Maximilian Bolingbroke
Robinson College
mb566

CST Part II Project 2007/2008

Adding SQL-Style List Comprehensions To The Glasgow Haskell Compiler

Proforma

Name: Maximilian Bolingbroke

College: Robinson College

Title: Adding SQL-Style List Comprehensions To The Glasgow Haskell Compiler

Examination: CST Part II

Year: 2007/2008

Word Count: 11,981

Project Originator: Simon Peyton Jones

Project Supervisor: Simon Peyton Jones

Project Aims:

1. Implementation in the Glasgow Haskell Compiler of a form of the ordering syntax from the paper.
2. Evaluation of the effect of a naive implementation of this syntax on the performance of existing programs.
3. Ensuring that the new syntax has the desired semantics by a process of rigorous testing or another appropriate method.
4. Evaluation of the characteristics of different desugarings for the ordering construct within the compiler given the experience of implementing a simple variant.
5. Implementation of any improvements to the desugaring that can be determined and an exploration of the time, space and compile time characteristics of this resulting system.

Project Work Completed:

1. Exploration of technically necessary syntactic refinements to the original syntax.
2. Implementation in the Glasgow Haskell Compiler of a refined version of the papers ordering syntax.
3. Additionally implemented and evaluated a refined version of the grouping syntax from the paper.
4. Quantitative evaluation of the resource consumption of alternative desugaring strategies for the extension.
5. Production of automated tests for ensuring semantic correctness of the extension.
6. Evaluation of impact of final extension on compile times, performance and its effects on existing programs.
7. Implementation, evaluation and integration of completed extension, Static Argument Transformation and fusable list literals extensions into GHC HEAD branch.

Special Difficulties: None

Declaration of Originality

I, Maximilian Bolingbroke of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date May 4, 2008

Contents

Chapter 1

Introduction

1.1 Overview

The Glasgow Haskell Compiler[13] is the leading implementation of the lazy functional programming language Haskell[12]. The designers of Haskell provided for a simple so-called list comprehension syntax which can be used in place of the traditional list manipulation functions `map` and `filter` but which is more user friendly. I demonstrate this in the next two Haskell expressions to compute twice all the odd numbers between 1 and 4: these program fragments compute the same result, but one has been expressed with these functions and the other with a comprehension.

```
k1 = [x * 2 |x <- [1, 2, 3, 4], x ‘mod‘ 2 != 0]
k2 = map (\x -> x * 2) (filter (\x -> x ‘mod‘ 2 != 0) [1, 2, 3, 4])
```

As you can see, the list comprehension is somewhat easier to read than the equivalent function-based expression. Furthermore, list comprehensions are implemented by the compiler and so it is free to make efficient choices about that implementation that the programmer may not have made for reasons of laziness or desiring to keep the code readable. However, they are somewhat limited: we have essentially designed a system which is capable of *selecting* and *projecting* from lists, but nothing else.

Recently a paper was published by Simon Peyton Jones and Phil Wadler[14] which provides a theoretical basis for two extensions to comprehensions in Haskell to give them some of the same expressiveness as SQL by adding *ordering* and *grouping* operators. Indeed, the semantics and type system for these syntactic forms there presented show that they are rather more general than their SQL equivalents! By implementing these in the leading Haskell compiler, GHC, I could potentially provide an avenue for replacing much existing ad-hoc code operating over lists in this manner, which would have the benefits previously outlined of clarity to human readers and expressiveness to the compiler. An example of their use, drawn from the paper, is this code:

```
[ (the dept, sum salary)
|(name, dept, salary) <- employees
, group by dept
, order by Down (sum salary)
, order using take 5 ]
```

This returns a list of department and salary list pairs where each department occurs only once and the salary list is that of the top 5 earners in the corresponding department. Note the use of *order using* instead of what would be a LIMIT clause in traditional SQL: this hints at the generalization of ordering that was made in the paper which we will explore later.

In this dissertation I show how I worked through the issues associated with this language extension and successfully integrated the resulting implementation into the Glasgow Haskell Compiler, so that it will be available to users with the next public release of the compiler. My conclusion in section 5 shows that I have accomplished and indeed exceeded the goals of my project proposal by implementing not only the proposed syntax but all of the capabilities of the syntax of the originating paper and two extensions to the project in the form of compiler optimizations.

This dissertation, by its nature, assumes some familiarity with the Haskell programming language. I have tried to write clear code that avoids using advanced Haskell features and I thoroughly explain the language feature I am actually developing in this dissertation, but the reader unfamiliar with Haskell would do well to read the enclosed tutorial in appendix C or to consult one of the many online resources for learning the language.

1.2 The Proposed Extension

We will begin by examining the proposed extensions to list comprehensions through examples of their use. I will make use of the following common code that models a database of equities:

```
data Exchange = LSE
              |NYSE
              |NASDAQ
              deriving (Eq, Show, Ord)

type Equity = (String, String, Exchange, Double)

equities ::  [Equity]
equities =
    [
        ("Vodafone",         "VOD.L",  LSE,    187.80),
        ("BT Group",         "BT-A.L", LSE,    266.00),
        ("Microsoft",        "MSFT",   NASDAQ, 34.38),
        ("Apple",            "AAPL",   NASDAQ, 180.05),
        ("Blackstone Group", "BX",     NYSE,   20.30),
        ("Goldman Sachs",    "GS",     NYSE,   199.93)
    ]
```

1.2.1 Order By

We can use the "order by" syntax extension to sort the list being generated by the comprehension by some field which has a corresponding `Ord` instance:

```
[name |(name, ticker, exchange, price) <- equities, order by price]
```

This first draws the data from the list of stocks, orders by the price (the last element of each tuple) and finally outputs the corresponding names in that order. The resulting list is:

```
["Blackstone Group", "Microsoft", "Apple", "Vodafone", "Goldman Sachs", "BT Group"]
```

It is important to note that with ordering (and indeed all subsequent new list comprehension features) there is no constraint on where in the comprehension it appears. In particular, it is not necessary for ordering to be the last action in the comprehension, as it is in SQL.

1.2.2 Order Using

In the rather more interesting "order using" variant of the ordering syntax the user is allowed to supply their own sorting function. This function is constrained to having the type $\forall\alpha.[\alpha] \rightarrow [\alpha]$, which has the consequence that it may not examine the list being ordered at all, just permute or select some subset from it. Indeed, the lack of a qualifying *Ord* instance means that it is in fact impossible to actually order the list, quite contrary to what the syntax suggests! Nonetheless this feature can come in useful, as the following example shows:

```
getEquityPageData ::  Integer -> [String]
getEquityPageData pageNumber = [name |(name, ticker, exchange, price) <- equities
                                     , order using (take pageLength) .  (drop pageStart)]
    where pageLength = 2
          pageStart = (pageNumber - 1) * pageLength
```

If we then request page number two we obtain this result:

```
["Microsoft", "Apple"]
```

1.2.3 Order By Using

The "order by using" syntax variant in some sense brings together the last two by allowing both a user defined sorting function and field selector to be used. What this means is that the function supplied to the *using* clause must have the type $\forall\alpha.(\alpha \rightarrow \tau) \rightarrow [\alpha] \rightarrow [\alpha]$, where τ is just the type of the expression supplied to the *by* clause. The first parameter to this function is instantiated with a function which allows it to project out a value of some known type from any element of the list being sorted. Because the element being projected out does not necessarily need be orderable for sensible "ordering" functions we relax the constraint of section 1.2.1 that τ must have an `Ord` instance. An example is;

```
[ticker |(name, ticker, exchange, price) <- equities
        , order by price
        , order by price < 100.0 using takeWhile]
```

Notice that the example has also made use of the form of ordering without a *using* clause of the form introduced in 1.2.1. It evaluates to the following result:

```
["BX", "MSFT"]
```

It is possible to consider the usage of section 1.2.1 a special case of this one where the *using* clause is implicitly supplied for you and points to a function with this definition:

```
sortWith ::  Ord b => (a -> b) -> [a] -> [a]
sortWith f = sortBy (\x y -> compare (f x) (f y))
```

Furthermore, the usage of section 1.2.2 can be seen as a special case where the function f supplied to its *using* clause is expanded to \\x -> f (for some fresh identifier x) and the *by* clause is implicitly set to `()`, so we would end up with this code for the example of that section:

```
getEquityPageData ::  Integer -> [String]
getEquityPageData pageNumber = [name |(name, ticker, exchange, price) <- equities
                                     , order by () using (\junk -> (take pageLength) .  (drop pageStart))]
    where pageLength = 2
          pageStart = (pageNumber - 1) * pageLength
```

1.2.4 Group By

The "group by" extension, as you might expect, allows the user to group a list into a number of sub-lists based on the equality of some expression. Actually, the expression is required to have not just an *Eq* instance (as you might expect) but also an *Ord* instance. This is because the results of processing the groups are presented according to the order imposed by the field that is being grouped by. An example of the grouping extension might be:

```
[(the exchange, name, average price) |(name, ticker, exchange, price) <- equities, group by exchange]
```

To execute this we also need two pieces of library code:

```
average ::  Fractional a => [a] -> a
average xs = foldl (+) 0 xs / (genericLength xs)


the ::  Eq a => [a] -> a
the (x:xs)
  |all (x ==) xs = x
  |otherwise     = error "the:  non-identical elements"
the []           = error "the:  empty list"
```

The example will then yield its result:

```
[(LSE, ["Vodafone", "BT Group"], 226.9), (NYSE, ["Blackstone Group", "Goldman Sachs"], 110.115),
(NASDAQ, ["Microsoft", "Apple"], 107.215)]
```

It is important to notice that although before the *group by* statement the *price*, *exchange* and *name* identifiers referred to single values drawn from the list of stocks, after it they refer instead to **lists of those single values**. The output clearly shows this: the second tuple element is printed as a literal list, and the prices that are shown are in fact the average of the prices in the initial list on a per-exchange basis.

One possible point of confusion is that we have had to apply a function *the* to the field we grouped on: all this does is check that all the elements in its list are the same and then returns that element. This is actually necessary due to the generality of the grouping operation that we introduce next. While we could hide this fact in this particular case by implicitly calling the *the* function in order to do this reduction implicitly for the user, this is confusing behaviour when viewed in the context of the other possible grouping operations we introduce next, where calling *the* is not necessarily the right thing to do.

1.2.5 Group Using

Just as in the case of ordering, the "group using" form of the grouping statement allows the user to supply a function to perform the grouping. Such a function is required to have the type $\forall\alpha.[\alpha] \rightarrow [[\alpha]]$. Again, no particular demand is made on the type classes α must have instances for and so it is impossible for the function to check the equality or relative ordering of two α values, so actual grouping cannot be performed in this form! However, we can nonetheless do useful operations, as this reimplementation of the pagination example from before shows:

```
runs ::  Integer -> [a] -> [[a]]
runs n = unfoldr (trySplitAt n)

trySplitAt ::  Integer -> [a] -> Maybe ([a], [a])
trySplitAt n xs = if left == [] then Nothing else Just (left, right)
  where (left, right) = genericSplitAt n xs
```

```
getEquityPageData ::  Integer -> [String]
getEquityPageData pageNumber = [ticker |(name, ticker, exchange, price) <- equities
                                         , group using runs 2] !!  (pageNumber - 1)
```

If we retrieve the second page, as before, we obtain this result:

```
["Microsoft", "Apple"]
```

1.2.6 Group By Using

This is the final component of the extension. Analogously to its equivalent for ordering that we presented in section 1.2.3, "group by using" unifies the two previous two syntax extensions for grouping. It takes a user defined grouping function and an expression to turn into projection function for that grouping function. The type of the grouping function must be $\forall\alpha.(\alpha \rightarrow \tau) \rightarrow [\alpha] \rightarrow [[\alpha]]$: the first argument is a function that allows the grouping function to evaluate the expression in the *by* clause (of type τ) for any given value in the list. Analogously with the ordering scenario, the requirement that τ necessarily have an `Ord` or `Eq` instance associated with it that we imposed in section 1.2.4 is relaxed. An example of the use of this is shown below, to determine the average price of equities listed on the NYSE versus the average price on all other exchanges:

```
partitionToList ::  (a -> Bool) -> [a] -> [[a]]
partitionToList f = (\(x, y) -> [x, y]) .  (partition f)

result = [(nub exchange, average price) |(name, ticker, exchange, price) <- equities
                                          , group by exchange == NYSE using partitionToList]
```

When evaluated, the `result` identifier refers to this value:

```
[([NYSE], 110.115), ([LSE, NASDAQ], 167.0575)]
```

As before, it is possible to consider the usage of section 1.2.4 as a a special case of this one where the *using* clause is implicitly supplied for you and points to a function with this definition:

```
groupWith ::  Ord b => (a -> b) -> [a] -> [[a]]
groupWith f = groupBy (\x y -> f x == f y) .  sortWith f
```

Likewise, the usage of section 1.2.5 can be seen as a special case where the function f supplied to its *using* clause is expanded to \ x -> f (for some fresh identifier x) and the *by* clause is implicitly set to `()`. So, the example in that section could also be expressed like so:

```
getEquityPageData ::  Integer -> [String]
getEquityPageData pageNumber = [ticker |(name, ticker, exchange, price) <- equities
                                         , group by () using (\junk -> runs 2)] !!  (pageNumber - 1)
```

1.2.7 Formalization

The language features demonstrated above were formalised by the motivating paper as an extension to Haskell's syntax, type system and semantics. This is reproduced in figures 1.1, 1.2and 1.3 respectively. Here, as throughout the dissertation, the semantics is presented as a translation into more primitive syntactic constructs. I will use these formalizations as the basis for the further work in this dissertation.

Variables x, y, z

Expressions $e, f, g ::= \ldots \mid$ `[` e `|` q `]`

Patterns $w ::= x \mid (w_1, \ldots, w_n)$

Qualifiers

p, q, r	$::=$	w `<-` e	Generator
	$\mid$	`let` w `=` e	Let
	$\mid$	e	Guard
	$\mid$	`()`	Empty qualifier
	$\mid$	p, q	Cartesian product
	$\mid$	p `|` q	Zip
	$\mid$	$q,$ `order` [`by` e] [`using` f]	Order (must include at least one clause)
	$\mid$	$q,$ `group` [`by` e] [`using` f]	Group (must include at least one clause)
	$\mid$	`(` q `)`	Parentheses

Figure 1.1: Syntax of extended list comprehensions proposed by [14]

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \texttt{[}\ e\ \texttt{|}\ q\ \texttt{]} : \texttt{[}\tau\texttt{]}}\ \textsc{comprehension}$$

$$\boxed{\vdash w : \tau \Rightarrow \Delta}$$

$$\frac{}{\vdash x : \tau \Rightarrow \{x : \tau\}}\ \textsc{variable}$$

$$\frac{\vdash w_1 : \tau_1 \Rightarrow \Delta_1 \quad \dots \quad \vdash w_n : \tau_n \Rightarrow \Delta_n}{\vdash (w_1, \dots, w_n) : (\tau_1, \dots, \tau_n) \Rightarrow \Delta_1 \cup \dots \cup \Delta_n}\ \textsc{tuple}$$

$$\boxed{\Gamma \vdash q \Rightarrow \Delta}$$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash e \Rightarrow ()}\ \textsc{guard} \qquad \frac{}{\Gamma \vdash () \Rightarrow ()}\ \textsc{empty}$$

$$\frac{\Gamma \vdash e : \texttt{[}\tau\texttt{]} \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash w\ \texttt{<-}\ e \Rightarrow \Delta}\ \textsc{generator}$$

$$\frac{\Gamma \vdash e : \tau \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash \texttt{let}\ w\ \texttt{=}\ e \Rightarrow \Delta}\ \textsc{let}$$

$$\frac{\begin{array}{c}\Gamma \ \vdash \ p \ \Rightarrow \ \Delta \\ \Gamma, \Delta \vdash q \Rightarrow \Delta'\end{array}}{\Gamma \vdash p, q \Rightarrow \Delta, \Delta'}\ \textsc{product}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \Rightarrow \Delta \\ \Gamma \vdash q \Rightarrow \Delta'\end{array}}{\Gamma \vdash p\ \texttt{|}\ q \Rightarrow \Delta, \Delta'}\ \textsc{zip}$$

$$\frac{\begin{array}{c}\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall a.\ (a \to \tau) \to \texttt{[}a\texttt{]} \to \texttt{[}a\texttt{]}\end{array}}{\Gamma \vdash q, \texttt{order by}\ e\ \texttt{using}\ f \Rightarrow \Delta}\ \textsc{order}$$

$$\frac{\begin{array}{c}\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall a.\ (a \to \tau) \to \texttt{[}a\texttt{]} \to \texttt{[[}a\texttt{]]}\end{array}}{\Gamma \vdash q, \texttt{group by}\ e\ \texttt{using}\ f \Rightarrow \texttt{[}\Delta\texttt{]}}\ \textsc{group}$$

Figure 1.2: Typing of extended list comprehensions proposed by [14]

$$
\begin{aligned}
\mathtt{[}\ e\ \mathtt{|}\ q\ \mathtt{]} &= \mathtt{map}\ (\lambda q_v.\ e)\ \llbracket q \rrbracket \\[1em]
\llbracket w\ \mathtt{<\text{-}}\ e \rrbracket &= e \\
\llbracket \mathtt{let}\ w\mathtt{=}d \rrbracket &= \mathtt{[}d\mathtt{]} \\
\llbracket g \rrbracket &= \mathtt{if}\ g\ \mathtt{then\ [()]\ else\ []} \\
\llbracket () \rrbracket &= [()] \\
\llbracket p, q \rrbracket &= \mathtt{concatMap} \\
&\qquad (\lambda p_v.\ \mathtt{map}\ (\lambda q_v.\ (p_v, q_v))\ \llbracket q \rrbracket) \\
&\qquad \llbracket p \rrbracket \\
\llbracket p\ \mathtt{|}\ q \rrbracket &= \mathtt{zip}\ \llbracket p \rrbracket\ \llbracket q \rrbracket \\
\llbracket q, \mathtt{order\ by}\ e\ \mathtt{using}\ f \rrbracket &= f\ (\lambda q_v.\ e)\ \llbracket q \rrbracket \\
\llbracket q, \mathtt{group\ by}\ e\ \mathtt{using}\ f \rrbracket &= \mathtt{map}\ \mathtt{unzip}_{q_v}\ (f\ (\lambda q_v.\ e)\ \llbracket q \rrbracket) \\[1em]
(w\ \mathtt{<\text{-}}\ e)_v &= w \\
(\mathtt{let}\ w\ \mathtt{=}\ d)_v &= w \\
(g)_v &= () \\
()_v &= () \\
(p, q)_v &= (p_v, q_v) \\
(p\ \mathtt{|}\ q)_v &= (p_v, q_v) \\
(q, \mathtt{order\ by}\ e\ \mathtt{using}\ f)_v &= q_v \\
(q, \mathtt{group\ by}\ e\ \mathtt{using}\ f)_v &= q_v
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{unzip}_{()}\ e &= () \\
\mathtt{unzip}_{x}\ e &= e \\
\mathtt{unzip}_{(w_1,\ldots,w_n)}\ e &= (\mathtt{unzip}_{w_1}\ (\mathtt{map}\ \mathtt{sel}_{1,n}\ e), \\
&\quad \ldots, \\
&\quad \mathtt{unzip}_{w_n}\ (\mathtt{map}\ \mathtt{sel}_{n,n}\ e)) \\[1em]
\mathtt{sel}_{i,n} &= \lambda(x_1, \ldots, x_n).x_i
\end{aligned}
$$

Figure 1.3: Translation of extended list comprehensions proposed by [14]

Chapter 2

Preparation

2.1 Overview

In this section I describe the results of my preparatory work investigating GHC and list comprehensions. In particular, I define some GHC jargon and begin to discuss the schemes currently used to compile list comprehensions which are the basis on which I build in section 3.4. I also describe the software development methodology that was planned for and used.

2.2 Familiarisation With GHC

Before work on implementing the compiler extension could begin, I needed to acquire an understanding of the compiler internals. GHC follows the standard pipelined architecture for compilers, whereby an input source file is gradually transformed into a final output format by successive discrete stages that each perform a manageable amount of work. The complete process is illustrated in figure 2.1. I will illustrate the various stages by compiling by hand the following (entirely contrived) example code:

```
frobnicate xs = [(x, y)
                |x <- xs
                , y <- xs
                , let x = 2 * x
                , y < x]
```

2.2.1 Renaming

This stage of the compiler pipeline resolves identifiers in the source code according to the lexical scope rules of Haskell and then replaces textual identifiers that the user entered (e.g. "foo") with identifiers decorated with a unique number that serve to disambiguate two occurrences of the same textual identifier in a program for later pipeline stages. The example program will be transformed into one decorated with "uniquifying" numbers like so:

```
frobnicate xs_1 = [(x_4, y_3)
                  |x_2 <- xs_1
                  , y_3 <- xs_1
                  , let x_4 = 2 * x_2
                  , y_3 < x_4]
```

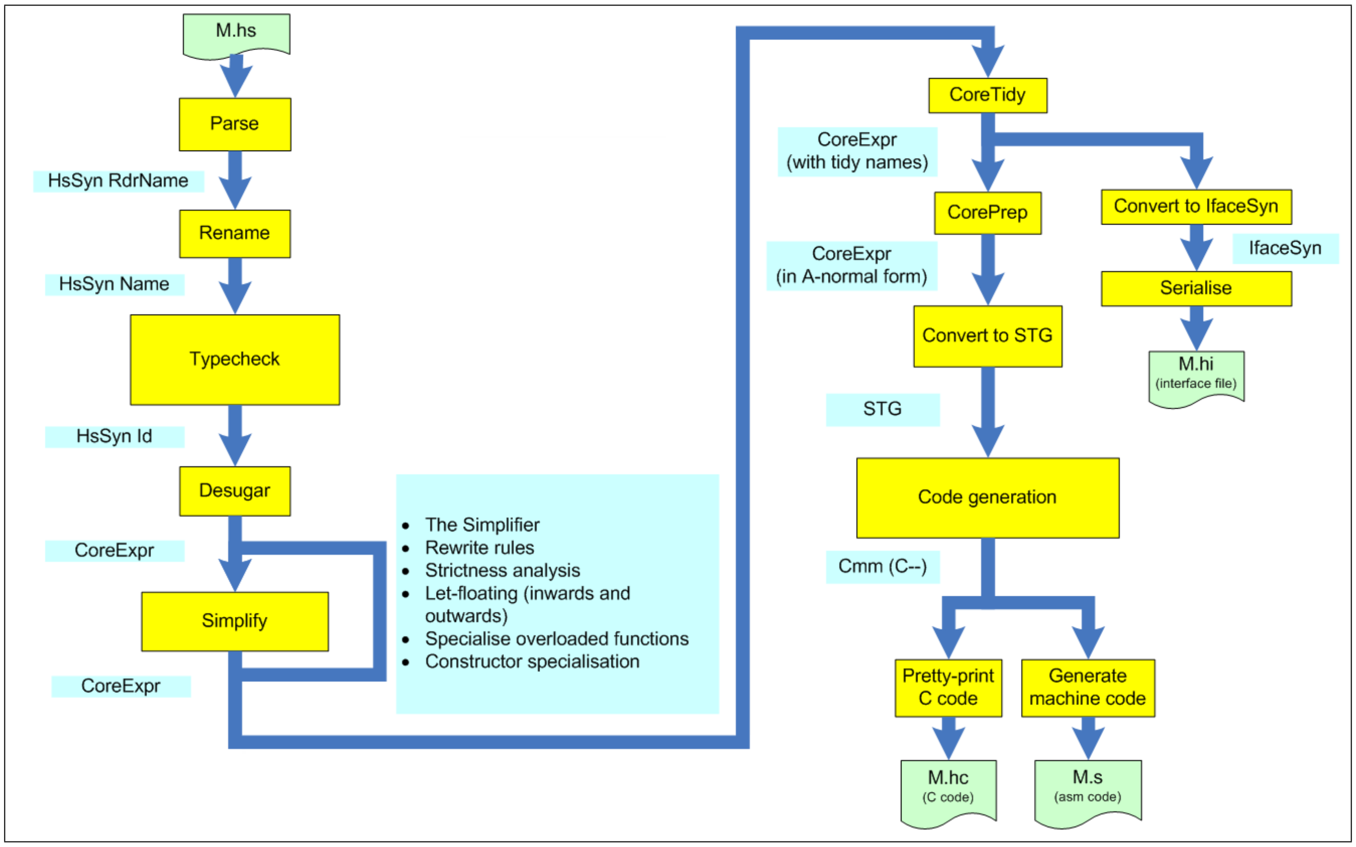

Figure 2.1: GHC compilation pipeline based on the diagram from the commentary [6]

2.2.2 Type Checking

In this stage, the compiler verifies that the provided program is type safe, performing type inference if necessary. The output of the stage is a program where identifiers have been decorated with type information and any necessary System F [10] Λ-abstractions and applications have been introduced (these abstractions are not user-visible and are elided throughout this dissertation). Our example program will type check and produce the following annotated program:

```
frobnicate ::  [Int] -> [(Int, Int)]   -- Inferred type signature
frobnicate (xs_1 ::  [Int]) = [(x_4, y_3) |x_2 ::  Int <- xs_1
                                          , y_3 ::  Int <- xs_1
                                          , let x_4 ::  Int = 2 * x_2
                                          , y_3 < x_4]
```

2.2.3 Desugaring

Desugaring is the process of transforming from the *HsExpr* data type, which encodes the full Haskell AST and has many convoluted alternatives, into the simple Core language. Core refers to the abstract syntax structure which is used by GHC as a minimal representation of the Haskell language. Core is actually an implementation of System FC [16] which is essentially System Fω [10] extended with limited type equality for the purposes of implementing generalized algebraic data types (GADTs).

The corresponding data type, *CoreExpr*, includes only nine different data constructors, of which two are used for simple housekeeping. This means that the subsequent stages only need concern themselves with a minimal number of possibilities for an expression, which vastly simplifies the implementation of things like optimization and code generation. It additionally means that new Haskell constructs immediately benefit from these operations simply by the addition of code to transform them into Core, which is good news for the implementation of my extension!

You might wonder why GHC does not do this before the type checking stage, in order to make our job at type checking much easier. The reasons we operate in this order are that:

- It is unclear if typeability is preserved by the desugaring operation.
- Some type information is needed to do the desugaring operation in one step (specifically, so GHC can decide what type class dictionaries to supply).
- Type-related error messages are clearer as they relate directly to the original user-visible source code.

List comprehensions (such as the one in my running example) are actually desugared in one of two different ways based on some rather intricate criteria. This will be explained further in section 3.4, but for now it will suffice to say that one possible desugaring is as follows:

```
frobnicate ::  [Int] -> [(Int, Int)]
frobnicate (xs_1 ::  [Int]) =
    let go_1 [] = []
        go_1 (x_2:xs_2) =
            let go_2 [] = (go_1 xs_2)
                go_2 (y_3:ys_3) =
                    let x_4 = 2 * x_2
                    in if y_3 < x_4
                       then (x_4, y_3) :  go_2 ys_3
                       else go_2 ys_3
            in go_2 xs_1
    in go_1 xs_1
```

As you can see, the Core language has no notion of list comprehensions, and the program has been reduced to a few primitive language features. However, here, as throughout the dissertation, I have taken certain liberties with the real Core representation that attempt to trade some faithfulness for much extra clarity:

- As mentioned in section 2.2.2, the types and type-abstractions of System FC are omitted. This requires the reader to infer the appropriate types for themselves, but has the advantage that it considerably unclutters the code.
- I have used pattern matching syntax: in fact, Core only supports an explicit `case` statement for these purposes.
- I will use the `[a, b, c]` syntax to represent list literals: these would actually be represented as a cons chain in Core.

2.3 Big Tuples

GHC has an inbuilt hard limit on the size of tuples it will support. The maximum arity m at the time of writing is 62 and although users are unlikely to want to create tuples of that size, it is quite possible to imagine a compiler stage running up against the limit in its automatically generated code. To head off this issue, the current list comprehension desugaring code makes use of a construct called “big tuples”, which are defined in figure 2.2 as the operation $\llparenthesis X \rrparenthesis$, where X is the list of binders or expressions involved in the big tuple. Essentially the idea is to turn tuples that would be larger than this limit into nested tuples-of-tuples instead, thus remaining inside the limit. The big tupling operation is defined recursively in such a way that tuples can be nested to any level, so that the number of levels required for tuples of size n is just $\lceil log_m n \rceil$. Due to this logarithmic behaviour, with only two levels of nesting it becomes possible to create big tuples of arities up to a size of $62^2 = 3844$.

In the following, the big tupling operation shall be understood to stand both for tuple constructing (allocation) and destruction (pattern matching) operations as appropriate for the context.

$$\llparenthesis X \rrparenthesis = \begin{cases} \llparenthesis (x_1, \ldots, x_m), (x_{m+1}, \ldots, x_{2m}), \ldots, (x_{km+1}, \ldots, x_{|X|}) \rrparenthesis & |X| > m \wedge k = \left\lfloor \frac{|X|-1}{m} \right\rfloor \\ (x_1, \ldots, x_{|X|}) & |X| \leq m \end{cases}$$

Figure 2.2: Big tuple desugaring for a maximum tuple size of m

2.4 Notes On Stream Fusion

Fusion is a technique to improve the space and time characteristics of some functional programs that make use of lists, which I will discuss further in section 3.4. Stream fusion is a system to achieve fusion that has been proposed [4] by Duncan Coutts et al. to replace the foldr/build fusion system that is currently ubiquitous throughout GHC. My original project proposal suggested I investigate the possibility of incorporating stream fusion into GHC as part of my project, however after my initial investigation I concluded that for various reasons it is not yet mature enough to be used for this purpose:

- Stream fusion has difficulty eliminating nested calls to `concatMap` that are handled by the existing foldr/build fusion.
- The reimplementation of the standard Haskell library to exploit stream fusion appears to have lingering functions where the implementation's strictness differs from that laid down by the Haskell specification.

What is more, correcting these faults is largely an orthogonal body of work to that of implementing the extended comprehensions. For these reasons I will not further consider stream fusion.

2.5 Parser Generator Conflicts

In order to make the necessary modifications to the parser I needed to remind myself of the two kinds of conflicts that can occur in a LR or LALR parser generator [28]:

- Shift-reduce conflict: this can occur when, given the current lookahead, it seems to the parser as though a valid parse would be assured if it either shifted the next symbol in the input onto its stack along with the appropriate next state, or if it reduced the top of the current stack to a single non-terminal symbol and then used the goto table to move into a new state. Typically a parser generator will warn about this condition but by default resolve the conflict by choosing to shift rather than reduce. This tends to create greedy parsers which accept as many symbols much they can for a non-terminal before reducing it, which is usually the desired behaviour.
- Reduce-reduce conflict: this is rather more serious than a shift-reduce conflict and occurs when for a given state there are at least two possible ways to reduce the symbols on the stack into a new non-terminal (which typically have different corresponding states to go to, as well). A parser generator will of course warn the user about this but typically also automatically resolve it by choosing one of the two reductions arbitrarily. This is highly unsatisfactory, as typically the grammar author had some particular meaning in mind which is likely to be preserved only by choosing a particular reduction.

2.6 Software Development Methodology

The development methodology I planned for and made use of on this project is the classic waterfall model described by Royce [22]. The high-level system and software requirements exist in the form of a paper on the extension that I intend to implement, per my proposal document. The analysis and program design phases are constrained sufficiently by this concrete guide and the existing phase-ordering structure of GHC that they are trivial. However, during the project I did see iterative interaction between the program design and coding phases, as described by Royce, as problems became apparent with the proposed specification: for details on the problem that occurred, see sections 3.1.2 and 3.1.3. After coding was complete, the testing and operations phases proceeded without further large glitches, although again testing fed back into a number of iterations of code design as bugs were discovered and fixed in the normal course of development.

Chapter 3

Implementation

This section will show how the extension was actually constructed. I will examine each section of the compiler in the order that the incoming source code encounters it, and discuss how the stage had to be extended to accommodate the new syntax. The result of this section will be a space and time efficient implementation of the proposed syntax extension in GHC.

In sections 3.5 and 3.6 I describe two standalone optimizations that I implemented in GHC as a result of my investigations into improving the desugaring of the syntax extension. These optimizations exceed the requirements of my project proposal but will provide concrete benefits to all Haskell programs compiled with future versions of GHC.

3.1 Parsing

3.1.1 Introduction

In order to lex and parse Haskell source code, GHC makes use of the lexical analyser generator Alex [15] and the parser generator Happy [9]. Happy generates standard LALR parsers which can execute associated context actions written in Haskell as the parse progresses. As is usual in compilers, GHC uses these context actions to build up an abstract syntax tree which represents the Haskell program being parsed.

I began my implementation work on the list comprehension extension by modifying the existing parser to accept the new list comprehension "qualifiers". In GHC jargon, qualifiers are those statements which may occur in list comprehensions, guards and the syntactic sugar for monadic and arrow based computations that GHC provides.

3.1.2 Resolving Parser Generator Limitations

Unfortunately a problem arose early in my implementation of the new syntax: there was a reduce-reduce conflict (see section 2.5) in the generated parser for the syntax given in the paper. Consider this partially parsed text, where • has been used to indicate up to where the parser has processed:

```
[foo | (i, e) • <- ies]
```

At this stage it is valid to either reduce `(i, e)` to a pair of qualifiers (i.e. treat `i` and `e` as guard expressions) or to reduce it to the tuple pattern match `(i, e)`, which is the intended behaviour in this case. There are a number of alternative ways I could solve this:

- Disallow the bracketing of qualifiers in the manner proposed altogether.
  - This has the advantage that it keeps the concrete syntax simple and should cover all common use cases.
  - It does come with the trade-off that it reduces the composability of the qualifier syntax somewhat.
- I could keep bracketing in this manner but resolve the ambiguity at a later stage.
  - This would involve quite a large architectural change to the compiler: specifically, the productions for qualifiers will have to be added to those for patterns so that the `(i, e)` in the example above occupies one syntactic class. Then, if a `<-` was encountered next by the parser it could use the pattern from the previous non-terminal directly, otherwise it could convert that pattern into a qualifier list to put into the new binding qualifier.
  - This unification might have negative implications on the readability of some parsing error messages and could cause a cascade of further reduce-reduce conflicts!
- Introduce new syntax to allow this idiom to be expressed unambiguously.
  - This is attractive because it is possible to argue that if the parser finds it hard to understand this syntax then any human reader would as well: something more readable should be sought.

In pursuit of option three, I solicited syntax suggestions from the Haskell community via a Wiki page [3] on the GHC Trac system, populated with some initial suggestions from myself and Simon Peyton Jones, my project supervisor. The eventual list of possibilities is reproduced below:

```
-- 1) A new keyword, combined with brackets that do not
--      conflict with those for list construction
[ foo |x <- e,
        nest { y <- ys,
               z <- zs },
        x > y + 3 ]

-- 2) Trying to suggest pulling things out of a sublist
--      without having to mention binders
[ foo |x <- e,
        <- [ ..  |y <- ys,
                  z <- zs ],
        x > y + 3 ]

-- 3) New kind of brackets (not necessarily those shown here)
[ foo |x <- e,
        (|y <- ys,
           z <- zs |),
        x < y + 3 ]

-- 4) Variation on 2), slightly more concise
[ foo |x <- e,
        <- [ y <- ys,
             z <- zs ],
        x > y + 3 ]

-- 5) Another variation on 2), moving the ".." into
--      the pattern rather than the comprehension body
[ foo |x <- e,
        ..  <- [ y <- ys,
                 z <- zs ],
        x > y + 3 ]
```

Given that I wanted my extension to be integrated into the main GHC codebase, and that the decision had no obvious "right" answer I sought the advice of my supervisor as to which option to choose. What is more, due to his experience nurturing Haskell he had the best grasp on what idiom felt "right" for the language, and on how much interference each option would have with other, more important features of the compiler. Based on these criteria he decided that as none of the proposed syntaxes was entirely satisfactory and the only other alternative was major parser surgery it was best to leave out the ability to modify the associativity of list comprehension qualifiers in this manner. However, we agreed that we should have the ability to quickly reintroduce it at a later date if there was user demand or a satisfactory syntax came to light. The remainder of my implementation took the requirement for qualifier parenthesization into account, even though the functionality cannot currently be accessed by any concrete syntax.

It is worth noting that all parenthesised list comprehensions can still be expressed in the language without those brackets, at the cost of a little verboseness. For example, consider this code, drawn from the original paper:

```
xs = [1,2]
ys = [3,4]
zs = [5,6]

p1 = [ (x,y,z)
     |( x <- xs
       |y <- ys )
     , z <- zs ]
```

When run, this should have the following output:

```
[(1,3,5), (1,3,6), (2,4,5), (2,4,6)]
```

Now, it is possible to apply an obvious transformation based on explicit tuple construction and destruction to remove the parenthesization but retain meaning:

```
p1 = [ (x,y,z)
     |(x, y) <- [ (x, y)
                 |x <- xs
                 |y <- ys ]
     , z <- zs ]
```

3.1.3 Remaining Problems With The Syntax

Even with the changes outlined in section 3.1.2, the syntax had some potential issues:

1. It required introducing all of `order`, `group`, `using` and `by` as keywords when the extended comprehensions are enabled in the compiler, which will in turn cause any code that was previously using at least one of these words as an identifier to almost certainly fail to compile. This problem is exacerbated by the fact that they are all tempting identifier names.

2. There is actually a library function *group* in the module *List* which is commonly imported into the global namespace of Haskell programs by the user.

3. The `order` syntax has a type of sufficient generality that it can actually be used to accomplish many things besides ordering, as some of the examples above show.

What is more, there are Haskell programs which would have compiled with the extension disabled that would compile to a program with a different meaning once the extension has been enabled! Imagine a program such as the following:

```
module Main where
import Bar

main = print [ x |x <- [1], order using (take 1) ]
```

In this example, imagine that the *Bar* module exported the identifiers *order* and *using*, where *using* has type τ and *order* has type $\alpha \to ([\beta] \to [\beta]) \to$ `Bool` for some α and β. This is clearly a valid program where the *order* application is intended to represent a guard qualifier. If we imagine that this guard always returns `False` then running the program yields the output `[]`. However, when the module is compiled with comprehensive comprehensions enabled the compiler will accept the program, but produce an executable whose output is `[1]`!

Although this is clearly a pathological case, it is undesirable for extensions to modify the meaning of existing programs in this manner.

3.1.4 Aside: Context Sensitive Keywords

A possible solution to the problem of keyword proliferation is to introduce these words as context sensitive keywords. This is an approach has been used successfully in the Microsoft sponsored Managed C++ Extensions [18] and C# [17]. The substance of the solution is that the parser would only recognise the aforementioned keywords in positions where they could actually appear without generating a parse error: for example, `order` would only be a keyword at the beginning of a list comprehension qualifier.

At the time of implementation I did not realize that context sensitive keywords were supported by the Alex and Happy suite of parser tools used by GHC. However, it came to light after implementation was complete that a technique exists for encoding them and indeed it is exploited by the parsing of Haskell “special-ids” such as `hiding`. This infrastructure could be reused to, for example, make the `by` keyword in the alternative syntax context sensitive. However the technique is necessarily limited because keywords that could potentially appear in the same positions as normal identifiers will not be able to be disambiguated. Hence the keyword `using` could not be context sensitive because `then group by e using f` could be parsed as `then group by (e using f)` or `then group by (e) using (f)`, which is an undesirable ambiguity. Likewise for the `group` keyword with the program text `then group by e`.

3.1.5 Resolving The Syntax Issues

After considering the remaining problems with the syntax discussed in section 3.1.3, alternative syntax suggestions were once again solicited on the GHC Wiki. The final suggestions were just those drawn from the original paper, and the format of the qualifiers in each case are summarized in the table below:

	Original	Alternative	Alternative With `using`, `by` Flipped
“Order By”	`order by e`	N/A	N/A
“Order Using”	`order using f`	`then f`	`then f`
“Order By Using”	`order by e using f`	`then f by e`	`then f using e`
“Group By”	`group by e`	`then group by e`	`then group using e`
“Group Using”	`group using f`	`then group using f`	`then group by f`
“Group By Using”	`group by e using f`	`then group by e using f`	`then group using e by f`

Variables x, y, z

Expressions $e, f, g ::= \ldots \mid$ `[` e `|` qs `]`

Patterns w

Qualifier Lists $qs ::= q, qs \mid \epsilon$

Qualifiers

$p, q, r ::=$	w `<-` e	Generator
$\mid$	`let` w `=` e	Let
$\mid$	e	Guard
$\mid$	$qs,$ `then` f	Transform
$\mid$	$qs,$ `then` f `by` e	Transform By
$\mid$	$qs,$ `then group by` e	Group By
$\mid$	$qs,$ `then group using` f	Group Using
$\mid$	$qs,$ `then group by` e `using` f	Group By Using

Figure 3.1: Final syntax of extended list comprehensions

Either of the alternative syntaxes attempt to address all three issues from the list above. To reduce the number of new keywords introduced while maintaining readability, addressing item 1, the existing keyword `then` is reused. This change has the two added advantages that firstly the syntax no longer suggests that a custom function you supply to it must necessarily perform ordering (which addresses item 3) and secondly it means that no existing Haskell program can have its meaning changed by compilation with the extension enabled. However it does come with the corresponding disadvantage that the "Order By" variant where you fail to specify a custom function becomes nonsensical since the compiler can in that situation no longer choose sorting as a reasonable default action.

The alternative syntaxes resolve item 2 only partially: since `group` can only occur after the `then` keyword, it is not possible to misread any use of it as a call to the *group* function. Unfortunately, it still leaves the actual *group* function inaccessible by default when transform comprehensions are enabled as its name unavoidably becomes a keyword! However, it is important to note that the lexer still parses code such as `List.group` as an identifier, so *group* can still be accessed by a qualified reference.

The two variants of the alternative syntax just alternate `using` and `by` keywords: this is a purely cosmetic choice which matters only for readability purposes.

After consulting with Simon Peyton Jones, we decided to move forward with the "unflipped" alternative syntax, as it had the advantages over the original proposal listed above and we felt that it was more conducive to readability than than the "flipped" variant. A possible avenue of future work would be to conduct some formal investigation into which option is actually considered more readable by a larger sample of users, but that issue is not the focus of this project.

The final syntax is summarised formally in figure 3.1.

3.2 Renaming

The process of renaming was described in section 2.2.1. In the section we will describe the process of extending the algorithm to work with the extended list comprehensions.

3.2.1 Scoping Rules

One subtlety with the implementation of renaming for extended comprehension occurs for programs that include a custom ordering or grouping function expression which mentions an identifier that was previously bound into the list comprehension, such as in this example:

```
f = take 1

[ x |x <- [1..10]
    , f <- [take 2, take 3]
    , then f ]
```

My initial implementation of renaming made the mistake of renaming the expressions representing custom grouping and transformation functions within the environment established by including the binders of the previous comprehension qualifiers. However, in the example this erroneous approach leads us trying to transform a list by every function it contains. Concretely, after renaming something like this is obtained:

```
f_1 = take 1

[ x_2 |x_2 <- [1..10]
      , f_2 <- [take 2, take 3]
      , then f_2 ]
```

Although it looks superficially plausible, this is not an operation which makes a lot of sense! Instead, contrary to what is suggested by the intuition of lexical scope, such custom function expressions must be renamed in the environment that strictly encloses the qualifiers being transformed. In our new restricted syntax this just means the environment outside the list comprehension, but in one with parenthesization it would instead mean the environment outside the current set of parenthesized qualifiers, if applicable. If we use this rule then the example renames like so:

```
f_1 = take 1

[ x_2 |x_2 <- [1..10]
      , f_2 <- [take 2, take 3]
      , then f_1 ]
```

For those left dead by this explanation, do not fear: the formal typing semantics presented in section 3.3 incorporates this information.

3.2.2 Name Rebinding For Grouping

Another subtle issue involved with the implementation is that any grouping operation actually has to rebind all the names that occur before it in the comprehension for use after the group. This is because **before** the group statement an identifier `x` refers to a single element of some type τ, and **after** it the lexically equivalent identifier `x` has the type $[\tau]$ and refers to many single elements. Thus, because each renamed identifier may only have one type associated with it a new binders corresponding to those identifiers x after the grouping has taken place must be introduced. This trick allows both types to coexist in the compiler. For example, consider the following program incorporating a grouping statement:

```
[ (z, x, y, l) |x <- [1..10]
               , y <- [1..10]
               , let z = x + y
```

```
              , group by x * y
              , let l = length x ]
```

When renamed it should look as follows:

```
[ (z_6, x_4, y_5, l_7) |x_1 <- [1..10]
                      , y_2 <- [1..10]
                      , let z_3 = x_1 + y_2
                      , group by x_1 * y_2
                      , let l_7 = length x_4 ]
```

Notice that there are now multiple renamed identifiers (e.g. `x_1`, `x_4`) corresponding to the intuitively similar identifiers in the original source file (e.g. `x`). In my implementation, the correspondence between these two identifiers is recorded by annotating the syntax tree with a list of pairs of ids: this list is not shown in any of the examples I will present, as it is clear from context.

3.3 Type Checking

A formal typing semantics for my formulation of the extension that also incorporates scoping information is given in figure 3.2.

The implementation of type checking for this extension is, for the most part, a simple transliteration of the formal semantics into code. There is however one interesting point, which is due to the problem described in section 3.2.2. When an identifier is rebound from before the grouping operation after it (which is done in the renaming stage) a brand new identifier is created which GHC considers totally independent from the one it was spawned from. However, it is not actually the case that they are independent: in fact, the type of the after-grouping binder must be constrained to just that of a list of the type of the initial binder.

This is done in the implementation by specially creating the type-carrying after-grouping binder as one which has an initial type equal to $[\tau]$, where τ is a type variable that holds the current most general type for the before-grouping binder. The type variables are removed when the type checking phase has ended by a so-called "zonking" stage, but in the interim the use of a variable ensures that any unification which happens to either of the binders automatically causes an update to the type of the other one appropriately.

3.4 Desugaring

As mentioned in section 2.2.3, there are two ways in which GHC may desugar list comprehensions: the TQ translation and the foldr/build translation. In the next two sections I will discuss the classical translations and their extensions to our new syntax. Throughout, the notation $[\![e]\!]$ will denote the Core language desugaring of the Haskell expression e.

The reader may be wondering GHC includes two methods for desugaring the comprehensions at all. This is due to the fact that the foldr/build translation always results in better code than the TQ translation except in two cases: if the user has made use of "zip" list comprehension qualifiers (which, incidentally, were handled in my implementation, though a description of them has been elided here for space reasons) or if the user has disabled GHC's rewrite rules system, in which case foldr/build is significantly worse. For this reason, GHC defaults to the foldr/build system but falls back on TQ should either of the two conditions above be met.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau}{\Gamma \vdash \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]} : \texttt{[}\tau\texttt{]}}\ \textsc{Comprehension}$$

$$\boxed{\vdash w : \tau \Rightarrow \Delta}$$

$$\frac{\text{For each typed binder } (x_i, \tau_i) \text{ in the pattern } w}{\vdash w \Rightarrow \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}\ \textsc{Patterns}$$

$$\boxed{\Gamma \vdash qs \Rightarrow \Delta}$$

$$\frac{\Gamma \vdash q \Rightarrow \Delta \quad \Gamma, \Delta \vdash qs \Rightarrow \Delta'}{\Gamma \vdash q\texttt{,}\ qs \Rightarrow \Delta, \Delta'}\ \textsc{QualifierListCons}$$

$$\frac{}{\Gamma \vdash \epsilon \Rightarrow \emptyset}\ \textsc{QualifierListNil}$$

$$\boxed{\Gamma \vdash q \Rightarrow \Delta}$$

$$\frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash e \Rightarrow \emptyset}\ \textsc{Guard}$$

$$\frac{\Gamma \vdash e : \texttt{[}\tau\texttt{]} \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash w\ \texttt{<-}\ e \Rightarrow \Delta}\ \textsc{Generator}$$

$$\frac{\Gamma \vdash e : \tau \quad \vdash w : \tau \Rightarrow \Delta}{\Gamma \vdash \texttt{let}\ w\ \texttt{=}\ e \Rightarrow \Delta}\ \textsc{Let}$$

$$\frac{\Gamma \vdash qs_1 \Rightarrow \Delta_1 \quad \ldots \quad \Gamma \vdash qs_n \Rightarrow \Delta_n}{\Gamma \vdash qs_1\ \texttt{|}\ \ldots\ \texttt{|}\ qs_n \Rightarrow \Delta_1, \ldots, \Delta_n}\ \textsc{Zip}$$

$$\frac{\Gamma \vdash qs \Rightarrow \Delta \quad \Gamma \vdash f : \forall\alpha.\ \texttt{[}\alpha\texttt{]} \to \texttt{[}\alpha\texttt{]}}{\Gamma \vdash qs, \texttt{then}\ f \Rightarrow \Delta}\ \textsc{Transform}$$

$$\frac{\begin{array}{c}\Gamma \vdash qs \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall\alpha.\ (\alpha \to \tau) \to \texttt{[}\alpha\texttt{]} \to \texttt{[}\alpha\texttt{]}\end{array}}{\Gamma \vdash qs, \texttt{then}\ f\ \texttt{by}\ e \Rightarrow \Delta}\ \textsc{TransformBy}$$

$$\frac{\Gamma \vdash qs \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : (\texttt{Ord}\ \tau)\ \texttt{=>}\ \tau}{\Gamma \vdash qs, \texttt{then group by}\ e \Rightarrow \texttt{[}\Delta\texttt{]}}\ \textsc{GroupBy}$$

$$\frac{\Gamma \vdash qs \Rightarrow \Delta \quad \Gamma \vdash f : \forall\alpha.\ \texttt{[}\alpha\texttt{]} \to \texttt{[[}\alpha\texttt{]]}}{\Gamma \vdash qs, \texttt{then group using}\ f \Rightarrow \texttt{[}\Delta\texttt{]}}\ \textsc{GroupUsing}$$

$$\frac{\begin{array}{c}\Gamma \vdash qs \Rightarrow \Delta \quad \Gamma, \Delta \vdash e : \tau \\ \Gamma \vdash f : \forall\alpha.\ (\alpha \to \tau) \to \texttt{[}\alpha\texttt{]} \to \texttt{[[}\alpha\texttt{]]}\end{array}}{\Gamma \vdash qs, \texttt{then group by}\ e\ \texttt{using}\ f \Rightarrow \texttt{[}\Delta\texttt{]}}\ \textsc{GroupByUsing}$$

$$\boxed{\texttt{[}\Delta\texttt{]}}$$

$$\texttt{[}\Delta\texttt{]} = \{(x : \texttt{[}\tau\texttt{]}) | (x : \tau) \in \Delta\}$$

Figure 3.2: Final typing of extended list comprehensions

3.4.1 TQ Translation

The TQ comprehension desugaring scheme is inspired by Philip Wadler's concatenate-removing transformation [31]. Perhaps the best way to get a flavour of what is does is to see an example:

```
[x * 2 |x <- [1, 2, 3, 4], x `mod` 2 /= 0]
```

Using the TQ translation this comprehension will desugar like so:

```
let h [] = []
    h (x:ws) =
      if x `mod` 2 /= 0
      then (x * 2) :  (h ws)
      else (h ws)
in h [1, 2, 3, 4]
```

I give a full semantics for the TQ translation on classical comprehensions in figure 3.3. Essentially the idea is for the transformation to carry around the already-desugared tail of the list comprehension in the meta-parameter L. By doing this it turns out every element-adding operation the comprehension needs to do can be implemented as a direct addition onto the head of this list, ensuring no concatenation needs to take place at any stage. This avoids the major inefficiency of a naïve implementation of list comprehensions which is the inclusion of a call to `concatMap` in the desugaring for the bind operation (i.e. `w <- e`), and hence TQ greatly improves the time spent constructing lists from a worst case of $O(n^2)$ to $O(n)$ in the length of the result list.

TQ Translation Activation

$$\llbracket [\ e \mid qs\ ] \rrbracket = \llbracket [\ e \mid qs\ ]\ \texttt{++}\ \texttt{[]} \rrbracket_{TQ}$$

TQ Translation

$$\begin{aligned}
\llbracket [\ e \mid \epsilon\ ]\ \texttt{++}\ L \rrbracket_{TQ} &= \llbracket e \rrbracket\ \texttt{:}\ L \\
\llbracket [\ e \mid w\ \texttt{<-}\ e_l\texttt{,}\ qs\ ]\ \texttt{++}\ L \rrbracket_{TQ} &= \texttt{let h [] = } L \\
&\qquad\quad \texttt{h (}\llbracket w \rrbracket\texttt{:ws) = } \llbracket [\ e \mid qs\ ]\ \texttt{++ (h ws)} \rrbracket_{TQ} \\
&\quad\ \texttt{in h } \llbracket e_l \rrbracket \\
\llbracket [\ e \mid \texttt{let}\ w\ \texttt{=}\ e_v\texttt{,}\ qs\ ]\ \texttt{++}\ L \rrbracket_{TQ} &= \texttt{let } \llbracket w \rrbracket \texttt{ = } \llbracket e_v \rrbracket \texttt{ in } \llbracket [\ e \mid qs\ ]\ \texttt{++}\ L \rrbracket_{TQ} \\
\llbracket [\ e \mid e_b\texttt{,}\ qs\ ]\ \texttt{++}\ L \rrbracket_{TQ} &= \texttt{if } \llbracket e_b \rrbracket \texttt{ then } \llbracket [\ e \mid qs\ ]\ \texttt{++}\ L \rrbracket_{TQ} \texttt{ else } L
\end{aligned}$$

Figure 3.3: TQ translation desugaring for classical list comprehensions

The challenge in this part of my implementation was to extend this classical treatment to incorporate the new syntax. The precise specification of the translation semantics are a key contribution of this dissertation, and they are presented in figure 3.4, which makes use of the obvious free/bound variable functions of figure 3.5. Given the following comprehension:

```
[ (the z, y) |z <- ["foo", "bar", "baz"]
             , y <- [1, 2, 3, 4]
             , then take 3
             , then group by z ]
```

My extended desugaring will translate this as follows (where `groupWith` is as defined in section 1.2.6):

```
let g (z, y) = z
in let h1 [] = []
       h1 ((z, y):ws) = (the z, y) :  (h1 ws)
```

```
  in h1 (map unzip_2 (groupWith g (
          let h2 [] = []
              h2 ((z, y):ws2) = (z, y) :  (h2 ws2)
          in h2 (take 3 (let h3 [] = []
                             h3 (z:ws3) = let h4 [] = h3 ws3
                                              h4 (y:ws4) = (z, y) :  (h4 ws4)
                                          in h4 [1, 2, 3, 4]
                         in h3 ["foo", "bar", "baz"])))))
```

A moderately interesting issue is deciding what exactly should constitute the element type of the inner list comprehension generated by this process. A naive implementation might simply create a list of tuples where each tuple served to "roll up" all those identifiers bound in the inner qualifiers. Concretely, this means that a program such as this:

```
[b |(a, b, c) <- foos
   , c > 2
   , then group by a]
```

Would be translated into one logically equivalent to this:

```
[b |(a, b, c) <- unzip3 $ groupWith (\(a, _, _) -> a)
                    [(a, b, c) |(a, b, c) <- foos
                               , c > 2 ]]
```

However, I have adopted the slightly more complex approach of determining which of the things bound in the inner qualifiers are actually referred to in the rest of the comprehension: only those that pass this test are bound into the tuple which is used to move information out of the inner comprehension. This serves to reduce memory usage by discarding intermediate results not used after the transformation or grouping operation before they take up space in the tuples. For our example, code something like the following will be generated:

```
[b |(a, b) <- unzip3 $ groupWith fst
                    [(a, b) |(a, b, c) <- foos
                            , c > 2 ]]
```

3.4.2 Foldr/Build Translation

The foldr/build translation [8] is the alternative comprehension desugaring scheme used by GHC. It is designed to correct an infelicity in the TQ translation in the handling of nested list comprehensions. Consider the following example:

```
[ y |x <- [1, 2, 3]
    , y <- [ x * (z * 2) |z <- [4, 5, 6] ]
    , y < 10 ]
```

Under the TQ scheme it will result in the following Core code:

```
let h1 [] = []
    h1 (x:hs1) =
        let h2 [] = h1 hs1
            h2 (y:hs2) =
                if y < 10
```

TQ Translation Extensions

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_1\ \texttt{|}\ \ldots\ \texttt{|}\ qs_n),\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\ = \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis\llparenthesis V_1\rrparenthesis,\ldots,\llparenthesis V_n\rrparenthesis\rrparenthesis\ \texttt{<-}\ zip_n\\
&\qquad\quad \llbracket \texttt{[}\ \llparenthesis V_1\rrparenthesis\ \texttt{|}\ qs_1\ \texttt{]}\rrbracket\ \ldots\ \llbracket \texttt{[}\ \llparenthesis V_n\rrparenthesis\ \texttt{|}\ qs_n\ \texttt{]}\rrbracket,\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V_i = BV_{qs}(qs_i) \cap FV_e(e)
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then}\ f),\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\ = \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V\rrparenthesis\ \texttt{<-}\ \llbracket f\rrbracket\ \llbracket \texttt{[}\ \llparenthesis V\rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]}\rrbracket,\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then}\ f\ \texttt{by}\ e_l),\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\ = \texttt{let g}\ \llparenthesis V\rrparenthesis\ \texttt{=}\ \llbracket e_l\rrbracket\\
&\qquad \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V\rrparenthesis\ \texttt{<-}\ \llbracket f\rrbracket\ \texttt{g}\ \llbracket \texttt{[}\ \llparenthesis V\rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]}\rrbracket,\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group by}\ e_l),\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\ = \texttt{let g}\ \llparenthesis V\rrparenthesis\ \texttt{=}\ \llbracket e_l\rrbracket\\
&\qquad \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V\rrparenthesis\ \texttt{<-}\ \texttt{map}\ unzip_{|V|}\\
&\qquad\qquad \texttt{(groupBy g}\ \llbracket \texttt{[}\ \llparenthesis V\rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]}\rrbracket\texttt{)},\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&db\texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group using}\ f),\ qs\ \texttt{]}\ \texttt{++}\ L_{TQ}\\
&\ = \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V\rrparenthesis\ \texttt{<-}\ \texttt{map}\ unzip_{|V|}\\
&\qquad\qquad \texttt{(}f\ \llbracket \texttt{[}\ \llparenthesis V\rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]}\rrbracket\texttt{)},\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group by}\ e_l\ \texttt{using}\ f),\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\ = \texttt{let g}\ \llparenthesis V\rrparenthesis\ \texttt{=}\ \llbracket e_l\rrbracket\\
&\qquad \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V\rrparenthesis\ \texttt{<-}\ \texttt{map}\ unzip_{|V|}\\
&\qquad\qquad \texttt{(}\ f\ \texttt{g}\ \llbracket \texttt{[}\ \llparenthesis V\rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]}\rrbracket\texttt{)},\ qs\ \texttt{]}\ \texttt{++}\ L\rrbracket_{TQ}\\
&\qquad \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

Unzip Function Generation

$$\begin{aligned}
unzip_n\ = &\ \texttt{let g (}\llparenthesis[x_1,\ldots,x_n]\rrparenthesis\texttt{:}\llparenthesis[xs_1,\ldots,xs_n]\rrparenthesis\texttt{)}\ \texttt{=}\ \llparenthesis[x_1\texttt{:}xs_1,\ldots,x_n\texttt{:}xs_n]\rrparenthesis\\
&\ \texttt{in foldr g}\ \llparenthesis[\texttt{[]}_1,\ldots,\texttt{[]}_n]\rrparenthesis
\end{aligned}$$

Figure 3.4: TQ translation desugaring for extended list comprehensions

```
                then y :  h2 hs2
                else h2 hs2
        in h2 (let h3 [] = []
                   h3 (z:hs3) = x * (z * 2) :  h3 hs3
               in h3 [4, 5, 6])
in h1 [1, 2, 3]
```

However, this is somewhat suboptimal in that it explicitly constructs an inner list of `x * (z * 2)` elements which is then immediately deconstructed to be incorporated into the outermost list. The foldr/build translation of list comprehensions would rather generate code like so:

```
build (\c n ->
    let f1 x b1 =
```

Free variables of qualifiers

$$\begin{aligned}
FV_{qs}(w \texttt{ <- } e, qs) &= FV_e(e) \cup (FV_{qs}(qs)/BV_w(w)) \\
FV_{qs}(\texttt{let } w = e, qs) &= FV_e(e) \cup (FV_{qs}(qs)/BV_w(w)) \\
FV_{qs}(e, qs) &= FV_e(e) \cup FV_{qs}(qs) \\
FV_{qs}((qs_i, \texttt{then } f), qs) &= FV_{qs}(qs_i) \cup FV_e(f) \cup (FV_{qs}(qs)/BV_{qs}(qs_i)) \\
FV_{qs}((qs_i, \texttt{then } f \texttt{ by } e), qs) &= FV_{qs}(qs_i) \cup FV_e(f) \cup ((FV_{qs}(qs) \cup FV_e(e))/BV_{qs}(qs_i)) \\
FV_{qs}((qs_i, \texttt{then group by } e), qs) &= FV_{qs}(qs_i) \cup ((FV_{qs}(qs) \cup FV_e(e))/BV_{qs}(qs_i)) \\
FV_{qs}((qs_i, \texttt{then group using } f), qs) &= FV_{qs}(qs_i) \cup FV_e(f) \cup (FV_{qs}(qs)/BV_{qs}(qs_i)) \\
FV_{qs}((qs_i, \texttt{then group by } e \texttt{ using } f), qs) &= FV_{qs}(qs_i) \cup FV_e(f) \cup ((FV_{qs}(qs) \cup FV_e(e))/BV_{qs}(qs_i))
\end{aligned}$$

Bound variables of qualifiers

$$\begin{aligned}
BV_{qs}(w \texttt{ <- } e, qs) &= BV_w(w) \cup BV_{qs}(qs) \\
BV_{qs}(\texttt{let } w = e, qs) &= BV_w(w) \cup BV_{qs}(qs) \\
BV_{qs}(e, qs) &= BV_{qs}(qs) \\
BV_{qs}((qs_i, \texttt{then } f), qs) &= BV_{qs}(qs_i) \cup BV_{qs}(qs) \\
BV_{qs}((qs_i, \texttt{then } f \texttt{ by } e), qs) &= BV_{qs}(qs_i) \cup BV_{qs}(qs) \\
BV_{qs}((qs_i, \texttt{then group by } e), qs) &= BV_{qs}(qs_i) \cup BV_{qs}(qs) \\
BV_{qs}((qs_i, \texttt{then group using } f), qs) &= BV_{qs}(qs_i) \cup BV_{qs}(qs) \\
BV_{qs}((qs_i, \texttt{then group by } e \texttt{ using } f), qs) &= BV_{qs}(qs_i) \cup BV_{qs}(qs)
\end{aligned}$$

Figure 3.5: Extended list comprehension free and bound variables

```
        let f2 y b2 =
            if y < 10
            then c y b2
            else b2
        in foldr f2 b1 (build (\c n ->
            let f3 z b3 = c (x * (z * 2)) b3
            in foldr f3 n [4, 5, 6]))
    in foldr f1 n [1, 2, 3])
```

Where *build* is defined as follows:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Now, it is not at first obvious that this strategy has actually resulted in better code: indeed, all I seem to have done is abstracted away from the cons (`(:)`) and nil (`[]`) list constructors and turned explicit recursion into uses of *foldr*. However, in later compiler stages GHC applies a term rewriting rule known as the foldr/build fusion rule that works some magic for us. The rule is as follows:

$$\texttt{foldr } c\ n\ (\texttt{build } g) = g\ c\ n$$

If we were to apply this to the initial output of the foldr/build translation above, this code would be obtained:

```
build (\c n ->
    let f1 x b1 =
        let f2 y b2 =
            if y < 10
            then c y b2
            else b2
        in let f3 z b3 = f2 (x * (z * 2)) b3
           in foldr f3 b1 [4, 5, 6]
    in foldr f1 n [1, 2, 3])
```

This implementation really is better than the equivalent TQ code of section 3.4.2, as no intermediate lists are generated.

My implementation extended the foldr/build translation of classical list comprehensions to cope with the extended syntax, and the resulting system is presented in figure 3.6. Translating the example extended comprehension of section 3.4.2 using those rules, the following following Core code is obtained:

```
build (\c n ->
  let g (z, y) = z
  in let f (z, y) b = c (the z, y) b
     in foldr f n (map unzip_2 (groupWith g (build (\c n ->
                  let f2 (z, y) b2 = c (z, y) b2
                  in foldr f2 n (take 3 (build (\c n ->
                                 let f3 z b3 = let f4 y b4 = c (z, y) b4
                                               in foldr f4 b3 [1, 2, 3, 4]
                                 in foldr f3 n ["foo", "bar", "baz"]))))))))
```

As you may notice, although many intermediate lists are being constructed, there are actually no opportunities to apply the fusion rule in this code: it seems as if the foldr/build translation has not won us anything. In fact, by defining auxiliary functions in terms of `foldr` and `build` and exploiting GHC's intricate system of rules and inlining, it is possible to achieve a very high degree of fusion in this example. This process of defining and testing various library functions tuned for fusion and the non-obvious stack behaviour of Haskell [27] consumed the majority of the time spent on the implementation. A brief treatment of this process is included in appendix B.

It turns out that fusion is especially important in code containing extended list comprehensions because every use of one of the new qualifiers causes an intermediate list to be generated, many of which can be safely removed.

3.4.3 The Use Of Sorting With The Schwartzian Transform

One possible way in which I might have tried to improve the efficiency of the extension is by redefining the *sortWith* function to make use of the so-called Schwartzian transformation[29]. This effectively memoizes calls to the projection function that obtain the per-element value to order with. Hence it can potentially increase performance if that function is expensive to compute. However, the trade-off is that more memory is used up: at least one extra word will be added to the size of each element of the list being sorted as each is tagged with the corresponding projected value. This will at worst double the size of the list being sorted because lists of the unboxed values, which are the only thing that may be less than a word in size, are rare in practice. An implementation of this algorithm for *sortWith* would look as follows:

```
sortWith ::  Ord b => (a -> b) -> [a] -> [a]
sortWith f xs = map snd (sortBy (\(x, _) (y, _) -> x `compare` y) (map (\x -> (f x, x)) xs))
```

There is a significant downside to using this strategy in terms of heap space required. This, combined with the expectation that most uses of the *sortWith* function with the extended comprehension syntax will make use of an inexpensive projection function such as a record accessor, I have come to conclude that this implementation would not be an improvement. Indeed, if the user does have the explicit desire to use the Schwartzian transformation all they need to do is to compute the per-element projection value in a `let` qualifier just before using a sorting or grouping qualifier which then sorts or groups `by` that projected value. Consider the following generalized list comprehension which will be forced to execute a long running computation $\Theta(N log N)$ times:

Top Level Translation

$$\llbracket \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]} \rrbracket_{FB} = \texttt{build (\textbackslash c n -> } \llbracket \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]}\ \texttt{c n} \rrbracket_{FBQ}\texttt{)}$$

Inner Translation

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ \epsilon\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = c\ \llbracket e \rrbracket\ n
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ w\ \texttt{<-}\ e_l\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{let f}\ \llbracket w \rrbracket\ \texttt{b = } \llbracket \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]}\ c\ \texttt{b} \rrbracket_{FBQ} \\
&\quad\ \texttt{in foldr f}\ n\ \llbracket e_l \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ \texttt{let}\ w\ \texttt{=}\ e_v\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{let}\ \llbracket w \rrbracket\ \texttt{=}\ \llbracket e_v \rrbracket\ \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ}
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ e_b\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{if}\ \llbracket e_b \rrbracket\ \texttt{then}\ \llbracket \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ}\ \texttt{else}\ n
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then}\ f)\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V \rrparenthesis\ \texttt{<-}\ \llbracket f \rrbracket\ \llbracket \texttt{[}\ \llparenthesis V \rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]} \rrbracket\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\quad\ \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then}\ f\ \texttt{by}\ e_l)\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{let g}\ \llparenthesis V \rrparenthesis\ \texttt{=}\ \llbracket e_l \rrbracket \\
&\quad\ \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V \rrparenthesis\ \texttt{<-}\ \llbracket f \rrbracket\ \texttt{g}\ \llbracket \texttt{[}\ \llparenthesis V \rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]} \rrbracket\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\quad\ \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group by}\ e_l)\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{let g}\ \llparenthesis V \rrparenthesis\ \texttt{=}\ \llbracket e_l \rrbracket \\
&\quad\ \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V \rrparenthesis\ \texttt{<- map}\ unzip_{|V|} \\
&\qquad\qquad \texttt{(groupBy g}\ \llbracket \texttt{[}\ \llparenthesis V \rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]} \rrbracket\texttt{),}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\quad\ \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group using}\ f)\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V \rrparenthesis\ \texttt{<- map}\ unzip_{|V|} \\
&\qquad\qquad (f\ \llbracket \texttt{[}\ \llparenthesis V \rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]} \rrbracket\texttt{),}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\quad\ \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_{qs}(qs))
\end{aligned}$$

$$\begin{aligned}
&\llbracket \texttt{[}\ e\ \texttt{|}\ (qs_{inner}\texttt{,}\ \texttt{then group by}\ e_l\ \texttt{using}\ f)\texttt{,}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\ = \texttt{let g}\ \llparenthesis V \rrparenthesis\ \texttt{=}\ \llbracket e_l \rrbracket \\
&\quad\ \texttt{in}\ \llbracket \texttt{[}\ e\ \texttt{|}\ \llparenthesis V \rrparenthesis\ \texttt{<- map}\ unzip_{|V|} \\
&\qquad\qquad \texttt{(}\ f\ \texttt{g}\ \llbracket \texttt{[}\ \llparenthesis V \rrparenthesis\ \texttt{|}\ qs_{inner}\ \texttt{]} \rrbracket\texttt{),}\ qs\ \texttt{]}\ c\ n \rrbracket_{FBQ} \\
&\quad\ \text{where } V = BV_{qs}(qs_{inner}) \cap (FV_e(e) \cup FV_e(e_l) \cup FV_{qs}(qs))
\end{aligned}$$

Figure 3.6: Foldr/build desugaring for extended list comprehensions

```
[x |x <- [1..n]
   , then sortWith by longRunningComputationOn x]
```

Here is a version to which I have applied the Schwartzian transformation so that the computer only need run that computation $\Theta(N)$ times at the cost of a constant factor increase in the transient memory usage of the comprehension:

```
[x |x <- [1..n]
   , let cached = longRunningComputationOn x
   , then sortWith by cached]
```

As you can see, our comprehensions rather elegantly subsume this common transformation.

3.5 Extension: Fusable List Literals

During my investigation into desugaring of the extension, I noticed that programs which included *foldr* calls over list literals, such as the following contrived example, did not enjoy the benefits of fusion:

```
spqr x y = foldr (*) 1 [1, 2, x, y, 3, 4]
```

This is in part due to a deliberate design choice: in some cases it is not desirable for fusion to occur. For example, the compiler could easily automatically transform all such programs by implementing the foldr/cons rewrite rule:

$$\texttt{foldr}\ c\ n (x : l) = c\ x\ (\texttt{foldr}\ c\ n\ l)$$

This would cause the example program to have this final form:

```
spqr x y = 1 * 2 * x * y * 3 * 4 * 1
```

Unfortunately, in this program the opportunity to convert the list `[1, 2]` into shared read-only data has been lost: instead it must be represented as function applications. This can lead to bad code bloat in general. However, the key insight is that only the part of original list after the x and y can be floated out in this way, as everything preceding that in the list must be constructed dynamically due to the presence of the variables, which might take on any value at all! This even applies to the literals 1 and 2 at the head of the list since the "next" pointers of their cons cells depend on that dynamic allocation. Hence the optimal fused version of the program is as follows:

```
spqr x y = 1 * 2 * x * y * (foldr (*) 1 [3, 4])
```

It is impossible to express this subtle constraint about dynamic vs. static list elements in the current GHC rewrite rules system, so I implemented this optimization not with a modified foldr/cons rule but rather by immediately desugaring the list literal as follows:

```
spqr x y = foldr (*) 1 (build (\c n -> c 1 (c 2 (c x (c y (foldr c n (3 :  4 :  [])))))))
```

This allows foldr/build fusion to take place as normal while retaining a visible "static tail" suitable for floating out. If the literal had appeared in a context where fusion did not occur then *build* would eventually be inlined and another rewrite rule would transform the resulting `foldr (:) [] (3 : 4 : [])` syntax tree into the more efficient `3 : 4 : []`.

3.6 Extension: The Static Argument Transformation

The static argument transformation (SAT) was first described by Santos in his PhD thesis [23]. It's raison d'être is to transform recursive functions which have arguments that are "static" in that they are invariant under recursion into equivalent forms where those arguments become free variables that are fixed by an enclosing environment. For example, the SAT would transform this function:

```
foldr c n [] = n
foldr c n (x:xs) = c x (foldr c n xs)
```

Into this one:

```
foldr c n = foldrp
    where foldrp [] = n
          foldrp (x:xs) = c x (foldrp xs)
```

Operationally this corresponds to transforming redundant stack argument-pushing operations into references to values via a closure. However, as the closure itself is an implicit static argument to the function, it turns out that this transformation is only worthwhile when at least two arguments are static. Furthermore, for static bodies (e.g. `foldr'`above) that typically recurse only a few times (e.g. imagine that most call sites give `foldr` an empty list) the cost of creating the additional static body thunk outweighs the savings of reducing the number of arguments to be pushed onto the stack. This subtlety means that Santos gave up on trying to find a satisfactory set of conditions to use to apply the SAT and the relevant code was commented out and left to rot in the GHC source tree.

I saw during my investigation of desugaring that I could obtain large improvements in execution time by explicitly applying such a transformation to selected parts of my code. Thus motivated, I did considerable work to restore Santos's implementation to a working state and experimented with various heuristic conditions under which to apply it, eventually settling on using it on functions with more than one static argument.

Chapter 4

Evaluation

In this section I discuss the comprehensive evaluation regime that I subjected my syntax extension to. In particular, I show that the extension is correct with respect to a suite of automated tests, that it has no negative effect on the performance of existing Haskell programs and that the desugaring developed in the course of the previous section is efficient. I will finish by reviewing the project in the context of the proposal to see how it has performed with respect to the original goals.

4.1 Evaluating Correctness

GHC has thousands of users, both hobbyists and from industry, and so it is imperative that any feature it implements is industrial strength and well tested. To this end I created a comprehensive suite of tests designed to stress my extension to the compiler in several dimensions and evaluate its correctness under these situations. Examples of the things that my automated tests examined are:

- All permutations of the extensions new syntactic forms and many of their combinations with each other.
- Uses of the extended syntax over lists which contained 100 binders, to stress big tuple support (see section 2.3).
- Error messages encountered if the extension is disabled but the syntax is used.
- Errors encountered when the user makes common scoping mistakes with the extension.
- Type checking stress tests that attempt to supply unacceptable functions or values to the extension.

All the tests created were verified as succeeding and integrated into the automated testing suite that comes as part of GHC so that the developers can have assurance that any later changes made to the compiler do not inadvertently break the extension.

Furthermore, GHC has a standard suite of regression tests which can be used to validate the operation of the compiler. I used these to verify that the changes I had made had not adversely affected the correct operation of unrelated parts of the compiler.

4.2 Evaluating High Level Performance Impact

In order to ensure that my modifications to GHC have not have a detrimental effect on existing Haskell programs, I carried out a high level performance comparison (using the standard nofib benchmark suite [20]) between the versions of the compiler just before and just after the final patch was applied. This benchmark suite is very comprehensive set of 91 tests that make use of many features of the Haskell 98 language. The anticipated result was there there should be no change to the performance of the compiled programs, as the final patch was explicitly designed to be local in its changes and thus its effects on the rest of GHC. As the benchmark suite is written in standard Haskell 98 it cannot make use of my syntax extension, and hence in theory no substantial portion of my new code is even being executed.

The results from this evaluation were largely as expected. Allocations recorded and generated executable size did not change for any of the programs in the suite. Program execution times did vary between those compiled with the two GHC versions, but the difference between any two corresponding runtimes was at most 1.3%, and the mean change had a magnitude of only 0.1%. Although I present no formal statistical evidence here, this appears to justify the hypothesis that the two versions of GHC I tested compiled the nofib suite very similarly.

The most interesting outcome for this benchmark is the changes in compile times reported: they fell in the range -3.4% to +6.1%, with a mean of +1.2%. I would naturally expect more variability for the compile times than the runtimes because the benchmark suite runs the programs it tests multiple times and our results report the mean of these runtimes, whereas the compilation of each program occurs only once. Nonetheless, the shift in the mean seems significant with respect to this variation, leading to the inescapable conclusion that compile times probably have increased.

The increase in compile time is small enough that it is acceptable, but it begs the question as to what exactly caused it. I would speculate that is is due to the modifications made to the parser, as all other code added by my extension should not have been run for programs that do not contain the generalized comprehensions (and nofib programs certainly do not). The new parser has to deal with several new keywords, as well as the use of old keywords in new positions, and has a more convoluted mechanism for the parsing of list comprehensions which was added to incorporate the generalization: these factors could all conceivably cause the performance of the parser to degrade, leading to the observed compile time increase.

The full results are included in an appendix, in appendix A.

4.3 Evaluating Local Performance Impact

Determining whether I have had any discernible effect on a selection of typical Haskell programs as I did in the last section is useful, but it might obscure localized performance changes to list comprehensions as these typical programs are unlikely to heavily use list comprehensions. To solve this, I created a set of simple micro-benchmarks designed to expose any time or space consumption issues with comprehensions. The final suite consisted of these programs:

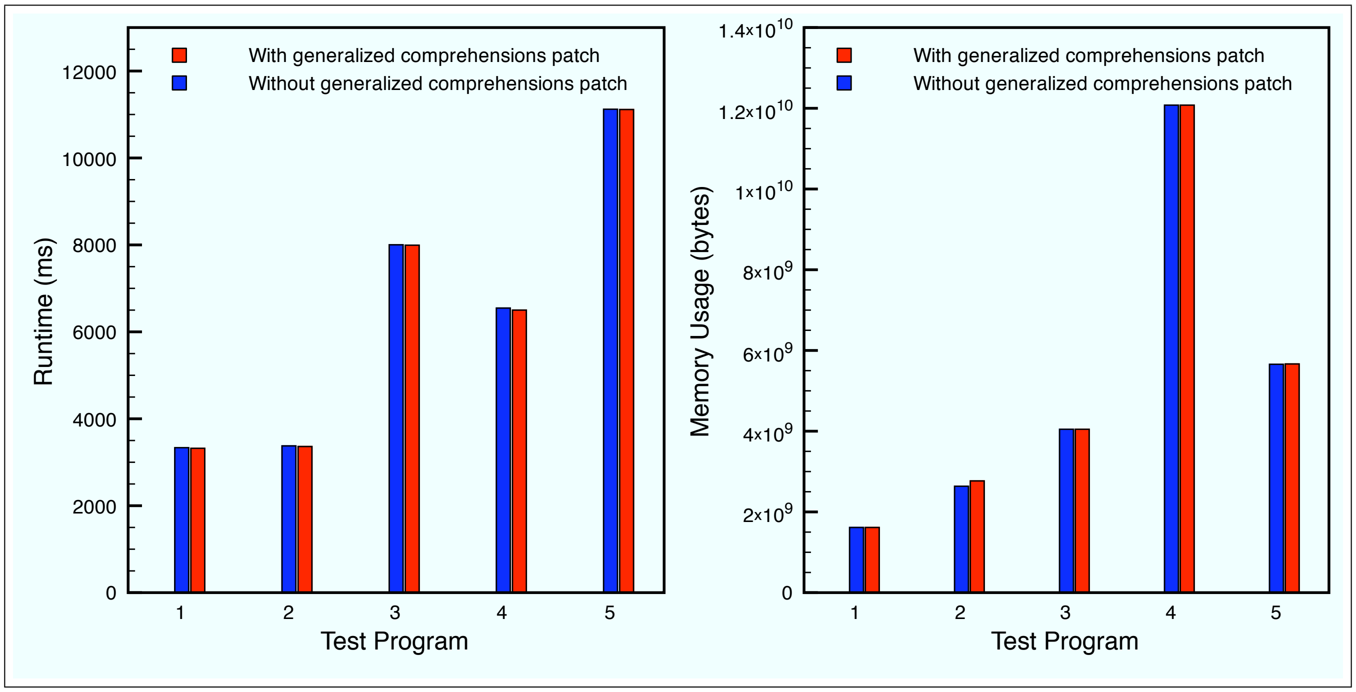

Figure 4.1: Runtime and memory usage of list comprehension benchmarks before and after list comprehensions patch

	Program	Description/Intended Performance Test
1	QuoteVolume	Queries an in-memory database of stock price quotes to determine some statistics about them. Tests database query like operations
2	QuoteChanges	Queries an in-memory database of stock prices to find large changes, incorporating an artificial use of the parallel list comprehensions extensions Tests database query like operations with zipping
3	PythogoreanTriples	Constructs a list of all Pythagorean triples in a particular range using a list comprehension Tests comprehensions involving many uses of nested lists (e.g. Cartesian products)
4	MatrixMultiplication	Performs some linear algebra in a naïve way by making use of many comprehensions and intermediate lists Tests nested comprehensions in use on regular data structures without filtering
5	Quicksort	Sort a list using a pure comprehensions based approach to quicksort partitioning Tests comprending over heterogeneous data while making use of a lot of filtering

The results of this evaluation are shown in figure 4.1. They show clearly that once again (as anticipated) my patch had no effect on the performance of standard list comprehensions.

4.4 Evaluating The Final Desugaring Schemes

Throughout this project I have developed two versions of the generalized comprehension desugaring: the foldr/build and the TQ style translations. It is interesting to consider how these two alternatives differ in performance, for the following reasons:

- It gives us a rough idea about how often fusion is taking place.

- It may reassure us that fusion is actually improving the performance of our programs.
- It is possible to make decisions about how to develop GHC based on the relative performance of the alternatives. For example, perhaps if the TQ style translation turned out to be acceptably fast, given that fusion introduces some complexity to the compiler, the developers could perhaps remove it and move to using TQ exclusively.

Similarly to the methodology of section 4.3, I created a suite of micro-benchmarks designed to exercise the performance of the extended list comprehensions in various use cases:

	Program	Description/Intended Performance Test
1	RayTracer	Uses a single list comprehension to implement a simple ray tracer Tests comprehensions over regular data structures with nesting
2	BearMarketeer	Queries an in-memory database of equity index prices to find the longest known runs of downwards trending (so-called "bear") markets Tests multiple sorts and groups operating linearly over a single list
3	HumanResources	Does a simple query for properties of salaries on an employees list Tests database query like operations
4	Frobnicate	An artificial benchmark designed to exercise list transformations Tests comprending with transformations being used on irregular data structures
5	Database	Performs a complicated multi-list query, emulating a moderately complicated SQL query over several tables which incorporates inner joins Tests multi-table database query like operations

In order to provide a baseline for comparison, I also compared the performance of the two styles on the micro-benchmarks for standard list comprehensions I had previously developed. The results for these benchmarks are shown in figure 4.2, and the results of the benchmarks making use of extended list comprehensions are shown in figure 4.3. From these graphs I have drawn the following conclusions:

- The performance of the two styles of translation are overall very similar: indeed, they are nigh-on indistinguishable from each other in most cases.
- Overall, the foldr/build translation seems to yield slightly faster programs than the TQ translation, but there is no advantage either way on memory usage.
- Foldr/build only seems to be slower than TQ in one case: the Frobnicate benchmark.

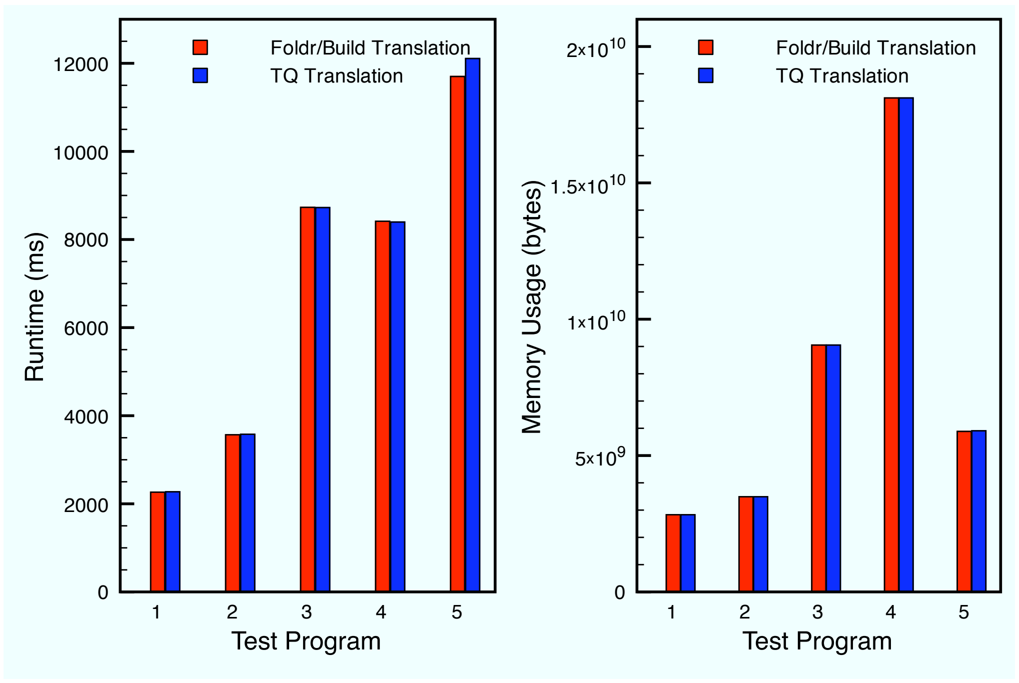

Figure 4.2: Runtime and memory usage of unextended list comprehension benchmarks with desugaring styles

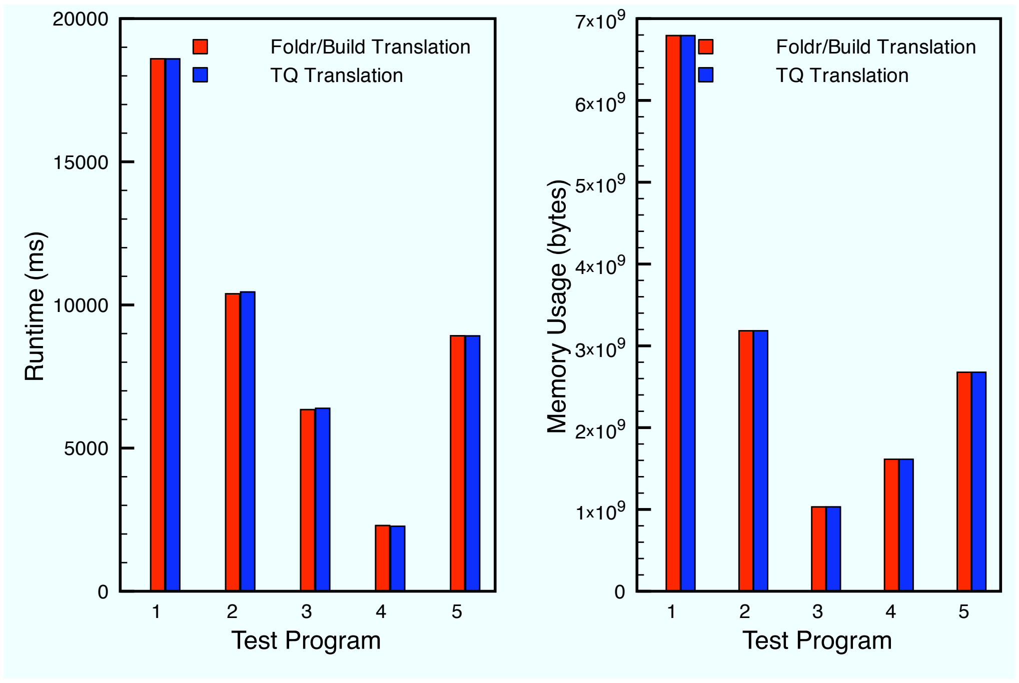

Figure 4.3: Runtime and memory usage of extended list comprehension benchmarks with desugaring styles

Program	Size Change	Allocations Change	Runtime Change
ansi	-0.3	-13.8	0.00
cacheprof	+0.0	-2.6	-2.1
circsim	+0.0	+0.0	+1.4
fluid	+0.2	-0.4	0.01
fulsom	+0.0	+0.0	+1.7
gamteb	+1.1	+0.3	0.11
gg	+0.0	-0.4	0.01
lift	+0.0	-8.7	0.00
maillist	+0.0	+0.5	0.05
rewrite	+0.0	-26.3	0.02
wheel-sieve1	+0.0	+0.0	+1.8
wheel-sieve2	+0.0	+0.0	+2.5
Min	-0.3	-26.3	-2.5
Max	+2.0	+0.5	+2.5
Geometric Mean	+0.0	-0.7	+0.1

Figure 4.4: The most significant effects on the nofib [20] benchmark suite of enabling fusable list literals

The fairly minimal difference between the two schemes may be somewhat unexpected as Gill reports in his PhD thesis [7] (which further developed the notion of shortcut fusion in the style of foldr/build from his original paper [8]) that he saw average improvements in the runtime and allocations in the nofib suite of 3% and 5% respectively when using his scheme instead of TQ. However, we must take into account that his results were obtained in a compiler where he was enabling foldr/build wholesale across the entire compiler rather than for list comprehensions individually, so they include the improvements made to Haskell functions that are not defined in terms of list comprehensions but are nevertheless fusible. Furthermore, nofib is a considerably better benchmark suite than the artificial one I have created here, and the added complexity of its constituent programs will almost certainly mean that more inter-comprehension list fusion opportunities are exposed than in my examples.

Overall it is encouraging that the performance characteristics of my new desugaring with respect to whether foldr/build fusion is in use or not are very similar or identical to those of the standard desugaring.

Returning to the reasons for performing this benchmark that were outlined at the start of this section, I find that I am not reassured about the value of the foldr/build style desugaring at all, and that it is indeed tempting to remove it and use the TQ style in all cases! However, such a modification should not be carried out without a more comprehensive benchmarking effort being undertaken. Such an effort could for example ensure that the nofib benchmarking suite does not have its performance adversely affected by the change, which would reassure us that these micro-benchmarks really are representative of larger programs.

4.5 Extension: Evaluating Fusable List Literals

The standard Haskell test suite nofib [20] was used to evaluate the effect of enabling the fusable list literals optimization in GHC. The empirical results of are encouraging: on average no code size increase was observed with them enabled (though there was one outlier that saw a 2% increase) and runtime stayed roughly constant. However, two programs in the suite that made extensive use of comprehensions and list literals enjoyed a 26% reduction in allocations performed with the additional optimization. The most significant results are summarized in figure 4.4.

Program	Size Change	Allocations Change	Runtime Change
atom	+0.0	+0.0	-5.9
circsim	+0.0	+0.0	-2.4
comp_lab_zift	+0.0	+0.0	-4.8
constraints	+0.0	+0.0	-0.6
cryptarithm1	+0.0	+0.0	-1.7
fulsom	+0.0	+0.0	+0.9
integrate	+0.0	+0.0	-3.1
lcss	+0.0	+0.0	-0.6
life	+0.0	+0.0	-0.8
maillist	+0.0	+0.5	0.06
multiplier	+0.9	-13.6	0.15
nucleic2	+0.0	+0.0	+1.1
parstof	+0.9	-2.0	+2.3
power	+0.0	+0.0	-1.0
scs	+0.0	+0.2	+0.5
simple	+0.0	+0.0	-1.2
solid	+0.0	+0.0	-0.4
typecheck	+0.0	+0.0	-0.8
wave4main	+0.0	+0.0	-0.8
wheel-sieve1	+0.0	+0.0	+0.3
wheel-sieve2	+0.0	+0.0	-2.8
Min	+0.0	-13.6	-5.9
Max	+0.9	+0.5	+2.3
Geometric Mean	+0.0	-0.2	-0.8

Figure 4.5: The most significant effects on the nofib [20] benchmark suite of enabling the static argument transformation

4.6 Extension: Evaluating The Static Argument Transformation

For this evaluation, I benchmarked the effects of enabling the static argument transformation when compiling nofib. The most significant results are summarized in figure 4.5. Overall, the effect of the transformation is good: allocations and runtime reduce significantly for some programs, at the cost of minor increases in allocations and runtime for a smaller number of outliers.

4.7 Evaluation Against Project Goals

An essential part of this project is ensuring that it has met the goals originally outlined in the proposal (see section D). All of the goals have been met or exceeded by the project, and in order to prove this I will examine and present evidence for each goal in the following sections.

4.7.1 Proposed Project Substance

To recap, the substance of the project was set out in the proposal as follows:

1. Implementation in the Glasgow Haskell Compiler of a form of the ordering syntax from the aforementioned list comprehension paper.
2. Evaluation of the effect of a naive implementation of this syntax on the performance of existing programs.
3. Ensuring that the new syntax has the desired semantics by a process of rigorous testing or another appropriate method.
4. Evaluation of the characteristics of different desugarings for the ordering construct within the compiler given the experience of implementing a simple variant.
5. Implementation of any improvements to the desugaring that can be determined and an exploration of the time, space and compile time characteristics of this resulting system.
6. Writing the dissertation.

If I take these points one by one:

1. The ordering syntax proposed in the paper could not be implemented directly, as explained in section 3.1.2. However, a suitable alternative was developed in section 3.1.5 which was implemented. What is more, this goal was exceeded as the syntax for extended list comprehension grouping was refined, implemented and performance and correctness tested as well.
2. This was accomplished by section 4.2, where I performed a test on the effect of the new desugaring on the performance of existing programs in the guise of the nofib standard benchmarking suite.
3. As I explain in section 4.1 a comprehensive automated test suite was developed to assess the correctness of the extension, fulfilling the requirements of this goal.
4. Section 3.4.2 explains that considerable time was spent on evaluating the performance of competing library function implementations in order to optimize the effectiveness of the foldr/build comprehensions desugaring, fulfilling this goal.
5. The implementation of improvements is covered by the above point. I have then further explored the performance of the final system in sections 4.2, 4.3 and 4.4, comprehensively meeting the requirements of this goal.
6. The dissertation has indeed been written, as the reader is hopefully able to discern.

4.7.2 Success Criterion

The project proposal stipulated that the following program, making use of the generalized ordering construct, would be able to be compiled by GHC and the resulting executable executed by the completion of the project:

```
module Main where

people = ["Richard", "Jane", "Carol", "Simon", "Robert"]

list = [(x, y) |x <- people |y <- [5,4..1], (length x) > 4, order by ((head x), y)]

main = putStrLn (show list)
```

This program makes use of "zip" qualifiers. I have elided discussion of these throughout the dissertation as they only obscure the presentation, but my implementation does indeed handle them. The output of the example was specified to be the following:

```
[("Carol", 3), ("Robert", 1), ("Richard", 5), ("Simon", 2)]
```

Although the syntax of the program specified in the success criterion is not, as I mentioned in section 3.1.5, valid under the implemented system the following spiritually equivalent program is indeed able to execute successfully and produce the specified output in the modified GHC system:

```
module Main where

import GHC.Exts(sortWith)

people = ["Richard", "Jane", "Carol", "Simon", "Robert"]

list = [(x, y) |x <- people |y <- [5,4..1], (length x) > 4, then sortWith by ((head x), y)]

main = putStrLn (show list)
```

What is more, programs involving the other forms of the "ordering" construct and all the grouping constructs proposed by the motivating paper are able to be compiled and executed successfully in this way, so the success criterion has been not just met but exceeded. I have also implemented, as extensions, one novel and one other compiler optimization that came out of my work on improving the efficiency of the desugaring, as documented in sections 3.5, 3.6, 4.5 and 4.6. These will benefit all future GHC-compiled programs.

4.7.3 Further Success Claims

My project proposal also included the following statement:

> The project will further be evaluated on the basis of the space and time complexity of programs using the new mechanisms for list comprehensions versus list comprehensions compiled under the current system which supports only the restricted projection and selection operations. To this end GHC provides an extensive test suite, called "nofib", for evaluating the impact of compiler changes on typical Haskell programs. However, as these programs clearly make no use of the proposed additional capabilities for comprehensions a further set of tests will be developed by me as part of the project to gauge the relative performance of the new capabilities versus their classical implementations in Haskell 98.

This prophesy has indeed been fulfilled by the production of a number of micro-benchmarks I performed throughout implementation (excluded from this dissertation by space constraints) and by the 10 micro-benchmarks produced in sections 4.3 and 4.4.

Chapter 5

Conclusions

The previous sections have conclusively shown that the project has met all of the criteria laid out in the proposal and thus can be considered a success.

My patches implementing this extension and the compiler optimizations of sections 3.5 and 3.6 have been accepted for inclusion within the primary GHC codebase. This vouches for the quality of the work that has been produced and that the GHC developers consider it a worthwhile extension to the language and their compiler. The process of getting this included required me to to produce user documentation for the extension, incorporating information on how to enable the extension and a guide to its use. This will be included in the next release of the GHC Handbook, which accompanies a compiler release.

The project has been marked with occasional periods of frustration on my part due to the complexity of working with the massive and in places Byzantine GHC codebase and build system, but there is little that could have been practically done to mitigate this. Overall the project has gone very smoothly, to which I owe an enormous debt of gratitude to my supervisor, Simon Peyton Jones, who has never been short of ideas and useful suggestions when a problem has arisen.

In the next section I will suggest further work that could usefully be carried out in areas related to this project, and then finally I will close with a survey of related projects that I have not mentioned elsewhere in the dissertation.

5.1 Further Work

There are a number of avenues by which the work in this dissertation could be extended: two generalizations, two possible performance improvements and one human factors style study. I discuss these possibilities in detail in the following sections.

5.1.1 Reintroduction Of Qualifier Associativity

Section 3.1.3 outlined the problems with the generalized list comprehension qualifier associativity syntax proposed by [14] and examined the alternatives, but found none entirely satisfactory. An obvious avenue for future work is to carry out a survey of Haskell users and possibly other groups of programmers to find out conclusively whether this feature is desired, and if it is what syntax should be used to introduce it. Actually implementing this feature is fairly straightforward as the actual behaviour called for is catered for by my implementation already: the parser would just have to be modified slightly to enable the feature.

5.1.2 Generalization To Arbitrary Monads

I will assume in this section that the reader is familiar with the category theoretic monad structure that is a frequent occurrence in Haskell. Those unfamiliar with the topic can learn more about it in many places and in particular in any of the accessible papers on the topic published in the 1990s by Wadler such as [32].

A possible view of the Haskell 98 list comprehension syntax is as a form of the monadic `do` notation specialized to the list monad: the monad that traditionally is used to represent nondeterministic choice. To see how this works, consider these two equivalent programs:

List Comprehension Style	General Monadic Style
`result = [(a, b, c)`	`import Control.Monad`
`           |a <- [1..10]`	
`           , b <- [1..10]`	`result = do`
`           , let sumSq = a^2 + b^2`	`    a <- [1..10]`
`           , c <- [1..10]`	`    b <- [1..10]`
`           , sumSq == c^2]`	`    let sumSq = a^2 + b^2`
	`    c <- [1..10]`
	`    guard (sumSq == c^2)`
	`    return (a, b, c)`

One interesting point is that I actually make use of the fact that the list type is an instance of `MonadPlus` as well as `Monad` in translating the guard qualifier. The `MonadPlus` class is related to monoids and expresses a monad with some extra structure that lets you "append" two instances of a monadic value and obtain an identity value with respect to that append operation. In the case of lists the append operation is just list concatenation and the identity value is the empty list, which is what is eventually used in the implementation of the standard `MonadPlus` based function `guard` seen in the above sample.

It is impossible to encode the "zip" list comprehension qualifier in the monadic style, since `zip` is not a monadic operation.The obvious implementation of a monadic operation with the appropriate type signature (i.e. $\texttt{Monad}\, m \Rightarrow m\, a \rightarrow m\, b \rightarrow m\,(a, b)$) will just implement the Cartesian product when applied at the list monad, which is not what we want at all.

However, it would indeed be fairly straightforward to extend the generalized list comprehensions described in this dissertation to arbitrary monads. The translation required was first articulated by Michael Adams [2] in a response to the original paper and you can read his full description there. To give you a flavour of how such a translation might work, I will translate this simple monadic code that makes use of one of our grouping qualifiers:

```
result = do
    a <- ma
    b <- mb
    then group by b using runs
    c <- mc
    return (a, b, c)
```

A desugaring translation similar in spirit to but slightly simpler than that proposed by Adams would translate this like so:

```
result =
    runs
        (\(a, b) -> b)
        (ma >>= (\a ->
         mb >>= (\b ->
         return (a, b))))
     >>= (\results ->
         let a = fmap (\(a, _) -> a) results
             b = fmap (\(_, b) -> b) results
         in mc >>= (\c ->
         return (a, b, c)))
```

The effect of the translation might be easier to understand if you consider that in the case of simple lists the `group using` functions had the type $\forall\alpha.(\alpha \to \tau) \to [\alpha] \to [[\alpha]]$ but in the version generalized to monads they must instead inhabit the type $(\texttt{Monad}\, m, \texttt{Functor}\, n) \Rightarrow \forall(\alpha \to \tau) \to m\,\alpha \to m\,(n\,\alpha)$. Essentially the "outer" list in the type has been generalized to a monad and the "inner" list has been generalized to any functor (hence *fmap* in our example desugaring above). Counter-intuitively, this desugaring is not actually significantly more work to implement on top of the existing syntax extension. However, as there is currently no use case for this any such any implementation would most likely just been a little-used distraction for the GHC maintainers to deal with.

5.1.3 Query Optimization

There are many cases in which it may be beneficial to rewrite a query comprehension so that it performs its the operations implied by its qualifiers in a different order. A simple example of this is filter promotion, where filters (i.e. guard qualifiers) are done as early as possible (i.e. as soon as the variables they mention are introduced) to reduce the volume of data that later stages of the comprehension have to process. Due to the new generality of list comprehensions introduced by the extension implemented by this project there are many more possible optimizations that could take place. For example, if the compiler were to spot a list comprehension that grouped by an identifier that it knew to occur in sorted order in the list at that point (perhaps because the most recent transform qualifier used it to call *sortWith*), it could eliminate the preliminary sorting stage that the *groupWith* function usually has to do.

Some authors have previously tried to suggest ways in which query improvement can be applied to list comprehensions [25], but it would be interesting to see how much more of the standard literature on query optimization could be applied to these problems in our generalized setting, and if any further enrichments to the syntax could expose further opportunities for optimization to the compiler. One possible avenue for work might be to use the work of Lars Pareto's PhD thesis [19], in which he introduces a notion of statically sized types, as a basis for a cost metric on lists to allow the compiler to make informed decisions about best to optimize a given query.

5.2 Related Work

This dissertation contains many references to related fields of work, especially in section 5.1. However, there are a number of other related projects which the reader may find interesting, which I will very briefly cover in this section.

List comprehensions have their origin in David Turner's functional language KRC [26], and their semantics were first formalised by Wadler [11] in terms of the TQ translation I discussed in this dissertation. Wadler later generalised them to monad comprehensions [30] in the style detailed in section 5.1.2, providing a convenient syntax for many operations beyond those for lists.

List comprehensions were adopted as a querying system by Trinder's functional database system [24], which exploited the regular compiler-visible structure of the list comprehensions to optimize the queries issued to the system by a user [25]. However, Trinder does not seem to have considered the common database operations of sorting or grouping in his thesis and so did not have an equivalent to the constructs discussed in this dissertation.

Kleisli [33] is a system designed for solving data integration problems at the Internet level. It does this by supporting a core functional query language called CPL which is an interface into a backend that can operate in a uniform manner over remote, heterogeneous data sources. Kleisli incorporates support for moving parts of the query the user makes to the remote server on which data resides, and also includes a query optimizer that tries to make the best use of time, memory and bandwidth resources by rearranging queries in much the same manner I discussed in section 5.1.3. However, in doing this it has the advantage over GHC of a cost metric on the collections being comprehended over (i.e. the likely size of the data source as reported by the remote server), making this optimization much more practical than in Haskell today. The version of CPL described in that paper does not have support for sorting or even grouping directly, however. Grouping is expressed by using an explicit inner join that makes use of a nested comprehension in combination with a guard.

LINQ [5] is a feature of the C# and VB.NET programming languages that is designed to enable many of the same possibilities for those imperative languages as list comprehensions have in functional programming for many decades. However, unlike list comprehensions it was designed at the outset to incorporate support for grouping and ordering, and includes powerful features for metaprogramming on the queries. This allows LINQ providers (which may have targets as diverse as databases, XML files or web services) to intelligently map LINQ queries to the underlying abstractions e.g. by incorporating guards in the LINQ query into a SQL statement as a WHERE clause.

Although LINQ has these powerful features, its syntax is actually less general than that expounded in this paper in that it allows only one form of list transformation (ordering by an expression) and only the obvious form of grouping, which precludes for example the use of the *partitionToList* function that I introduced in section 1.2.6. This is motivated by a desire to match the capabilities of SQL.

Bibliography

[1] Anthony Aaby. Haskell tutorial. Available at `http://cs.wwc.edu/KU/PR/Haskell.html`.

[2] Michael Adams. Comprehensive comprehensions wiki page. Available at `http://haskell.org/haskellwiki/Simonpj/Talk:ListComp`, 2007.

[3] Maximilian Bolingbroke. GHC wiki: SQLLikeComprehensions. Available at `http://hackage.haskell.org/trac/ghc/wiki/SQLLikeComprehensions`.

[4] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all, 2007. Available at `http://www.cse.unsw.edu.au/~dons/papers/stream-fusion.pdf`.

[5] Brian Beckman Erik Meijer and Gavien Bierman. Linq: Reconciling objects, relations, and xml in the .net framework. Available at `http://research.microsoft.com/~emeijer/Papers/LINQSigmod.pdf`, 2006.

[6] Simon Peyton Jones et al. The GHC Commentary. Technical report.

[7] Andrew Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, The University of Glasgow, January 1996.

[8] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation, 1993. Available at `http://research.microsoft.com/~simonpj/Papers/deforestation-short-cut.ps.Z`.

[9] Andy Gill and Simon Marlow. Happy parser generator. Available at `http://www.haskell.org/happy/`.

[10] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[11] Simon Peyton Jones. *The Implementation Of Functional Programming Languages*. Prentice Hall, 1987.

[12] Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. The Haskell 98 Language Report (Revised Version). Technical report, Microsoft Research Cambridge, 2002. Available at `http://www.haskell.org/onlinereport/`.

[13] Simon Peyton Jones and Simon Marlow. The Glasgow Haskell Compiler. Available at `http://www.ghc.org`.

[14] Simon Peyton Jones and Phil Wadler. Comprehensive comprehensions: comprehensions with 'order by' and 'group by', 2007. Available at `http://research.microsoft.com/~simonpj/papers/list-comp/index.htm`.

[15] Simon Marlow. Alex lexer analyser generator. Available at `http://www.haskell.org/alex/`.

[16] Manuel Chakravarty Martin Sulzmann and Simon Peyton Jones. System F with type equality coercions, 2007. Available at `http://research.microsoft.com/%7Esimonpj/papers/ext%2Df/`.

[17] Microsoft. C# yield keyword. Available at `http://msdn2.microsoft.com/en-us/library/9k7k7cf0.aspx`.

[18] Microsoft. Context sensitive keywords (c++). Available at `http://msdn2.microsoft.com/en-us/library/8d7y7wz6.aspx`.

[19] Lars Pareto. *Types for crash prevention.* PhD thesis, 2000.

[20] W. Partain. The nofib benchmark suite of Haskell programs. Available at `http://citeseer.ist.psu.edu/partain93nofib.html`, 1993.

[21] Joseph Fasel Paul Hudak, John Peterson. A gentle introduction to haskell. Available at `http://www.haskell.org/tutorial/`.

[22] Winston Royce. Managing the development of large software systems. Available at `http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf`, 1970.

[23] A Santos. Compilation by transformation in non-strict functional languages. 1995.

[24] Phil Trinder. A functional database. Technical Report CSC/90/R10, 1990.

[25] Phil Trinder and Philip Wadler. Improving list comprehension database queries. In *IEEE Region 10 Conference (TENCON)*, Bombay, India, 1989.

[26] D Turner. Krc language manual. Technical report, 1981.

[27] Unknown. Haskell wiki: Stack overflow. Available at `http://www.haskell.org/haskellwiki/Stack_overflow`.

[28] Unknown. Wikipedia: LR parser. Available at `http://en.wikipedia.org/wiki/LR_parser`.

[29] Unknown. Wikipedia: Schwartian transform. Available at `http://en.wikipedia.org/wiki/Schwartzian_transform`.

[30] Phil Wadler. Comprehending monads. Available at `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`, 1990.

[31] Philip Wadler. The concatenate vanishes. Technical report, 1988.

[32] Philip Wadler and Simon Peyton Jones. Imperative functional programming. Available at `http://homepages.inf.ed.ac.uk/wadler/papers/imperative/imperative.ps`, 1993.

[33] Limsoon Wong. Kleisli, a functional query system. Available at `citeseer.ist.psu.edu/wong98kleisli.html`.

Appendix A

High Level Benchmarking Results

Program	Size Change	Allocations Change	Runtime Change	Compile Time Change
anna	+0.0	+0.0	0.08	+0.9
ansi	+0.0	+0.0	0.00	+0.0
atom	+0.0	+0.0	0.19	+2.2
awards	+0.0	+0.0	0.00	+3.3
banner	+0.0	+0.0	0.00	+0.0
bernouilli	+0.0	+0.0	-0.5	+2.5
boyer	+0.0	+0.0	0.03	-0.7
boyer2	+0.0	+0.0	0.01	+2.9
bspt	+0.0	+0.0	0.01	+1.6
cacheprof	+0.0	+0.0	+0.5	-0.3
calendar	+0.0	+0.0	0.00	+1.4
cichelli	+0.0	+0.0	0.08	+3.5
circsim	+0.0	+0.0	+0.0	-3.4
clausify	+0.0	+0.0	0.03	+0.0
comp_lab_zift	+0.0	+0.0	0.18	-0.9
compress	+0.0	+0.0	0.15	+2.4
compress2	+0.0	+0.0	0.13	+4.5
constraints	+0.0	+0.0	-0.6	+0.9
cryptarithm1	+0.0	+0.0	-1.3	+0.0
cryptarithm2	+0.0	+0.0	0.01	+2.1
cse	+0.0	+0.0	0.00	+1.1
eliza	+0.0	+0.0	0.00	+0.0
event	+0.0	+0.0	0.12	+1.0
exp3_8	+0.0	+0.0	0.11	+0.0
expert	+0.0	+0.0	0.00	+3.0
fem	+0.0	+0.0	0.02	+2.4
fft	+0.0	+0.0	0.02	+0.0
fft2	+0.0	+0.0	0.05	+0.9
fibheaps	+0.0	+0.0	0.02	-2.8
fish	+0.0	+0.0	0.01	-3.2
fluid	+0.0	+0.0	0.01	+0.9
fulsom	+0.0	+0.0	+0.0	+1.8
gamteb	+0.0	+0.0	0.11	+3.2
gcd	+0.0	+0.0	0.03	+0.0

gen_regexps	+0.0	+0.0	0.00	+2.8
genfft	+0.0	+0.0	0.02	+0.8
gg	+0.0	+0.0	0.01	+1.1
grep	+0.0	+0.0	0.00	-1.0
hidden	+0.0	+0.0	+0.0	+3.0
hpg	+0.0	+0.0	0.18	+1.0
ida	+0.0	+0.0	0.09	+0.8
infer	+0.0	+0.0	0.03	+3.6
integer	+0.0	+0.0	+0.1	+2.5
integrate	+0.0	+0.0	-0.0	+0.0
knights	+0.0	+0.0	0.00	+1.1
lcss	+0.0	+0.0	+0.0	+0.0
life	+0.0	+0.0	0.17	+2.0
lift	+0.0	+0.0	0.00	+3.3
listcompr	+0.0	+0.0	0.10	+0.8
listcopy	+0.0	+0.0	0.11	+0.8
maillist	+0.0	+0.0	0.05	+2.4
mandel	+0.0	+0.0	0.17	+3.7
mandel2	+0.0	+0.0	0.01	+2.0
minimax	+0.0	+0.0	0.00	+4.7
mkhprog	+0.0	+0.0	0.00	+0.0
multiplier	+0.0	+0.0	0.10	-1.3
nucleic2	+0.0	+0.0	+0.0	+0.7
para	+0.0	+0.0	+0.0	-0.5
paraffins	+0.0	+0.0	0.09	+1.1
parser	+0.0	+0.0	0.03	-1.3
parstof	+0.0	+0.0	0.00	-2.0
pic	+0.0	+0.0	0.01	+2.5
power	+0.0	+0.0	+0.7	+0.0
pretty	+0.0	+0.0	0.00	+5.1
primes	+0.0	+0.0	0.04	+0.0
primetest	+0.0	+0.0	+0.1	+4.1
prolog	+0.0	+0.0	0.00	+2.3
puzzle	+0.0	+0.0	0.16	+1.3
queens	+0.0	+0.0	0.02	+3.8
reptile	+0.0	+0.0	0.01	+1.0
rewrite	+0.0	+0.0	0.03	+2.5
rfib	+0.0	+0.0	0.06	+4.5
rsa	+0.0	+0.0	+0.4	+3.6
scc	+0.0	+0.0	0.00	+6.1
sched	+0.0	+0.0	0.02	+1.7
scs	+0.0	+0.0	+0.0	+0.3
simple	+0.0	+0.0	+0.0	-2.4
solid	+0.0	+0.0	0.09	+0.3
sorting	+0.0	+0.0	0.00	+1.5
sphere	+0.0	+0.0	0.11	-1.6
symalg	+0.0	+0.0	+0.0	+1.3
tak	+0.0	+0.0	0.01	+4.5
transform	+0.0	+0.0	-0.7	+0.0
treejoin	+0.0	+0.0	0.20	+2.5
typecheck	+0.0	+0.0	+0.0	+0.0
veritas	+0.0	+0.0	0.00	+0.4

wang	+0.0	+0.0	0.07	+1.9
wave4main	+0.0	+0.0	-0.7	+0.6
wheel-sieve1	+0.0	+0.0	+0.0	+0.0
wheel-sieve2	+0.0	+0.0	+0.0	+0.0
x2n1	+0.0	+0.0	0.03	+3.4
Min	+0.0	+0.0	-1.3	-3.4
Max	+0.0	+0.0	+0.7	+6.1
Geometric Mean	-0.0	-0.0	-0.1	+1.2

Appendix B

Investigating Library Fusion

In order to exploit the system of foldr/build fusion our library functions `groupWith` and `sortWith` should consume their input with `foldr` and produce their output with `build`.

However, it can be argued that any possible definition of `groupWith` in terms of `foldr` will suffer from stack overflow. The reason is that before the `foldr` worker can output any item of the result list, it must make sure that the current item does not belong in a group that starts in the remainder of the list. This means that the worker must always potentially be strict in its second argument (i.e. `worker` $x \perp = \perp$), triggering stack overflow for moderately sized lists [27]. A similar argument can be made for `sortWith`, suggesting that attempting to define either as `foldr`-based consumers is a lost cause.

However, nothing stops us from defining either as `build` based producers. This idea will be examined in the next two sections.

B.1 Fusable Grouping

A quantitative evaluation of three implementations of `groupWith` that make use of `build` was performed. The candidate functions were as follows:

```
-- In the following implementations the sorting the input has been excluded as it is a
-- common cost to all three functions and hence will not affect their relative performance
-- in the evaluation

-- This comman definition is used by all three implementations
span                    ::  (a -> Bool) -> [a] -> ([a],[a])
span _xs[]            =  (xs, xs)
span p xs(x:xss)
         |p x          =  let (ys,zs) = span p xss in (x:ys,zs)
         |otherwise    =  ([],xs)

-- This is the original definition which does not make use of build
{-# INLINE groupWith1 #-}
groupWith1 ::  Ord b => (a -> b) -> [a] -> [[a]]
groupWith1 f = groupBy (\x y -> f x == f y)

groupBy ::  (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []      =  []
groupBy eq (x:xs) =  (x:ys) :  groupBy eq zs
```

```
  where (ys, zs) = span (eq x) xs


-- This definition does make use of build in the most naive possible way
{-# INLINE groupWith2 #-}
groupWith2 ::  Ord b => (a -> b) -> [a] -> [[a]]
groupWith2 f xs = build (\c n -> groupByFB c n (\x y -> f x == f y) xs)

groupByFB2 ::  ([a] -> lst -> lst) -> lst -> (a -> a -> Bool) -> [a] -> lst
groupByFB2 _n _  []     =  n
groupByFB2 c n eq (x:xs) =  c (x:ys) (groupByFB2 c n eq zs)
  where (ys, zs) = span (eq x) xs


-- This version of the function makes use of build but attempts to reduce
-- its stack space usage over the previous definition
{-# INLINE groupWith3 #-}
groupWith3 ::  Ord b => (a -> b) -> [a] -> [[a]]
groupWith3 f xs = build (\c n -> groupByFB3 c n (\x y -> f x == f y) xs)

groupByFB3 ::  ([a] -> lst -> lst) -> lst -> (a -> a -> Bool) -> [a] -> lst
groupByFB3 c n eq xs = groupByFB3Core xs
  where groupByFB3Core [] = n
        groupByFB3Core (x:xs) = c (x:ys) (groupByFB3Core zs)
            where (ys, zs) = span (eq x) xs
```

I found that when the `foldr` consumer from `groupWith` was strict, fusion considerably degraded the runtime and worsened the stack characteristics of all the implementations. When the consumer was not strict (as is more usual, such as when the consumer is `map`), `groupWith3` had a 23% runtime and 4% memory consumption advantage over its competitors, that all performed roughly the same as each other. The `groupWith3` implementation was selected as the one to include in the codebase.

I also experimented with allowing fusion within `groupWith3` by using this definition of `groupByFB3`:

```
{-# INLINE groupByFB3 #-}
groupByFB3 ::  ([a] -> lst -> lst) -> lst -> (a -> a -> Bool) -> [a] -> lst
groupByFB3 c n eq xs =
    let groupByFB3Core [] = n
        groupByFB3Core (x:xs) = c (augment (\c n -> c x n) (takeWhile (eq x) xs))
                                  (groupByFB3Core (dropWhile (eq x) xs))
    in groupByFB3Core xs
```

Where `augment` is a standard function that forms part of the fusion system:

```
{-# INLINE [1] augment #-}
augment ::  forall a.  (forall b.  (a->b->b) -> b -> b) -> [a] -> [a]
augment g xs = g (:)  xs
```

This satisfies the additional term rewriting fusion rule:

$$\texttt{augment}\ g\ (\texttt{build}\ h) = \texttt{build (\textbackslash c n ->}\ g\ \texttt{c (}\ h\ \texttt{c n))}$$

Although this implementation of `groupByFB3` lead to a 21% reduction in memory usage for fusible scenarios, it slowed all users of the function down by 9%, even the non-fusible ones. I deemed this an unacceptable trade-off and so did not use this definition.

B.2 Fusable Sorting

I produced a modified version of GHC's standard merge sort implementation that injected the `build` "cons" function into just the last iteration of the merge sort. This meant that the output from the last stage could be consumed directly, via fusion, by any `foldr` consumer. A quantitative evaluation showed that although the new implementation was better, the efficiency gain over the simpler existing implementation was essentially nil (only 2% faster and requiring 1% less memory). I would speculate that this is due to the gains from not constructing the final list being very small compared to the irreducible cost of creating all the intermediate lists that are used during the merge sort.

The complexity of the fusible sort implementation comes from two directions: firstly, much code must be duplicated in order that it might deal with merging two lists to a fused result (for the last merge operation) as well as two lists to a list (for all other merge operations). Secondly, I had to write a series of rewrite rules to replace unfused instances of `mergeSort` with the original implementation. This is because the fusible implementation may be considerably worse than the original implementation when it is instantiated to just building a result list.

Due to the small resource consumption gains for the large increase in code complexity, I decided not to include this optimization in the library.

Appendix C

Haskell Tutorial

The following short tutorial has been constructed by blending together two freely available online tutorials. The first [1], from which the bulk of the material was taken, is in the public domain. The second tutorial [21], although much more comprehensive, has only been excerpted here due to its length. It is distributed subject to the condition that the following notice appear:

The intention is that the tutorial should contain just enough information on Haskell that you will be able to entirely understand the content of this dissertation, but no more.

C.1 Introduction

Haskell is a general purpose, purely functional programming language named after the logician Haskell B. Curry.

It's features include higher-order functions, non-strict (lazy) semantics, static polymorphic typing, user-defined algebraic datatypes, type-safe modules, stream and continuation I/O, lexical, recursive scoping, curried functions, pattern-matching, list comprehensions, extensible operators and a rich set of primitive data types.

C.2 The Structure Of Haskell Programs

A module defines a collection of values, datatypes, type synonyms, classes, etc. and *exports* some of these resources, making them available to other modules.

A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, and `main` must have type `IO ()`.

Modules may reference other modules via explicit `import` declarations, each giving the name of a module to be imported, specifying its entities to be imported, and optionally renaming some or all of them. Modules may be mutully recursive.

The name space for modules is flat, with each module being associated with a unique module name.

There are no mandatory type declarations, although Haskell programs often contain type declarations. The language is strongly typed. No delimiters (such as semicolons) are required at the end of definitions - the parsing algorithm makes intelligent use of layout. Note that the notation for function application is simply juxtaposition, as in `sq n`.

Single line comments are preceded by `--` and continue to the end of the line. For example:

```
succ n = n + 1  -- this is a successor function
```

Multiline and nested comments begin with `{-` and end with `-}`, thus:

```
{- this is a
      multiline
         comment -}
```

C.3 Lexical Issues

Haskell code will be written in typewriter font as in `f (x+y) (a-b)`. Case matters. Bound variables and type variables are denoted by identifiers beginning with a lowercase letter; types, constructors, modules, and classes are denoted by identifiers beginning with an uppercase letter.

Haskell provides two different methods for enclosing declaration lists. Declarations may be explicitly enclosed between braces `{ }` or by the layout of the code.

For example, instead of writing:

```
f a + f b where { a = 5; b = 4; f x = x + 1 }
```

One may write:

```
f a + f b = where a = 5; b = 4
                  f x = x + 1
```

Function application is curried, associates to the left, and always has higher precedence than infix operators. Thus `f x y + g a b` parses as `((f x) y) + ((g a) b)`

C.4 Values And Types

All computation is done via the evaluation of *expressions* (syntactic terms) to yield *values*. Values are divided into disjoint sets called *types* — integers, functions, lists, etc. Values are *first-class* objects. First-class values may be passed as arguments to functions, returned as results, placed in data structures, etc. Every value has a *type* (intuitively a type is a set of values). *Type expressions* are syntactic terms which denote *type values* (or just *types*). Types are not first-class in Haskell.

Expressions are syntactic terms that denote *values* and thus have an associated *type*.

C.4.1 Type System

Haskell is *strongly typed* - every expression has exactly one most general type, called the principle type.

Types may be *polymorphic*, i.e. they may contain *type variables* which are universally quantified over all types. Furthermore, it is always possible to statically infer this type. User supplied type declarations are *optional*.

C.4.2 Pre-defined Data Types

Haskell provides several pre-defined data types: Integer, Int, Float, Double, Bool, and Char.

Haskell provides for structuring of data through *tuples* and *lists*. Tuples have the form `(e_1, e_2, ..., e_n)` where $n >= 2$. If `e_i` has type `t_i` then the tuple has type `(t_1, t_2, ..., t_n)`

Lists have the form: `[e_1, e_2, ..., e_n]` where $n >= 0$ and every element `e_i` must have the same type, say t, and the type of the list is then `[t]`. The above list is equivalent to: `e_1:e_2:...:e_n:[]` that is, `:` is the infix operator for cons.

C.4.3 User-defined Data Types

User defined data types are introduced via a `data` declaration having the general form:

```
data T u_1 ... u_n = C_1 t_11 ... t_1k1
| ...
| C_n t_n1 ... t_nkn
```

Where `T` is a *type constructor*; the `u_i` are *type variables*; the `C_i` are *(data) constructors*; and the `t_ij` are the *constituent types* (possibly containing some `u_i`).

The presence of the `u_i` implies that the type is *polymorphic* - it may be instantiated by substituting specific types for the `u_i`.

Here are some examples:

```
data Bool = True | False
data Color = Red | Green | Blue | Indigo | Violet
data Point a = Pt a a
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

`Bool` and `Color` are *nullary type constructors* because they have no arguments. `True, False, Red,` etc are *nullary data constructors*. `Bool` and `Color` are *enumerations* because all of their data constructors are nullary. `Point` is a *product* or *tuple* type constructor because it has only one constructor; `Tree` is a *union* type; often called an algebraic data type.

C.4.4 Type Declarations

While Haskell does not require explicit type declarations (the type inference system provides static type checking), it is good programming practice to provide explicit type declarations. Type declarations are of the following form:

```
e :: t
```

Where *e* is an expression and *t* is a type. For example, the factorial functions type declaration is written like so:

```
fac :: Integer -> Integer
```

While the function `length` which returns the length of a list has a type of the form:

```
length :: [a] -> Integer
```

Not that `[a]` denotes a list whose elements may be any type.

C.5 Functions

Functions are first-class and therefore higher-order. They may be defined via declarations, or anonymously via *lambda abstractions*. For example:

```
\x -> x + 1
```

Is a lambda abstraction and is equivalent to the function `succ` defined by:

```
succ x = x + 1
```

If `x` has type `t_1` and `exp` has type `t_2` then `\x -> exp` has type `t_1 -> t_2`. Function definitions and lambda abstractions are curried, thus facilitating the use of higher-order functions. For example, given the definition

```
add x y = x + y
```

the function `succ` defined earlier might be redefined as:

```
succ = add 1
```

The curried form is useful in conjunction with the function `map` which applies a function to each member of a list. In this case:

```
map (add 1) [1, 2, 3] => [2,3,4]
```

The `map` function applies the curried function `add 1` to each member of the list `[1,2,3]` and returns the list `[2,3,4]`.

Functions are defined by using one or more equations. To illustrate the variety of forms that function definitions can take are are several definitions of the factorial function. The first definition is based on the traditional recursive definition.

```
fac n = if n == 0 then 1
        else n*fac( n - 1)
```

The second definition uses two equations and pattern matching of the arguments to define the factorial function.

```
fac 0     = 1
fac (n+1) = (n+1)*fac(n)
```

The next definition uses two equations, pattern matching of the arguments and uses the library function `product` which returns the product of the elements of a list. It is more efficient than the traditional recursive factorial function.

```
fac 0     = 1
fac (n+1) = product [1..(n+1)]
```

The final definition uses a more sophisticated pattern matching scheme and provides error handling.

```
fac n | n <  0 = error "input␣to␣fac␣is␣negative"
      | n == 0 = 1
      | n >  0 = product [1..n]
```

The infix operators are really just functions. For example, the list concatenation operator is defined as:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Since infix operators are just functions, they may be curried. Curried operators are called *sections*. For example, the first two functions add three and the third is used when passing the addition function as a parameter.

```
(3+)
(+3)
(+)
```

C.5.1 Block Structure

It is also permitted to introduce local definitions on the right hand side of a definition, by means of a where clause. Consider for example the following definition of a function for solving quadratic equations (it either fails or returns a list of one or two real roots):

```
quadsolve a b c | delta < 0  = error "complex␣roots"
                | delta == 0 = [-b/(2*a)]
                | delta > 0  = [-b/(2*a) + radix/(2*a),
                                -b/(2*a) - radix/(2*a)]
                  where
                  delta = b*b - 4*a*c
                  radix = sqrt delta
```

The first equation uses the builtin error function, which causes program termination and printing of the string as a diagnostic.

Where clauses may occur nested, to arbitrary depth, allowing Haskell programs to be organized with a nested block structure. Indentation of inner blocks is compulsory, as layout information is used by the parser.

C.5.2 Parametric Polymorphism

Functions and datatypes may be *polymorphic*; i.e., universally quantified in certain ways over all types. For example, the `Tree` datatype is polymorphic:

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Values of type `Tree Int` contain trees of fixnums; `Tree (Char -> Bool)` is the type of trees of functions mapping characters to Booleans, etc. Furthermore, in this definition:

```
fringe (Leaf x)            = [x]
fringe (Branch left right) = finge left ++ fringe right
```

The type of `fringe` is `Tree a -> [a]`, i.e. for all types `a`, `fringe` maps trees of `a` into lists of `a`.

Here:

```
id x = x
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
map f [] = []
map f (x:xs) = f x : map f xs
```

The function `id` has type `a->a`, `(++)` (append) has type: `[a]->[a]->[a]`, and `map` has type `(a->b)->[a]->[b]`. These types are inferred *automatically*, but may optionally be supplied as *type signatures*:

```
id   :: a -> a
(++) :: [a] -> [a] -> [a]
map  :: (a->b) -> [a] -> [b]
```

C.5.3 Type Synonyms

For convenience, Haskell provides a way to define *type synonyms* — i.e. names for commonly used types. Type synonyms are created using *type declarations.* Examples include:

```
type String = [Char]
type Person = (Name, Address)
type Name   = String
data Address = None | Addr String
```

This definition of `String` is part of Haskell, and in fact the literal syntax `"hello"` is shorthand for:

```
['h','e','l','l','o']
```

C.6 Pattern Matching

We have already seen examples of pattern-matching in functions (`fringe`, `++`, etc.); it is the primary way that elements of a datatype are distinguished.

Functions may be defined by giving several alternative equations, provided the formal parameters have different patterns. This provides another method of doing case analysis which is often more elegant than the use of guards.

Accessing the elements of a tuple is done by pattern matching. For example the selection functions on 2-tuples can be defined thus:

```
fst (a,b) = a
snd (a,b) = b
```

Here are some simple examples of functions defined by pattern matching on lists:

```
sum [] = 0
sum (a:x) = a + sum x

reverse [] = []
reverse (a:x) = reverse x ++ [a]
```

As-patterns are used to name a pattern for use on the right-hand side. For example, the function which duplicates the first element in a list might be written as:

```
f (x:xs) = x:x:xs
```

but using an *as-pattern* as follows:

```
f s@(x:xs) = x:s
```

Wild-cards will match anything and are used where we don't care what a certain part of the input is. For example:

```
head (x:_)  = x
tail (_:xs) = xs
```

C.6.1 Case Expressions

Pattern matching is specified in the Report in terms of case expressions. A function definition of the form:

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

is semantically equivalent to:

```
f x1 ... xk = case (x1, ..., xk) of (p11, ..., p1k) -> e1
                                    ...
                                    (pn1, ..., pnk) -> en
```

C.6.2 Lists

Lists are pervasive in Haskell and Haskell provides a powerful set of list operators. Lists may be appended by the ++ operator. Other useful operations on lists include the infix operator : which prefixes an element to the front of a list, and infix !! which does subscripting. Here are some examples:

```
["Mon","Tue","Wed","Thur","Fri"] ++ ["Sat","Sun"]
                  => ["Mon","Tue","Wed","Thur","Fri","Sat","Sun"]
0:[1,2,3]         => [0,1,2,3]
[0,1,2,3] !! 2 => 2
```

Note that lists are subscripted beginning with 0.

C.6.3 Arithmetic Sequences

There is a shorthand notation for lists whose elements form an arithmetic series.

```
[1..5]     => [1,2,3,4,5]
[1,3..10] => [1,3,5,7,9]
```

In the second list, the difference between the first two elements is used to compute the remaining elements in the series.

C.6.4 List Comprehensions

List comprehensions give a concise syntax for a rather general class of iterations over lists. The syntax is adapted from an analogous notation used in set theory (called set comprehension). A simple example of a list comprehension is:

```
[ n*n | n <- [1..100] ]
```

This is a list containing (in order) the squares of all the numbers from 1 to 100. The above expression would be read aloud as list of all n*n such that n is drawn from the list 1 to 100. Note that n is a local variable of the above expression. The variable-binding construct to the right of the bar is called a generator - the `<-` sign denotes that the variable introduced on its left ranges over all the elements of the list on its right. The general form of a list comprehension in Haskell is:

```
[ body | qualifiers ]
```

Where each qualifier is either a generator, of the form: `var <- exp`, or else a filter, which is a boolean expression used to restrict the ranges of the variables introduced by the generators. When two or more qualifiers are present they are separated by commas. An example of a list comprehension with two generators is given by the following definition of a function for returning a list of all the permutations of a given list:

```
perms [] = [[]]
perms x  = [ a:y | a <- x; y <- perms (x
[a]) ]
```

The use of a filter is shown by the following definition of a function which takes a number and returns a list of all its factors:

```
factors n = [ i | i <- [1..n]; n `mod` i = 0 ]
```

List comprehensions often allow remarkable conciseness of expression. We give two examples. Here is a Haskell statement of Hoare's Quicksort algorithm, as a method of sorting a list,

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = quicksort [ x | x <- xs, x <= p ]
                   ++ [ p ] ++
                   quicksort [ x | x <- xs, x >  p ]
```

Here is a Haskell solution to the eight queens problem. We have to place eight queens on chess board so that no queen gives check to any other. Since any solution must have exactly one queen in each column, a suitable representation for a board is a list of integers giving the row number of the queen in each successive column. In the following program the expression `queens n` returns all safe ways to place queens on the first n columns. A list of all solutions to the eight queens problem is therefore obtained by printing the value of `queens 8`:

```
queens 0 = [[]]
queens (n+1) = [ q:b | b <- queens n, q <- [0..7], safe q b ]
safe q b = and [ not checks q b i | i <- [0..(b-1)] ]
checks q b i = (q == b !! i) || (abs (q - b!!i) = i + 1)
```

C.6.5 Lazy Evaluation And Infinite Lists

Haskell's evaluation mechanism is lazy, in the sense that no subexpression is evaluated until its value is required. One consequence of this is that is possible to define functions which are non-strict (meaning that they are capable of returning an answer even if one of their arguments is undefined). For example we can define a conditional function as follows:

```
cond True x y = x
cond False x y = y
```

And then make use of it in such situations as `cond (x=0) 0 (1/x)`.

The other main consequence of lazy evaluation is that it makes it possible to write down definitions of infinite data structures. Here are some examples of Haskell definitions of infinite lists (note that there is a modified form of the `..` notation for endless arithmetic progressions):

```
nats = [0..]
odds = [1,3..]
ones = 1 : ones
nums_from n = n : nums_from (n+1)
squares = [ x**2 | x <- nums_from 0 ]
odd_squares xs = [ x**2 | x <- xs, odd x ]
cp xs ys = [ ( x, y ) | x <- xs, y <- ys ]        -- Cartesian Product
```

The elements of an infinite list are computed on demand, thus relieving the programmer of specifying consumer-producer control flow.

One interesting application of infinite lists is to act as lookup tables for caching the values of a function. For example here is a (naive) definition of a function for computing the nth Fibonacci number:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

This naive definition of fib can be improved from exponential to linear complexity by changing the recursion to use a lookup table, thus:

```
fib 0 = 1
fib 1 = 1
fib (n+2) = flist !! (n+1) + flist !! n
    where flist = map fib [ 0.. ]
```

C.6.6 Higher-Order Functions

Haskell is a fully higher order language — functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so `f x y` it is parsed as `(f x) y`, meaning that the result of applying f to x is a function, which is then applied to y.

In Haskell every function of two or more arguments is actually a higher order function. This permits partial parameterization. For example `member` is a library function such that `member x a` tests if the list `x` contains the element a (returning True or False as appropriate). By partially parameterizing member we can derive many useful predicates, such as:

```
vowel = member ['a','e','i','o','u']
digit = member ['0','1','2','3','4','5','6','7','8','9']
month = member ["Jan","Feb","Mar","Apr","May","Jun",
                "Jul","Aug","Sep","Oct","Nov","Dec"]
```

As another example of higher order programming consider the function `foldr`, defined by:

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

All the standard list processing functions can be obtained by partially parameterizing `foldr`. Here are some examples:

```
sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
          where postfix a x = x ++ [a]
```

C.6.7 Type Classes And Overloading

There is one final feature of Haskell's type system that sets it apart from other programming languages. The kind of polymorphism that we have talked about so far is commonly called *parametric* polymorphism. There is another kind called *ad hoc* polymorphism, better known as *overloading*. Here are some examples of *ad hoc* polymorphism:

- The literals `1`, `2`, etc. are often used to represent both fixed and arbitrary precision integers.
- Numeric operators such as `+` are often defined to work on many different kinds of numbers.
- The equality operator (`==` in Haskell) usually works on numbers and many other (but not all) types.

Note that these overloaded behaviors are different for each type (in fact the behavior is sometimes undefined, or error), whereas in parametric polymorphism the type truly does not matter (`fringe`, for example, really doesn't care what kind of elements are found in the leaves of a tree). In Haskell, *type classes* provide a structured way to control *ad hoc* polymorphism, or overloading.

Let's start with a simple, but important, example: equality. There are many types for which we would like equality defined, but some for which we would not. For example, comparing the equality of functions is generally considered computationally intractable, whereas we often want to compare two lists for equality. To highlight the issue, consider this definition of the function `elem` which tests for membership in a list:

```
x `elem` []     = False
x `elem` (y:ys) = x == y || (x `elem` ys)
```

We have chosen to define `elem` in infix form. `==` and `||` are the infix operators for equality and logical or, respectively. Intuitively speaking, the type of `elem` "ought" to be: `a -> [a] -> Bool`. But this would imply that `==` has type `a -> a -> Bool`, even though we just said that we don't expect `==` to be defined for all types.

Furthermore, as we have noted earlier, even if `==` were defined on all types, comparing two lists for equality is very different from comparing two integers. In this sense, we expect `==` to be *overloaded* to carry on these various tasks. *Type classes* conveniently solve both of these problems. They allow us to declare which types are *instances* of which class, and to provide definitions of the overloaded *operations* associated with a class. For example, let's define a type class containing an equality operator:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Here `Eq` is the name of the class being defined, and `==` is the single operation in the class. This declaration may be read "a type `a` is an instance of the class `Eq` if there is an (overloaded) operation `==`, of the appropriate type, defined on it." (Note that `==` is only defined on pairs of objects of the same type.) The constraint that a type `a` must be an instance of the class `Eq` is written `Eq a`. Thus `Eq a` is not a type expression, but rather it expresses a constraint on a type, and is called a *context*. Contexts are placed at the front of type expressions. For example, the effect of the above class declaration is to assign the following type to `==`:

```
(==) :: (Eq a) => a -> a -> Bool
```

This should be read, "For every type `a` that is an instance of the class `Eq`, `==` has type `a -> a -> Bool`". This is the type that would be used for `==` in the `elem` example, and indeed the constraint imposed by the context propagates to the principal type for `elem`:

```
elem :: (Eq a) => a -> [a] -> Bool
```

This is read, "For every type `a` that is an instance of the class `Eq`, `elem` has type `a -> [a] -> Bool`". This is just what we want - it expresses the fact that `elem` is not defined on all types, just those for which we know how to compare elements for equality.

So far so good. But how do we specify which types are instances of the class `Eq`, and the actual behavior of `==` on each of those types? This is done with an *instance declaration*. For example:

```
instance Eq Integer where
  x == y =  x `integerEq` y
```

The definition of `==` is called a *method.* The function `integerEq` happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side, just as for any other function definition. The overall declaration is essentially saying: "The type `Integer` is an instance of the class `Eq`, and here is the definition of the method corresponding to the operation `==`." Given this declaration, we can now compare fixed precision integers for equality using `==`. Similarly:

```
instance Eq Float where
  x == y =  x `floatEq` y
```

Allows us to compare floating point numbers using `==`. Recursive types such as `Tree` defined earlier can also be handled:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a         == Leaf b         = a == b
  (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) && (r1 == r2)
  _              == _              = False
```

Note the context `Eq a` in the first line—this is necessary because the elements in the leaves (of type `a`) are compared for equality in the second line. The additional constraint is essentially saying that we can compare trees of `a`'s for equality as long as we know how to compare `a`'s for equality. If the context were omitted from the instance declaration, a static type error would result. Haskell also supports a notion of *class extension.* For example, we may wish to define a class `Ord` which *inherits* all of the operations in `Eq`, but in addition has a set of comparison operations and minimum and maximum functions:

```
class (Eq a) => Ord a  where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

Note the context in the `class` declaration. We say that `Eq` is a *superclass* of `Ord` (conversely, `Ord` is a *subclass* of `Eq`), and any type which is an instance of `Ord` must also be an instance of `Eq`.

One benefit of such class inclusions is shorter contexts: a type expression for a function that uses operations from both the `Eq` and `Ord` classes can use the context `(Ord a)`, rather than `(Eq a, Ord a)`, since `Ord` "implies" `Eq`. More importantly, methods for subclass operations can assume the existence of methods for superclass operations. For example, the standard `Ord` declaration contains this default method for `(<)`:

```
  x < y = x <= y && x /= y
```

As an example of the use of `Ord`, the principal typing of `quicksort` is:

```
quicksort            ::   (Ord a) => [a] -> [a]
```

In other words, `quicksort` only operates on lists of values of ordered types. This typing for `quicksort` arises because of the use of the comparison operators `<` and `>=` in its definition. Haskell also permits *multiple inheritance*, since classes may have more than one superclass. For example, the declaration:

```
class (Eq a, Show a) => C a where ...
```

Creates a class `C` which inherits operations from both `Eq` and `Show`. Class methods are treated as top level declarations in Haskell. They share the same namespace as ordinary variables; a name cannot be used to denote both a class method and a variable or methods in different classes. Class methods may have additional class constraints on any type variable except the one defining the current class. For example, in this class:

```
class C a where
  m                    :: Show b => a -> b
```

the method `m` requires that type `b` is in class `Show`. However, the method `m` could not place any additional class constraints on type `a`. These would instead have to be part of the context in the class declaration.

Note that type applications are written in the same manner as function applications. The type `T a b` is parsed as `(T a) b`.

C.7 Haskell Functions Used In This Dissertation

```
-- 'take' @n@, applied to a list @xs@, returns the prefix of @xs@
-- of length @n@, or @xs@ itself if @n > 'length' xs@:
--
-- > take 5 "Hello World!" == "Hello"
-- > take 3 [1,2,3,4,5] == [1,2,3]
-- > take 3 [1,2] == [1,2]
-- > take 3 [] == []
-- > take (-1) [1,2] == []
-- > take 0 [1,2] == []
--
take :: Int -> [a] -> [a]


-- 'drop' @n xs@ returns the suffix of @xs@
-- after the first @n@ elements, or @[]@ if @n > 'length' xs@:
--
-- > drop 6 "Hello World!" == "World!"
-- > drop 3 [1,2,3,4,5] == [4,5]
-- > drop 3 [1,2] == []
-- > drop 3 [] == []
-- > drop (-1) [1,2] == [1,2]
-- > drop 0 [1,2] == [1,2]
--
drop :: Int -> [a] -> [a]


-- 'takeWhile', applied to a predicate @p@ and a list @xs@, returns the
-- longest prefix (possibly empty) of @xs@ of elements that satisfy @p@:
--
-- > takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
-- > takeWhile (< 9) [1,2,3] == [1,2,3]
-- > takeWhile (< 0) [1,2,3] == []
--
takeWhile :: (a -> Bool) -> [a] -> [a]


-- The 'genericLength' function is an overloaded version of 'length'.  In
-- particular, instead of returning an 'Int', it returns any type which is
-- an instance of 'Num'.  It is, however, less efficient than 'length'.
--
genericLength :: (Num i) => [b] -> i

-- The 'genericSplitAt' function is an overloaded version of 'splitAt', which
-- accepts any 'Integral' value as the position at which to split.
--
genericSplitAt :: (Integral i) => i -> [b] -> ([b],[b])


-- The 'nub' function removes duplicate elements from a list.
-- In particular, it keeps only the first occurrence of each element.
--
nub :: (Eq a) => [a] -> [a]
```

```
-- The 'partition' function takes a predicate, a list and returns
-- the pair of lists of elements which do and do not satisfy the
-- predicate, respectively; i.e.,
--
-- > partition p xs == (filter p xs, filter (not . p) xs)
--
partition :: (a -> Bool) -> [a] -> ([a],[a])


-- The 'sortBy' function is the non-overloaded version of 'sort'.
--
sortBy :: (a -> a -> Ordering) -> [a] -> [a]


-- Map a function over a list and concatenate the results.
--
concatMap :: (a -> [b]) -> [a] -> [b]


-- Application operator.  This operator is redundant, since ordinary
-- application @(f x)@ means the same as @(f '$' x)@. However, '$' has
-- low, right-associative binding precedence, so it sometimes allows
-- parentheses to be omitted; for example:
--
-- > f $ g $ h x = f (g (h x))
--
-- It is also useful in higher-order situations, such as:
--
-- > map ('$' 0) xs
-- > zipWith ($) fs xs
--
($)                             :: (a -> b) -> a -> b


-- The 'Functor' class is used for types that can be mapped over.
-- Instances of 'Functor' should satisfy the following laws:
--
-- > fmap id  ==  id
-- > fmap (f . g)  ==  fmap f . fmap g
--
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b


-- The 'Monad' class defines the basic operations over a /monad/,
-- a concept from a branch of mathematics known as /category theory/.
-- From the perspective of a Haskell programmer, however, it is best to
-- think of a monad as an /abstract datatype/ of actions.
-- Haskell's @do@ expressions provide a convenient syntax for writing
-- monadic expressions.
--
-- Instances of 'Monad' should satisfy the following laws:
--
-- > return a >>= k  ==  k a
-- > m >>= return  ==  m
-- > m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

```
--
-- Instances of both 'Monad' and 'Functor' should additionally satisfy the law:
--
-- > fmap f xs  ==  xs >>= return . f
--
class  Monad m  where
    -- Sequentially compose two actions, passing any value produced
    -- by the first as an argument to the second.
    (>>=)  :: forall a b. m a -> (a -> m b) -> m b

    -- Inject a value into the monadic type.
    return :: a -> m a
```

Appendix D

Project Proposal

Maximilian Bolingbroke
Robinson College
mb566

CST Part II Project Proposal

Adding SQL-Style List Comprehensions To The Glasgow Haskell Compiler

17/10/2007

Project Originator: Simon Peyton Jones

Resources Required: See attached Project Resource Form

Project Supervisor: Simon Peyton Jones

Signature:

Director Of Studies: Dr. Alastair Beresford

Signature:

Overseers: Dr. Neil Dodgson and Dr. Timothy Griffin

Signature:

Introduction and Description of the Work

The Glasgow Haskell Compiler[13] is the leading implementation of the lazy functional programming language Haskell[12]. The designers of Haskell provided for a simple so-called list comprehension syntax which can be used in place of the traditional list manipulation functions *map* and *filter*. The next two Haskell expressions to compute twice all the even numbers between 1 and 4 are equivalent, but one has been expressed with these functions and the other with a comprehension:

```
k1 = [x * 2 |x <- [1, 2, 3, 4], x ‘mod‘ 2 != 0]
k2 = map (\x -> x * 2) (filter (\x -> x ‘mod‘ 2 != 0) [1, 2, 3, 4])
```

As you can see, the list comprehension is rather easier to read than the equivalent list function based expression. Furthermore, list comprehensions are implemented by the compiler and so it is free to make choices about that implementation that the programmer may not have made for reasons of tedium or clarity. However, they are somewhat limited: we have essential designed a system which is capable of *selecting* and *projecting* from lists, but nothing else.

Recently a paper was published by Simon Peyton Jones and Phil Wadler[14] which provides a theoretical basis for two extensions to comprehensions in Haskell that give them strictly more expressiveness than SQL by adding *ordering* and *grouping* operators to the Haskell syntax and semantics which are slight generalizations of their SQL equivalents. By implementing these in the leading Haskell compiler, GHC, we could potentially provide an avenue for replacing much existing ad-hoc code operating over lists in this manner, which would have the benefits previously outlined of clarity and expressiveness to the compiler. An example of their use, drawn from the paper, is this code:

```
[ (the dept, sum salary)
|(name, dept, salary) <- employees
, group by dept
, order by Down (sum salary)
, order using take 5 ]
```

This returns a list of department and salary list pairs where each department occurs only once and the salary list is that of the top 5 earners in the corresponding department. Note the use of *order using* instead of what would be a LIMIT clause in traditional SQL: this hints at the generalization of ordering that was made in the paper.

Resources Required

No special resources are required.

Starting Point

I have largely reimplemented MinCaml using Haskell, which taught me much about compiler construction. Furthermore, in the last week I have read some of the source code of GHC and implemented a compiler pipeline stage which performs an experimental optimization transformation called “partial inlining” which is near completion.

GHC itself is a substantial compiler which includes many intricate optimization steps and exotic features. However, it does not currently have any support for the ordering or grouping syntax described above.

Substance and Structure of the Project

The project has the following main sections:

1. Implementation in the Glasgow Haskell Compiler of a form of the ordering syntax from the aforementioned list comprehension paper.
2. Evaluation of the effect of a naive implementation of this syntax on the performance of existing programs.

3. Ensuring that the new syntax has the desired semantics by a process of rigorous testing or another appropriate method.
4. Evaluation of the characteristics of different desugarings for the ordering construct within the compiler given the experience of implementing a simple variant.
5. Implementation of any improvements to the desugaring that can be determined and an exploration of the time, space and compile time characteristics of this resulting system.
6. Writing the dissertation.

The project will make use of several data structures already present in GHC, such as the representation of syntax as it passes through the compiler. Furthermore I will have to refine GHC to make use of a new data structure for list comprehensions. Currently, it internally represents them as a list of the so-called "qualifiers" that make up the comprehension. However, this is insufficiently expressive because in the generalization of comprehensions proposed by Simon Peyton Jones et al. it makes sense to allow the user to choose how to associate comprehensions since different associations correspond to different logical nested lists. This means that the comprehensions would be more naturally structured as a tree of qualifiers.

The most important algorithm in use by this project will be that which performs desugaring. The paper itself suggests two possible desugaring algorithms: a naturally recursive one based on nested tuples and a more refined one based on a defined notion of tuple addition. The choice of algorithm here will be affected by my study of the characteristics of the *foldr/build* fusion deforestation algorithm[8] in use by GHC and its competitor algorithms such as stream fusion[4].

Other data structures and algorithms will be used as appropriate where the implementation demands them.

Success Criterion

I will consider the project a success if it is possible to invoke GHC on this Haskell program:

```haskell
module Main where

people = ["Richard", "Jane", "Carol", "Simon", "Robert"]

list = [(x, y) |x <- people |y <- [5,4..1], (length x) > 4, order by ((head x), y)]

main = putStrLn (show list)
```

To produce an executable which when run outputs the following text:

```
[("Carol", 3), ("Robert", 1), ("Richard", 5), ("Simon", 2)]
```

Note that given my unfamiliarity with the GHC codebase, the success criteria requires only implementing the *order by* construction of the comprehensions paper. What's more, the order by construction only has to be supported at the *end* of the qualifier list of a list comprehension. This can be justified by considering that it is both the most frequent use case and the only one supported by SQL.

The project will further be evaluated on the basis of the space and time complexity of programs using the new mechanisms for list comprehensions versus list comprehensions compiled under the current system which supports only the restricted projection and selection operations. To this end GHC provides an extensive test suite, called "nofib", for evaluating the impact of compiler changes on typical Haskell programs. However, as these programs clearly make no use of the proposed additional capabilities for comprehensions a further set of tests will be developed by me as part of the project to gauge the relative performance of the new capabilities versus their classical implementations in Haskell 98.

Plan Of Work

Week 1 - Week 4 (Monday 22nd October 2007 - Friday 16th November 2007)

Study the use of *foldr/build* fusion in GHC and alternative fusion techniques. Begin to implement the new syntax required in GHC, ensuring it only works if a particular flag is set.

Milestones: Have read and understood the relevant stream fusion and *foldr*/*build* fusion papers. GHC is able to parse the new "order by" syntax, although an error occurs if the construct reaches the renamer.

Week 5 - Week 8 (Monday 19th November 2007 - Friday 14th December 2007)

Begin to consider alternatives for the desugaring of the order by construct. This will draw on both the underlying paper and any original ideas I might have. Concentrate on coming up with something naive that works, but understand the issues behind its performance. In parallel, do further work on GHC itself.

Milestones: Have a concrete plan for desugaring the order by construct. The GHC renamer and type checker handle the order by construct correctly, but compilation fails at the desugarer.

Week 9 - Week 12 (Monday 17th December 2007 - Friday 11th January 2008)

Implement the desugaring of the order by construction decided on in the last weeks. Ensure that the new qualifier syntax is handled correctly throughout the compiler paying e.g. in monads and pattern guards.

Milestones: The compiler can now correctly build programs using ordering over lists.

Week 13 - Week 16 (Monday 14th January 2008 - Friday 8th February 2008)

Prepare the progress report, clearly identifying the areas of further work that have been uncovered by my implementation. Test the performance of this new syntax versus explicit ordering and furthermore determine the effect of the new desugaring rules on existing list comprehension performance using both the "nofibs" benchmark suite and hand written test cases.

Milestones: The progress report is complete and handed in. The performance of the new system has been quantified and the effect of the new syntax on the possibilities for optimization is better understood.

Week 17 - Week 20 (Monday 11th February 2008 - Friday 7th March 2008)

Refine the desugaring in order to make the program outlined in the criteria for success not only work, but work as quickly as possible. Use the previously created testing regime to quantify the effects of these changes. Begin the main thrust of writing up the project into a dissertation.

Milestones: New desugaring is in place and the success criteria are fulfilled. Performance trade-offs are quantified. Introduction and preparation are complete, and excellent headway has been made into the implementation section of the writeup.

Week 21 - Week 24 (Monday 10th March 2008 - Friday 4th April 2008)

Dedicated time for writing up the dissertation based on the work of the previous weeks.

Milestones: Dissertation is essentially complete and in a fit state to be handed in.

Week 25 - Week 28 (Monday 7th April 2008 - Friday 2nd May 2008)

Overflow time for working on the dissertation should the unforeseen occur.

Week 29 - Week 30 (Monday 5th May 2008 - Friday 16th May 2008)

Finish any last minute remaining details in the dissertation and hand it in.

Milestones: The dissertation is handed in on time with complete introduction, preparation, implementation and evaluation sections.