 20}}$
`in f 1`
	$\Longrightarrow$	`let`
		` x =` $\boxed{\text{fib 20}}$
		` f v = if v = 11`
		`       then 0`
		`       else f (v + 1) + x`
		`in f 1`

Here similarly after the transformation `fib 20` is only computed once. One of the drawbacks of this transformation is that we may again have a space leak, like in the common subexpression elimination. It would arise, in this case, if the "value" floated outside of the loop is a structure which allocates a large amount of memory, which would only be freed in the end of the loop, and would be recomputed for every iteration otherwise. Again, this is basically a space/time trade off, since on has the option of computing the value only once and keeping longer the space used by it or recomputing it and reclaiming the space sooner.

8.6.6 Loop unrolling

Loop unrolling consists of reducing the number of iterations a loop executes and trying to get some local optimisations on the unrolled code and (possibly) delay jump instructions (specially useful for pipelined machines). In the following example we see the effect of unrolling the loop once, therefore halving the number of iterations.

```
                                for i := 1 to 10 step 2 do begin
for i := 1 to 10 do   ==>         x := x * i;
  x := x * i                      x := x * (i + 1)
                                end
```

In functional languages a similar effect is obtained by inlining recursive definition, therefore reducing the number of recursive calls. An example of that is the definition below, which is inlined (unrolled) once.

```
                              fact x
fact x                          = if x < 1
  = if x < 1           ==>        then 1
    then 1                        else x * (if (x-1) < 1
    else x * fact (x-1)                     then 1
                                            else (x-1) * fact (x-2))
```

8.6.7 Procedure inlining

Procedure inlining consists of heuristically selecting some (usually small) procedures to be inlined, that is, every call to the procedure is replaced by the actual code of the procedure. This aims to save time by eliminating the overhead of these procedure calls and increasing the opportunity for other optimisations, as the procedure code is now exposed to local context information and therefore to more optimisations. Of course this must be done to specific and small procedures, since excessive inlining can easily lead to a large increase in code size due to code duplication.

In the functional framework this idea is similarly used in the concept of inlining function definitions. There is the same risk of code explosion due to excessive code duplication, but done in a controlled way similar benefits can be obtained, as opportunities for local optimisation should appear.

8.6.8 Procedure cloning

Procedure cloning is quite similar to procedure inlining, but tries to reduce the code duplication by sharing the code. Instead of inlining the procedure whenever it is called, one tries to match characteristics of different call sites and generate specialised versions of the procedures, for example, for different arguments [CHK92]. The idea uses similar techniques used in partial evaluation.

In functional programming the techniques of partial evaluation can also be applied. The technique can also be used, for example, to reduce the extra overhead imposed by overloading in languages like Haskell. In this case, different versions of functions can be generated for specific contexts (types) in which they are used. This technique is discussed in [SP92], where whenever a function is used always with the same context (type) it is replaced by a specific (non-overloaded) version of the function.

8.6.9 Redundant instruction elimination

Redundant instruction elimination tries to avoid redundant loads and stores to memory locations of data that is or could be kept in a register. In a sequence of instructions like:

```
store R0 mem
load  R0 mem
```

which stores the contents of register `R0` to a memory location and then loads the value of the same memory location in the same register, clearly we can eliminate the second instruction, since the register already contains the data.

We sometimes achieve a similar effect by avoiding redundant boxing and unboxing operations, with the `case` reduction transformation. Consider the expression `x + x`, which in Core language becomes:

```
case x of
  MkInt a# -> case x of
```

```
                MkInt b# -> case (a# +# b#) of
                              r# -> MkInt r#
```

Since we are unboxing `x` twice, we could remove the extra unboxing of `x` and obtain a more efficient version:

```
case x of
  MkInt a# -> case (a# +# a#) of
                r# -> MkInt r#
```

8.6.10 Flow of control optimisation

Flow of control optimisation is a peephole optimisation which, for example, tries to optimise jump instructions whose destinations are also jump instructions.

$$\begin{array}{rl} & \texttt{goto}\ L_1 \\ & \vdots \\ L_1: & \texttt{goto}\ L_2 \end{array} \quad\Longrightarrow\quad \begin{array}{rl} & \texttt{goto}\ L_2 \\ & \vdots \\ L_1: & \texttt{goto}\ L_2 \end{array}$$

in the functional framework a similar optimisation can be achieved directly by copy propagation, for example, or combining η-reduction and copy propagation. In a definition like

$$\begin{array}{l} \texttt{let f x = g x} \\ \texttt{in ...f...f...} \end{array} \Longrightarrow \begin{array}{l} \texttt{let f = g} \\ \texttt{in ...f...f...} \end{array} \Longrightarrow \texttt{...g...g...}$$

it is clear that the calls to `f` will simply add an extra indirection level to a call to `g`. η-reduction gives us the first transformation, and copy propagation does the rest.

Chapter 9

A Cost Semantics

A way of proving a transformation's correctness is by presenting expressions before and after the transformation is applied and then showing that the two forms are semantically equivalent, e.g. using denotational semantics [Sch86].

But for the class of transformations we are interested in (code improving program transformations) we would like to prove not only that the transformations are correct, but also that they are indeed improving the code.

All the transformations we have presented were suggested by our intuitions about what constitutes an optimisation. Although we have discussed and measured the effects of the transformations we presented, we would like to have a more abstract and implementation independent way of *proving* that we are indeed reducing (or maintaining) the evaluation costs. Ideally this formal framework for *reasoning* about optimising transformations should:

- be *abstract* enough to be tractable;
- be *concrete* enough to model sharing and the cost of evaluating expressions.

In this chapter we use a natural (operational) semantics for the lazy lambda calculus (based on the one presented in [Lau93]), extended with the notion of *cost*, to perform such proofs for some of the transformations we presented in Chapter 3. We define a *cost relation* and show examples of transformations that preserve or reduce costs.

An important property of such an relation would be that it is *contextual*, i.e. that if two expressions e_1 and e_2 are related then, under any arbitrary context $C[-]$, $C[e_1]$ is related to $C[e_2]$. We do not provide such a proof for our cost relation here, as we later found out that such a proof is directly related to a a known open problem in general [PS93]. We discuss this in Section 9.2.3. Nonetheless we believe that the idea of associating costs to a semantics in the form we suggest is a useful tool in understanding the efficiency aspects of transformations.

9.1 A cost semantics

The cost semantics we use is based on the natural semantics presented in [Lau93], extended with the notion of costs associated with the rules. This allows us not only to prove the correctness of the transformations we present, but also prove that a transformation preserves, reduces or increases the cost of evaluating an expression.

We believe these notions of costs are abstract enough not to be restricted to a specific implementation technique, but apply to lazy functional languages in general.

Judgements have the form:

$\Gamma : e \Downarrow_n \Theta : z$

meaning that under a heap Γ (binding variables to expressions) the expression e reduces, with cost n, to a heap Θ and a weak head normal form expression z.

The main difference from the original semantics is the notion of cost (annotation on $\Downarrow$), which is incremented when a particular rule is applied. The costs we use are:

- A, the cost of using the application rule.
- V, the cost of evaluating a variable.
- U, the cost of an update.
- L, the cost of allocating a closure in the heap.
- C, the cost of evaluating a case expression.
- O, the cost of a basic operation.

One can argue that some of these costs are too abstract. Indeed some of them may actually vary according to many factors (e.g. the cost of a closure allocation may depend on the number of free variables of the closure), but one of our aims is exactly to have an *abstract* notion of costs! We also believe that one can easily make these costs more concrete and still use the semantics to reason about the effects of program transformations.

$$\Gamma : C_k\ v_1 \ldots v_l\ \Downarrow_0\ \Gamma : C_k\ v_1 \ldots v_l \qquad \textit{Constructor}$$

$$\Gamma : \lambda x.e\ \Downarrow_0\ \Gamma : \lambda x.e \qquad \textit{Lambda}$$

$$\frac{\Gamma : e_1\ \Downarrow_m\ \Delta : z_1 \quad \Delta : e_2\ \Downarrow_n\ \Theta : z_2}{\Gamma : e_1 \oplus e_2\ \Downarrow_{m+n+O}\ \Theta : z_1 \oplus z_2} \qquad \textit{Basic Operation}$$

$$\frac{\Gamma : e\ \Downarrow_m\ \Delta : \lambda y.e' \quad \Delta : e'[x/y]\ \Downarrow_n\ \Theta : z}{\Gamma : e\ x\ \Downarrow_{m+n+A}\ \Theta : z}\ \textit{App} \qquad \textit{Application}$$

$$\frac{\Gamma : e\ \Downarrow_n\ \Delta : z}{(\Gamma, x \mapsto e) : x\ \Downarrow_{V+U+n}\ (\Delta, x \mapsto z) : \hat{z}}\ \textit{UVar} \qquad \textit{Updatable Variable}$$

$$\frac{\Gamma : e\ \Downarrow_n\ \Delta : z}{(\Gamma, x \mapsto e) : x\ \Downarrow_{V+n}\ (\Delta, x \mapsto e) : \hat{z}}\ \textit{NUVar} \qquad \textit{NonUpdatable Variable}$$

$$\frac{(\Gamma, x_1 \mapsto e_1) : e\ \Downarrow_n\ \Delta : z}{\Gamma : \texttt{let}\ x_1\texttt{=}e_1\ \texttt{in}\ e\ \Downarrow_{n+L}\ \Delta : z}\ \textit{Let} \qquad \textit{Let}$$

$$\frac{\Gamma : e\ \Downarrow_m\ \Delta : C_k\ v_1 \ldots v_l \quad \Delta : e_k[v_i/x_i]_{i=1}^{l}\ \Downarrow_p\ \Theta : z}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_1 \ldots x_{q_i}\ \texttt{->}\ e_i\}_{i=1}^{n}\ \Downarrow_{m+p+C}\ \Theta : z}\ \textit{Case} \qquad \textit{Case}$$

9.2 The cost relation $\lesssim_e$

Since call by need semantics is just a more efficient implementation of call by name semantics, the existing definitions of equivalence of expressions evaluated using call by name can be used directly to prove the correctness of program transformations in lazy functional languages.

But since call by need semantics has an inherent notion of efficiency, we need to make the cost of evaluation of an expression (or the *sharing* of evaluation, as in [MOTW95]) an observable property of any notion of equivalence.

Our goal is to establish a "less than" relation, $\lesssim_e$, between *semantically equivalent* expressions, whose intuitive meaning is

$$e_1 \lesssim_e e_2 \text{ iff evaluating } e_1 \text{ is less expensive than evaluating } e_2.$$

9.2.1 Observational cost relation

But what the relation $\lesssim_e$ actually means? Ultimately we want it to be an *observational cost relation*:

$e_1 \lesssim_{obs} e_2$ if for all closing boolean contexts $C[\,]$ and heaps Δ, $\Delta : C[e_1] \Downarrow_m \Delta' : \texttt{true}$ iff $\exists \Delta''.\Delta : C[e_2] \Downarrow_n \Delta'' : \texttt{true}$ and $m \leq n$.

This definition (without the condition the $m \leq n$) is used to define *observational equivalence* [RP94], on which this definition is based.

9.2.2 Direct cost relation

But the quantification over all contexts makes it very difficult to prove that two expressions are in the $\lesssim_{obs}$ relation. Therefore we seek a more direct definition of $\lesssim_e$. In this section we develop such a definition, and in Section 9.2.3 we discuss the question of proving that $\lesssim_e$ implies $\lesssim_{obs}$.

Of course the cost of evaluating e_1 and e_2 depends on the value of their free variables, so we start by defining that an expression e_1 is related to another expression e_2 if and only if, given an arbitrary heap, the two heap-expression pairs are related under a $\lesssim_{he}$ relation:

Definition 1

$$e_1 \lesssim_e e_2 \;\; \textit{if} \;\; \forall \Gamma . (\Gamma, e_1) \lesssim_{he} (\Gamma, e_2)$$

We then proceed to define the $\lesssim_{he}$ relation for the heap-expression pairs. The $\lesssim_{he}$ relation is co-inductively defined, based on similar definitions used for defining *applicative bisimulation* for pure functional languages [Gor93].

First we define a relation $\lesssim_{he}$ for heap-expression pairs in which the expression in in weak head normal form:

Definition 2

$$(\Delta_1, \lambda x.e_1) \lesssim_{he} (\Delta_2, \lambda x.e_2) \ \ \mathit{if} \ \ \forall e.((\Delta_1, x \mapsto e), e_1) \lesssim_{he} ((\Delta_2, x \mapsto e), e_2)$$

The definition above implicitly implies that $\mathit{dom}\ \Delta_1 = \mathit{dom}\ \Delta_2$, since we do not restrict e to be a closed expression (if the two domains are different it is enough to pick e to be a variable not in the intersection to make the definition to be false).

Definition 3

$$(\Delta_1, C\ a_1 \ldots a_n) \lesssim_{he} (\Delta_2, C\ b_1 \ldots b_n) \ \ \mathit{if} \ \ \forall i \in \{1, \ldots, n\}.(\Delta_1, a_i) \lesssim_{he} (\Delta_2, b_i)$$

Now we need a definition of $\lesssim_{he}$ relation when the expression is not in weak head normal form.

A failed attempt

Our first idea was to use a relation identical to the one often used when defining applicative bisimilarity, but extended with comparisons for the cost of the evaluation of the expressions:

Definition 4.a

$$(\Gamma_1, e_1) \lesssim_{he} (\Gamma_2, e_2) \ \ \mathit{if} \ \ \forall \Delta_1, \Delta_2, m, n, z_1, z_2. \left(\begin{array}{c} \Gamma_1 : e_1 \ \Downarrow_m \ \Delta_1 : z_1 \ \wedge \ \Gamma_2 : e_2 \ \Downarrow_n \ \Delta_2 : z_2 \\ \Longrightarrow \\ m \leq n \\ \wedge \\ (\Delta_1, z_1) \lesssim_{he} (\Delta_2, z_2) \end{array} \right)$$

Alas this definition is clearly not enough from what can be observed by comparing the two programs below:

```
let w = e
in \x -> w + 1

let w = e
in \x -> e + 1
```

clearly in a context where the lambda expression is shared and evaluated multiple times the first one is cheaper, since `e` will only be evaluated once. An example of such a context would be

```
let f = let w = e
        in \ x -> w + 1
in f 1 + f 2

let f = let w = e
        in \x -> e + 1
in f 1 + f 2
```

But using the relation we defined above this difference is not noticed if we look at the expressions out of context. Actually the first one is considered to be more expensive as it includes a variable lookup and update, while the latter does not. The difference only arises if the expression is shared.

A second attempt

To solve the problem presented above we introduce an alternative version of the cost relation for expressions, which basically demands not only the cost of evaluating the expressions to be related but that the resulting heaps are also related:

Definition 4.b

$$(\Gamma_1, e_1) \lesssim_{he} (\Gamma_2, e_2) \quad \textit{if} \quad \forall \Delta_1, \Delta_2, m, n, z_1, z_2. \left(\begin{array}{c} \Gamma_1 : e_1 \Downarrow_m \Delta_1 : z_1 \;\wedge\; \Gamma_2 : e_2 \Downarrow_n \Delta_2 : z_2 \\ \Longrightarrow \\ m \leq n \\ \wedge \\ (\Delta_1, z_1) \lesssim_{he} (\Delta_2, z_2) \\ \wedge \\ \Delta_1 \lesssim_h \Delta_2 \end{array} \right)$$

And the definition of the $\lesssim_h$ relation is:

$$\Delta_1 \lesssim_h \Delta_2 \quad \textit{if} \quad \forall e. (\Delta_1, e) \lesssim_{he} (\Delta_2, e)$$

Reexamining the example that failed with the previous definition:

```
let w = e
in \x -> w + 1

let w = e
in \x -> e + 1
```

We can notice that:

- The two expressions reduce to weak head normal form with the same cost (the cost of allocating the closure `w`).
- But the weak head normal form expressions are *not* related under $\lesssim_{he}$. We first add a binding from `x` to an arbitrary expression `e'` to the heap and then we check that the two subexpressions `w + 1` and `e + 1` are related under $\lesssim_{he}$ again. Here is where the new condition we introduced becomes important, since after the evaluation of the two expressions the two heaps will not be cost-related anymore: in one of them w will be evaluated while in the other it won't.

Therefore with the extra condition we are actually checking that the amount of evaluation performed on the heap was the same after each expression is evaluated to weak head normal form.

But is this definition correct? To establish this we would have to prove that our cost relation is an *observational cost relation*.

9.2.3 Observational cost relation revisited

As we mentioned before, the quantification over all contexts makes it very difficult to prove that two expressions are in the $\lesssim_{obs}$ relation. We would like to prove that $e_1 \lesssim_e e_2$ iff $e_1 \lesssim_{obs} e_2$.

This is not a new problem. It is very similar to the task of proving that applicative bisimulation is equivalent to observational equivalence [AO93, How89, Gor93]. This proof is known to be difficult, and uses a clever technique due to Howe [How89]. This technique is used by Sands [San93] for his *time analysis*, which is very similar to our cost semantics, but restricted to call by name semantics. There he also discusses the difficulties he found in trying to extend it to call by need semantics.

Alas our problem seems to be significant more difficult. The difficulties we found are related to difficulties described in [PS93] with observational properties in the presence of dynamically created names (which we use to model heaps). In [PS93], the presence of dynamically created local names (without bindings or updates, which we have) is shown to pose significant difficulties to establish observational equivalence. For his simple language, and using a more elaborated method to show observational equivalence, he shows the method to be complete for expressions of first order types, but incomplete at higher types.

Therefore we were not able to prove that $e_1 \lesssim_e e_2$ iff $e_1 \lesssim_{obs} e_2$, i.e. that our cost relation $\lesssim_e$ is the same as the $\lesssim_{obs}$ relation.

[RP94] uses a definition of applicative bisimilarity to prove full abstraction (i.e. contextual equivalence) for a translation between a lambda calculus with reference types and Standard ML. They uses an environment to model the references, but, since they are dealing with a strict language, the environment has no notion of updates or bindings to unevaluated expressions, like our heaps do. They prove that their applicative bisimilarity *implies* observational equivalence, since due to the presence of dynamically created local state the two notions do not coincide [PS93].

9.3 Some examples

In this section we present some proofs that can be obtained using the cost relation we introduced in the previous section.

9.3.1 `let` floating from application

The proof that expressions before and after the `let` floating from application transformation is applied keep the same cost and semantics is presented by showing that, starting from the same assumptions (i.e. the same heap), we get to the same resulting expression at the same cost, although using different reduction rules (or a different sequence of reduction rules) and starting with different expressions. Other proofs follow the same line[1].

Theorem: $(\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e)\ x \sim \texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e\ x$

Proof:

[1]We use $e_1 \sim e_2$ to mean $e_1 \lesssim_e e_2$ and $e_2 \lesssim_e e_1$.

$$\dfrac{\dfrac{(\Gamma, v \mapsto e_v) : e \Downarrow_m \Delta : \lambda y.e'}{\Gamma : \texttt{let } v \texttt{ = } e_v \texttt{ in } e \Downarrow_{m+L} \Delta : \lambda y.e'}\,\textit{Let} \quad \Delta : e'[x/y] \Downarrow_n \Theta : z}{\Gamma : (\texttt{let } v \texttt{ = } e_v \texttt{ in } e)\ x \Downarrow_{m+n+A+L} \Theta : z}\,\textit{App}$$

$$\dfrac{\dfrac{(\Gamma, v \mapsto e_v) : e \Downarrow_m \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_n \Theta : z}{(\Gamma, v \mapsto e_v) : e\ x \Downarrow_{m+n+A} \Delta : z}\,\textit{App}}{\Gamma : \texttt{let } v \texttt{ = } e_v \texttt{ in } e\ x \Downarrow_{m+n+A+L} \Theta : z}\,\textit{Let}$$

9.3.2 `case` floating from application

Theorem:

$(\texttt{case } e \texttt{ of } \{C_i\ v_1 \ldots v_q \texttt{ -> } e_i\}_{i=1}^n)\ x \sim \texttt{case } e \texttt{ of } \{C_i\ v_1 \ldots v_q \texttt{ -> } e_i\ x\}_{i=1}^n$

Proof:

$$\dfrac{\dfrac{\Gamma : e \Downarrow_m \Pi : C_k\ v_1 \ldots v_l \quad \Pi : e_k[v_i/x_i]_{i=1}^l \Downarrow_p \Delta : \lambda y.e'}{\Gamma : \texttt{case } e \texttt{ of } \{C_i\ x_1 \ldots x_q \texttt{ -> } e_n\}_{i=1}^n \Downarrow_{m+p+C} \Delta : \lambda y.e'}\,\textit{Case} \quad \Delta : e'[x/y] \Downarrow_o \Theta : z}{\Gamma : (\texttt{case } e \texttt{ of } \{C_n\ x_1 \ldots x_q \texttt{ -> } e_n\}_{i=1}^n)\ x \Downarrow_{m+p+o+C+A} \Theta : z}\,\textit{App}$$

$$\dfrac{\Gamma : e \Downarrow_m \Pi : C_k\ v_1 \ldots v_l \quad \dfrac{\Pi : e_k \Downarrow_p \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_o \Theta : z}{\Pi : (e_k\ x)[v_i/x_i]_{i=1}^l \Downarrow_{p+o+A} \Theta : z}\,\textit{App}}{\Gamma : \texttt{case } e \texttt{ of } \{C_n\ x_1 \ldots x_q \texttt{ -> } e_n\ x\}_{i=1}^n \Downarrow_{m+p+o+C+A} \Theta : z}\,\textit{Case}$$

9.3.3 `let` floating from `case` scrutinee

Theorem:

$$\texttt{case } (\texttt{let } v \texttt{ = } e_v \texttt{ in } e) \texttt{ of } \{C_i\ v_1 \ldots v_q \texttt{ -> } e_i\}_{i=1}^n$$
$$\sim$$
$$\texttt{let } v \texttt{ = } e_v \texttt{ in case } e \texttt{ of } \{C_i\ v_1 \ldots v_q \texttt{ -> } e_i\}_{i=1}^n$$

Proof:

$$\dfrac{\dfrac{(\Gamma, v \mapsto e_v) : e \Downarrow_m \Theta : C_k\ v_1 \ldots v_l}{\Gamma : \texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e \Downarrow_{m+L} \Theta : C_k\ v_1 \ldots v_l}\ \mathit{Let} \quad \Theta : e_k[v_i/x_i]_{i=1}^{l} \Downarrow_p \Delta : z}{\Gamma : \texttt{case (let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e\texttt{) of}\ \{C_n\ x_1 \ldots x_q\ \texttt{->}\ e_n\}_{i=1}^{n} \Downarrow_{m+p+A+L} \Delta : z}\ \mathit{Case}$$

$$\dfrac{\dfrac{(\Gamma, v \mapsto e_v) : e \Downarrow_m \Theta : C_k\ v_1 \ldots v_l \quad \Theta : e_k[v_i/x_i]_{i=1}^{l} \Downarrow_p \Delta : z}{(\Gamma, v \mapsto e_v) : \texttt{case}\ e\ \texttt{of}\ \{C_n\ x_1 \ldots x_q\ \texttt{->}\ e_n\}_{i=1}^{n} \Downarrow_{m+p+A} \Delta : z}\ \mathit{Case}}{\Gamma : \texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in case}\ e\ \texttt{of}\ \{C_n\ v_1 \ldots v_q\ \texttt{->}\ e_n\}_{i=1}^{n} \Downarrow_{m+p+A+L} \Delta : z}\ \mathit{Let}$$

9.3.4 Unboxing `let` to `case`

We want to prove the following:

If v is of a constructor type and e is strict in v then

$\texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e \gtrsim \texttt{case}\ e\ \texttt{of}\ C_k\ v_1 \ldots v_l\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ C_k\ v_1 \ldots v_l\ \texttt{in}\ e_k$

To be able to reason about the effect of this transformation we need to introduce a notion of an expression being strict on a variable, i.e. of a variable that is guaranteed (due to strictness analysis) to be demanded during the evaluation of an expression.

A possible definition of such a property is:

$$\textit{if } e \textit{ is strict in } v \textit{ then } (\Gamma, v \mapsto e_v) : e \Downarrow_l (\Delta, v \mapsto z_v) : z$$

From this definition we can infer an important fact about the reduction $(\Gamma, v \mapsto e_v) : e \Downarrow_l (\Delta, v \mapsto z_v) : z$: we have used the *UVar* rule, since that is the only way v could have been updated. From this fact we derive the following rule for a strict `let`:

If e is strict in v then

$$\dfrac{\Gamma : e_v \Downarrow_m \Delta_v : z_v \quad (\Delta_v, v \mapsto z_v) : e \Downarrow_n (\Delta_v, v \mapsto z_v) : z}{(\Gamma, v \mapsto e_v) : e \Downarrow_{m+n+U} (\Delta_v, v \mapsto z_v) : z}\ \mathit{SLet}$$

Although we do not have a formal proof that this rule is correct, the intuition behind it comes from a basic property of strictness: if we know that an expression is going to be evaluated we may evaluate the expression in advance (i.e. transforming call by need into call by value). Since we are not actually doing that (yet), but just using that identity, we still add the cost U for the update (assuming e_v is not in weak head normal form). If e_v is already in weak head normal form (i.e. it is already z_v) there is no extra U cost.

We then proceed to analyse the transformation we suggest:

$$\dfrac{\dfrac{\Gamma : e_v \Downarrow_p \Delta : C_k\ v_1 \ldots v_l \quad (\Delta, v \mapsto C_k\ v_1 \ldots v_l) : e \Downarrow_n \Theta : z}{(\Gamma, v \mapsto e_v) : e \Downarrow_{p+n+U} \Theta : z}\ \textit{SLet}}{\Gamma : \texttt{let}\ v\ \texttt{=}\ e_v\ \texttt{in}\ e \Downarrow_{p+n+U+L} \Theta : z}\ \textit{Let}$$

$$\dfrac{\Gamma : e_v \Downarrow_p \Delta : C_k\ v_1 \ldots v_l \quad \dfrac{(\Delta, v \mapsto C_k\ v_1 \ldots v_l) : e \Downarrow_n \Theta : z}{\Delta : \texttt{let}\ v\ \texttt{=}\ C_k\ v_1 \ldots v_l\ \texttt{in}\ e \Downarrow_{n+L} \Theta : z}\ \textit{Let}}{\Gamma : \texttt{case}\ e_v\ \texttt{of}\ C_k\ v_{k1} \ldots v_{kl}\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ C_k\ v_{k1} \ldots v_{kl}\ \texttt{in}\ e \Downarrow_{p+n+C+L} \Theta : z}\ \textit{Case}$$

In this case we have not ended up with the same resulting cost, but we can now have a condition under which this transformation would reduce the cost of the expressions: if $U \geq C$.

9.3.5 `let` floating from `let`

We want to show that:

If e is strict in v then $\texttt{let}\ v\ \texttt{=}\ (\texttt{let}\ w\ \texttt{=}\ e_w\ \texttt{in}\ e_v)\ \texttt{in}\ e \gtrsim \texttt{let}\ w\ \texttt{=}\ e_w ; v\ \texttt{=}\ e_v\ \texttt{in}\ e$

Case 1: v is demanded by the evaluation of e (i.e. e strict in v):

$$\dfrac{\dfrac{(\Gamma, w \mapsto e_w) : e_v \Downarrow_m \Delta_v : z_v}{\Gamma : \texttt{let}\ w\ \texttt{=}\ e_w\ \texttt{in}\ e_v \Downarrow_{m+L} \Delta_v : z_v}\ \textit{Let} \quad (\Delta_v, v \mapsto z_v) : e \Downarrow_n \Delta : z}{\Gamma : \texttt{let}\ v\ \texttt{=}\ (\texttt{let}\ w\ \texttt{=}\ e_w\ \texttt{in}\ e_v)\ \texttt{in}\ e \Downarrow_{m+L+n+L+U} \Delta : z}\ \textit{SLet}$$

$$\dfrac{\dfrac{(\Gamma, w \mapsto e_w) : e_v \Downarrow_m \Delta_v : z_v \quad (\Delta, v \mapsto z_v) : e \Downarrow_n \Delta : z}{(\Gamma, w \mapsto e_w, v \mapsto e_v) : e \Downarrow_{m+n+U} \Delta : z}\ \textit{SLet}}{\Gamma : \texttt{let}\ w\ \texttt{=}\ e_w ; v\ \texttt{=}\ e_v\ \texttt{in}\ e \Downarrow_{m+n+U+L+L} \Delta : z}\ \textit{Let}$$

Case 2: v is not demanded by the evaluation of e:

$$\frac{(\Gamma, v \mapsto \texttt{let } w \texttt{ = } e_w \texttt{ in } e_v) : e \ \Downarrow_m \ (\Delta, v \mapsto \texttt{let } w \texttt{ = } e_w \texttt{ in } e_v) : z}{\Gamma : \texttt{let } v \texttt{ = (let } w \texttt{ = } e_w \texttt{ in } e_v\texttt{) in } e \ \Downarrow_{m+L} \ (\Delta, v \mapsto \texttt{let } w \texttt{ = } e_w \texttt{ in } e_v) : z} \ \mathit{Let}$$

$$\frac{(\Gamma, w \mapsto e_w, v \mapsto e_v) : e \ \Downarrow_n \ (\Delta, w \mapsto e_w, v \mapsto e_v) : z}{\Gamma : \texttt{let } w \texttt{ = } e_w; v \texttt{ = } e_v \texttt{ in } e \ \Downarrow_{n+L+L} \ (\Delta, w \mapsto e_w, v \mapsto e_v) : z} \ \mathit{Let}$$

We have done some extra work after the transformation in the second case (L, since $m = n^2$). Therefore:

1. if the `let` is strict we keep the same cost;
2. if the let is ever evaluated we have the same result of a strict `let` (!), therefore we also keep the same cost;
3. if the `let` is never evaluated we have made the code worse.

Actually for 1 and 2 above there is yet another possibility: if e_v is in weak head normal form (i.e. it is already z_v), we end up saving the update cost U after the transformation, and therefore we are improving the code.

This is precisely what we stated when we described this transformation in Chapter 3!

9.3.6 `case` floating from `let`

Theorem:

$$\texttt{let } v \texttt{ = case } e_v \texttt{ of } \{C_i \ v_1 \ldots v_q \texttt{ -> } e_i\}_{i=1}^n \texttt{ in } e$$
$$\sim$$
$$\texttt{case } e_v \texttt{ of } \{C_i \ v_1 \ldots v_q \texttt{ -> let } v \texttt{ = } e_i \texttt{ in } e\}_{i=1}^n$$

e is strict in v, $v \notin fv \ e_v$ and $\{v_1, \ldots, v_q\} \bigcap fv \ e = \emptyset$.

[2]Actually, we have no formal proof that $m = n$, as the heaps have different bindings! This shows that ideally we would like to have a less restrictive definition for $\lesssim$.

$$\dfrac{\dfrac{\Gamma : e_v \ \Downarrow_m \ \Theta : C_k\ v_1 \ldots v_l \quad \Theta : e_k[v_i/x_i]_{i=1}^{l} \ \Downarrow_p \ \Delta_v : z_v}{\Gamma : \texttt{case}\ldots \ \Downarrow_{m+p+C} \ \Delta_v : z_v}\ \mathit{Case} \qquad (\Delta_v, v \mapsto z_v) : e \ \Downarrow_o \ \Delta : z}{\Gamma : \texttt{let}\ v\ \texttt{=}\ \texttt{case}\ e_v\ \texttt{of}\ \{C_n\ x_1 \ldots x_q\ \texttt{->}\ e_n\}_{i=1}^{n}\ \texttt{in}\ e \ \Downarrow_{m+n+C+p+L+U} \ \Delta : z}\ \mathit{SLet}$$

$$\dfrac{\Gamma : e_v \ \Downarrow_m \ \Theta : C_k\ v_1 \ldots v_l \qquad \dfrac{\Theta : e_k[v_i/x_i]_{i=1}^{l} \ \Downarrow_p \ \Delta_v : z_v \quad (\Delta_v, v \mapsto z_v) : e \ \Downarrow_o \ \Delta : z}{\Theta : (\texttt{let}\ v\ \texttt{=}\ e_k\ \texttt{in}\ e)[v_i/v_k i]_{i=1}^{l} \ \Downarrow_{p+o+L+U} \ \Delta : z}\ \mathit{SLet}}{\Gamma : \texttt{case}\ e_v\ \texttt{of}\ \{C_n\ x_1 \ldots x_q\ \texttt{->}\ \texttt{let}\ v\ \texttt{=}\ e_n\ \texttt{in}\ e\}_{i=1}^{n} \ \Downarrow_{m+p+U+o+C+L} \ \Delta : z}\ \mathit{Case}$$

9.4 Conclusions and future work

In this chapter we presented a definition for a cost semantics, together with a cost relation for a call by need language.

We presented the difficulties involved in obtaining a suitable definition of such a cost relation. This is caused by the inherent non-compositionality of such a definition for a semantics with dynamically created names, since two expressions which seem to be cost-related on their own may be shown not to be cost-related under certain contexts.

The cost-relation we suggest, although useful for reasoning about many transformations, may still be too restrictive due to the requirement that heaps should have the same bindings. It would be interesting to try to obtain definitions that relax this restriction. [RP94] had to introduce a similar restriction for their definition of equivalence for modelling a language with references.

It would be very useful to have a proof that our cost relation is a contextual cost relation. Unfortunately we have found that just the presence of dynamically created names (as we use to model heaps) already pose many difficulties to obtain such a proof, and is still an open problem in general. Since we not only dynamically create names, but also have bindings to these names and perform updates, we were not able to obtain such a proof.

Nevertheless we have shown that a cost semantics for the call by need lambda calculus is a useful way to assess the effects of program transformations in an abstract form, and this seems to be a promising area for future research.

Chapter 10

Conclusions

We have presented and systematically analysed a large set of local transformations, discussed their importance and measured their occurrence. We have also measured the effect of a number of them in a large set of programs. Although many of them do not achieve much on their own, when combined, these transformations interact in non-obvious ways to achieve major improvements in the performance of real programs. Fine tuning the local positioning of `lets`, is shown to be an important transformation that was not studied before.

Full laziness, a transformation that has been known for quite a while, was investigated in detail. We have shown that it can have a major effect in programs, and that the risks of space leaks that it creates are not only rare, but also can be greatly reduced.

We present the static argument transformation, which does the opposite of lambda lifting. It had far bigger effects on programs than we initially suspected, and turned out to be an important transformation to have in an optimising compiler.

We have shown that not having to perform lambda lifting is an important feature of the STG machine, and claim that implementations that have to do it may be paying a significant performance penalty for that. We restricted lambda lifting to specific cases where it might be beneficial for the STG machine, but although we got some improvements in heap allocation this was not reflected in improvements in instructions executed. We proceeded to combine this selective lambda lifting with the static argument transformation, but that did not get any major improvements.

We have also presented the effects of inlining, showing that we quickly get diminishing returns from it, and therefore the optimal amount of inlining seems to be far smaller than one would initially suspect. We also did not have problems with excessive code duplication due to inlining.

The cost semantics we presented suggests an abstract way of relating program transformations with its effects in performance. This allows the effect of transformations to be formally studied independently from a particular implementation, and to formalise the notion of code improvement.

10.1 General conclusions

A substantial hidden benefit of performing the measurements presented in this thesis was the debugging and fine tuning of the transformations themselves, since more often than not we had one or two programs that instead of benefiting from a transformation were actually getting worse. This was often due to an obscure interaction with other transformation that was not obvious when we started to implement it, and that would probably go unnoticed if we were not working with such a large set of programs. Therefore it was very important to use such a large set of programs, and not small toy programs. One could easily get to the wrong conclusions by measuring the effects on small programs or in only a few programs.

It is also clear that one cannot obtain an optimal result for most of the transformations we presented, since one can create examples in which they would result in less efficient code. Of course this is also true (in a smaller scale) for many program transformations, even for imperative languages. Only by performing experiments in a reasonably large scale and with a diverse set of programs (as we did) one can actually decide whether they are on average worthwhile transformations.

We believe a lot of effort has been done on studying large scale transformations, and not much on the small local transformations, although these when combined can have just as big an effect as more complicated global transformations.

Another interesting observation from all our results is that sometimes a significant effect in heap allocation is not reflected on actual performance improvement. While performing experiments we have seen a program allocate 3 times more heap than another version, but still have better performance. This shows the importance of not relying on measuring the effect of transformations on heap allocation to predict its effect on execution time.

Although quite a few of the transformations presented result in a small *average* improvement, it is clear that all of them have a major impact in at least a few programs. Therefore a good optimising compiler should indeed perform all of them, as they are bound to have major effects in some programs.

The vast majority of the transformations presented can be used in any lazy functional language compiler. We believe should present effects similar to the ones we presented in this thesis.

10.2 Future work

There are some interesting topics that certainly deserve some further investigation:

- The use of linear type systems and update analysis [MTW95] should certainly help to reduce the number of updates performed and also help on inlining, as it would tell us which lambdas are entered only once, which would allow more inlining of expressions without any risks of work duplication. It can also provide useful information for the full laziness pass, avoiding that we `let`-bind expressions to be floated past a lambda that will not be shared (if the lambda is entered only once) and will actually create an overhead (extra closure). We have indeed seen cases where this happens.
- Reduce the number of iterations needed for the simplifier to reach a fixed point. This can probably be done using a more systematic approach, as we currently do it in a very *ad-hoc* manner. [AJ94] describes the approach used to minimise the number of iterations in a similar pass of the SML-NJ Compiler, as well as a linear time algorithm to perform it. The algorithm tries to keep track of the usage counts of variables (the occurrence information) during the simplification process, therefore reducing the number of iterations needed to reach a fix-point. Although the set of transformations performed is far smaller than the one in the Glasgow Haskell Compiler, a similar approach can probably be used in it.
- The set of transformations that we know are confluent and terminating should be extended.
- The static argument transformation can be improved so that the cases in which it shows improvement with 1 static argument can be selected. Important cases like, for example, the function `map` that has only one static argument, but can benefit from being transformed and inlined are being missed (this would improve strictness analysis in the place where it is used).
- The interaction between the static argument transformation and the selective lambda lifter can probably be improved, as we have not managed to combine the best results obtained by the two transformations.

- It is important to obtain a proof that our *cost relation* for our cost semantics in a *contextual cost relation*. It is also important to try to obtain a less restrictive cost relation, which would allow more programs to be comparable under it.

Appendix A

Some function definitions

In this appendix we present the definition of some data types and function definitions as they are used in the Glasgow Haskell Compiler.

A.1 Arithmetic

First we define the basic `Int` data type:

```
data Int = MkInt Int#
```

The `Int` data type is a *boxed* data type with a single constructor, `MkInt`, which has an *unboxed* `Int#` as an argument.

The basic functions over `Int`s are defined by first *unboxing* the arguments and then applying the *primitive* version of the function on the unboxed arguments:

```
(+) x y = case x of
            MkInt x# -> case y of
                          MkInt y# -> case x# +# y# of
                                        r# -> MkInt r#
```

In this case the `+#` is the primitive addition function, that works on `Int#`s.

The other operators (`+`, `-`, `*`, etc.) are similarly defined, using their unboxed counterparts. `Float`s and `Double`s are also implemented in a similar way.

A.2 Comparison

The `Bool` data type is a *boxed* data type:

```
data Bool = True | False
```

Currently there is no provision in the Glasgow Haskell Compiler for *unboxed* versions of such enumerated types. This, for efficiency reasons, lead us to have the *primitive* comparison operators (e.g. `>#`) returning *unboxed* integers (i.e. `Int#`) instead of `Bool`s:

```
(>) x y = case x of
            MkInt x# -> case y of
                          MkInt y# -> case x# ># y# of
                                        0# -> False
                                        1# -> True
```

Other comparison operators are similarly defined.

A.3 Boolean operators

Finally, below are the definitions of the `&&` (and), `||` (or) and `not` boolean operators:

```
(&&) x y = case x of
             True  -> case y of
                        True  -> True
                        False -> False
             False -> False

(||) x y = case x of
             True -> True
             False -> y

not x = case x of
          True  -> False
          False -> True
```

Bibliography

[AJ94] Andrew W. Appel and Trevor Jim. Making lambda calculus smaller, faster. Technical report CS-TR-477-94, November 1994. to appear in the Journal of Functional Programming.

[AO93] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, (105):159–267, 1993.

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App94] Andrew A. Appel. Loop headers in lambda-calculus or CPS. *LISP and Symbolic Computation*, 7(4):337–343, 1994.

[ASU87] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1987.

[Aug85] Lennart Augustsson. Compiling pattern matching. In Functional Programming Languages and Computer Architecture [Fun85], pages 368 – 381.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, November 1987.

[Aug93] Lennart Augustsson. Implementing haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73. ACM Press, 1993.

[Bar84] Henk P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. North Holland, 1984.

[BD77] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bee92] M. Beemster. The lazy functional intermediate language Stoffel. Technical Report CS-92-16, University of Amsterdam, December 1992.

[Bee93] Marcel Beemster. Optimizing transformations for a lazy functional language. In W.-J. Withagen, editor, *7th Computer systems*, pages 17–40, Eindhoven, The Netherlands, Nov 1993. Eindhoven Univ. of Technology.

[CD91] W. N. Chin and John Darlington. Removing higher-order expressions by program transformation, February 1991.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, London, March 1990.

[CHK92] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *IEEE Computer Society 1992 International Conference on Computer Languages*, pages 96–105, April 1992.

[Cho83] F. C. Chow. A portable machine-independent global optimizer — design and measurements. Technical Report 83–254, Stanford University, 1983.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[Dan95] Olivier Danvy. Lambda-dropping: transforming recursive equations into programs with block structure. Technical report DART 252, Computer Science Department, Aarhus University, Aarhus, Denmark, January 1995.

[DB76] John Darlington and Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[DH88] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software Practice and Experience*, 18:775–790, 1988.

[DH92] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[Fea82] M. S. Feather. A system for assisting program transformation. *ACM TOPLAS*, 4(1):1–20, January 1982.

[Fir90] M. A. Firth. *A Fold/Unfold Transformation System for a Non-Strict Language*. PhD thesis, University of York, December 1990.

[FLM91] M. Fradet and Daniel Le Metayer. Compilation of functional languages by program transformation. *ACM Trans. on Programming Languages and Systems*, 13(1), January 1991.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Languages Design and Implementation*, pages 237–247. ACM, 1993.

[Fun85] *Functional Programming Languages and Computer Architecture*, number 201 in LNCS, Nancy, September 1985. Springer-Verlag.

[Fun93a] *Functional Programming, Glasgow 1993*, Ayr, Scotland, 1993. Springer Verlag, Workshops in Computing.

[Fun93b] *Functional Programming Languages and Computer Architecture*, Copenhagen, June 1993. ACM Press.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[Gil95] Andrew J. Gill. *A Cheap Deforestation for Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1995.

[GLP93] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In Functional Programming Languages and Computer Architecture [Fun93b], pages 223–232.

[Gor93] Andrew Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, 1993. TR 285.

[Gor95] Andrew Gordon. Bisimilarity as a theory of functional programming. In *Mathematical Foundations of Programming Semantics*, New Orleans, March 1995. Elsevier Electronic Notes in Theoretical Computer Science, volume 1.

[Har94] P.H. Hartel. Benchmarking implementations of lazy functional languages ii: two years later. Technical report, Department of Computer Systems, University of Amsterdam, December 1994.

[HB93] D. B. Howe and Geoffrey L. Burn. Using strictness in the STG machine. In Functional Programming, Glasgow 1993 [Fun93a].

[HBH93] K. Hammond, Geoffrey L. Burn, and D. B. Howe. Spiking your caches. In Functional Programming, Glasgow 1993 [Fun93a].

[HGW91] Pieter Hartel, Hugh Glaser, and John Wild. On the benefits of different analyses in the compilation of a lazy functional language. In *Workshop on the Parallel Implementation of Functional Languages*, pages 123–146, Southampton, June 1991.

[HHaPW92] Cordelia Hall, Kevin Hammond, and Simon Peyton Jones amd Philip Wadler. Type classes in Haskell. Report, Department of Computing Science, Glasgow University, 1992.

[HL93] Pieter H. Hartel and Koen G. Langendoen. Benchmarking implementations of lazy functional languages. In Functional Programming Languages and Computer Architecture [Fun93b], pages 341–349.

[HMM86] R. Harper, D. McQuenn, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, University of Edinburgh, 1986.

[Hol90] Carsten K. Holst. Improving full laziness. In *Glasgow Workshop on Functional Programming*. Springer-Verlag, 1990.

[How89] D. J. Howe. Equality in lazy computation systems. In *Logic in Computer Science*, pages 198–203, 1989. IEEE Computer Society Press.

[Hug82] John M. Hughes. Super combinators: A new implementation method for applicative languages. In *ACM Conference on Lisp and Functional Programming*, pages 1–10, Pittsburg, 1982.

[Hug83] John M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

[Hug89] John M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[Joh83] Thomas Johnsson. The G-machine: An abstract machine for graph reduction. In *Declarative Programming Workshop*, pages 1–19, University College London, April 1983.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Functional Programming Languages and Computer Architecture [Fun85], pages 190–203.

[Kel89] R. A. Kelsey. *Compilation by Program Transformation*. PhD thesis, Yale University, Department of Computer Science, May 1989. YALEU/DCS/RR-702.

[KKR+86] David. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: An optimizing compiler for scheme. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 219–233, June 1986. SIGPLAN Notices 21(7).

[Kra88] D. A. Kranz. *ORBIT - an optimising compiler for Scheme*. PhD thesis, Yale University, Department of Computer Science, May 1988.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *ACM SIGPLAN Principles of Programming Languages*, Charleston, 1993.

[LGH+92] John Launchbury, Andrew Gill, John M. Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In Launchbury and Sansom [LS92].

[LS92] John Launchbury and P. M. Sansom, editors. *Functional Programming, Glasgow 1992*, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.

[Mar93] Simon Marlow. Update avoidance analysis by abstract interpretation. In Functional Programming, Glasgow 1993 [Fun93a].

[Mat93] Brian Matthews. MERILL: An equational reasoning system in Standard ML. In *5th International Conference on Rewriting Techniques and Applications*, number 690 in LNCS, pages 414–445. Springer-Verlag, 1993.

[Mat94] Brian Matthews. Analysing a set of transformation rules using completion. 1994.

[MOTW95] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda-calculus. In *Mathematical Foundations of Programming Semantics*, 1995. Electronic Notes in Theoretical Computer Science 1.

[MTW95] C. Mossin, David N. Turner, and Philip Wadler. Once upon a type. In *Functional Programming Languages and Computer Architecture*, San Diego, June 1995.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Dept. of Computer Science, University of Edinburgh, 1981.

[Par92] William Partain. The nofib benchmarking suite. In Launchbury and Sansom [LS92].

[PCSH87] S. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – a high-performance architecture for parallel graph reduction. In *IFIP conference on Functional Programming Languages and Computer Architecture*. Springer Verlag, September 1987.

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Pey92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PHH+93] Simon Peyton Jones, C. Hall, Kevin Hammond, William Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, March 1993.

[PJ93] John Peterson and Mark Jones. Implementing type classes. In *Proceedings of ACM SIGPLAN Symposium on Programming Language Design and Implementation.* ACM SIGPLAN, June 1993.

[PL91a] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Functional Programming Languages and Computer Architecture*, pages 636–666, September 1991.

[PL91b] Simon Peyton Jones and David Lester. A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, May 1991.

[PP93] Simon Peyton Jones and William Partain. On the effectiveness of a simple strictness analyser. In Functional Programming, Glasgow 1993 [Fun93a].

[PS93] Andrew M. Pitts and Ian D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science*, pages 122–141, Berlin, 1993. LNCS 711.

[PS94] Simon Peyton Jones and André Santos. Compilation by transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Ayr, Scotland, 1994. Springer Verlag, Workshops in Computing.

[RG89] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Inform. Proc. Lett.*, 32:137–142, 1989.

[RP94] Eike Ritter and Andrew M. Pitts. A fully abstract translation between a λ-calculus with reference types and standard ml. 1994.

[San93] D. Sands. A naïve time analysis and its theory of cost equivalence. Technical Report DIKU D-173, University of Copenhagen, 1993.

[Sch86] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Sew94] J. R. Seward. *Abstract Interpretation of Functional Languages: A Quantitative Assessment*. PhD thesis, University of Manchester, September 1994.

[SP92] André Santos and Simon Peyton Jones. On program transformation in the Glasgow Haskell Compiler. In Launchbury and Sansom [LS92].

[SP93] Patrick Sansom and Simon Peyton Jones. Generational garbage collection for Haskell. In Functional Programming Languages and Computer Architecture [Fun93b], pages 106 – 116.

[Sun93] Sun Microsystems. *Introduction to SpixTools*. 1993.

[Tak88] Masato Takeichi. Lambda-hoisting: A transformation technique for fully lazy evaluation of functional programs. *New Generation Computing*, (5):377–391, 1988.

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[Tur81] David A. Turner. The future of applicative programming. In *ECI-81*, Munich, October 1981.

[Wad71] C. Wadsworth. *Semantics and Pragmatics for the Lambda Calculus*. PhD thesis, Department of Mathematics, Oxford University, 1971.

[Wad87] Philip Wadler. Efficient compilation of pattern-matching. In [Pey87], 1987.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM TOPLAS*, 4(3):496–517, July 1982.

[WB89] Philip Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Symposium on Principles of Programming Language*, Austin, 1989.