Better Implicit Parallelism

Improving ApplicativeDo Heuristics Through Weight Annotations

MSc thesis in Computer Science: Algorithms, Language and Logic

Edvard Hübinette and Fredrik Thune

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

MSc thesis 2018

Better Implicit Parallelism

Improving ApplicativeDo Heuristics Through Weight Annotations

Edvard Hübinette and Fredrik Thune

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Better Implicit Parallelism

Improving ApplicativeDo Heuristics Through Weight Annotations

Edvard Hübinette and Fredrik Thune

Supervisor: Mary Sheeran, Computer Science and Engineering
Advisor: Simon Marlow, Facebook Inc.
Examiner: Carlo A. Furia, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Lines as parallel as our final programs.

Typeset in LaTeX
Gothenburg, Sweden 2018

Abstract

`ApplicativeDo` tries to remove the monad binds where possible when desugaring Haskells' do-notation. It does this by looking at inter-statement dependencies and connecting them with applicative operations when possible; this introduces implicit parallelism in the code. Deciding which structure to generate is tricky, since many different are usually possible. Currently, this is solved with simple minimum cost analysis with an unit cost model, assuming each statement takes the same amount of time to evaluate. By extending the cost model inside the `ApplicativeDo` algorithm for variable evaluation costs, we perform smarter generation of code by prioritising parallelisation of heavy statements.

At Facebook, Marlow et al. developed `Haxl`, a data-fetching library that can use applicative structure in expressions for free parallelism. To our knowledge, it is the only library that can do automatic parallelism on applicative expressions. We observe significant wall-clock speedups in many cases when running benchmarks with `Haxl`, compared to the original `ApplicativeDo`-desugared programs, given relative costs of statements.

The `ApplicativeDo` extension is agnostic to the source of costs, but one way of providing them is investigated as optional weight annotations on `do`-statements. This works well when the relative costs are known, but this kind of run-time estimation is notoriously difficult, particularly in Haskell. We suggest different directions for future work relating to sources of costs.

Keywords: Computer science, engineering, MSc thesis, functional programming, Haskell, inferred parallelism, Haxl, ApplicativeDo.

Acknowledgements

We want to extend our most sincere thanks to Mary Sheeran; she has been a wonderful supervisor with just the right mix of positivity and scepticism. Without her, we would have half as many examples and twice the work to do in the last weeks.

It has been a luxury to have Simon Marlow as an advisor; while being a very busy individual he still took the time to give us guidance and advice, both over email and personal meetings. We want to thank him for this and the splendid work on both `ApplicativeDo` and `Haxl`, as well as GHC and Haskell in general.

Thanks also to our examiner Carlo A. Furia for allotting us time, as well as asking interesting questions and motivating us at the half-time and final presentations.

We are also grateful for the support we have gotten from mPowered Solutions, with Emil Axelsson in particular. Looking at your production `Haxl` code was insightful and much appreciated, even though it did not happen to mesh well with our project.

Huge thanks to the GHC developers over at `#ghc@freenode.net`, they have helped us figure out a great deal of terribly confusing details of the compiler internals. In particular, Matthew Pickering gave us valuable insights on several occasions.

We also want to extend our gratitude to Matti Gíssurarson for helping us initially figure out the direction of this thesis, your experience and perspective gave us inspiration.

Thanks to Pierre Krafft for your interesting questions and comments in the role of our opponent. Invaluable feedback!

Edvard Hübinette and Fredrik Thune

Gothenburg, June 2018

Contents

1 Introduction

In a time when practically every machine has the potential to execute code in parallel, most programs do not fully lever this, letting machine power go to waste. Despite the fact that frameworks and techniques to simplify the process of writing parallel code exists, using them is still a demanding task in both time and resources.

It would be nice if we could abstract over parallelism and let the compiler do the heavy lifting for us. Imagine writing a legible program free from explicit evaluation order logic, with the compiler identifying points of possible parallelism and restructuring the code in a semantics-preserving way to make this opportunity for parallelism explicit. This restructuring could enable automatic execution tooling for parallelism to schedule the code for multi-core execution.

This idea may sound visionary, but is the same path we took with memory management; in some circumstances, we still need the manual control, but in most, the memory management provided by the compiler is more than satisfactory. In the same way, we would like our programs to be able to make use of available parallelism without the overhead of code for that sole purpose and to leave manual tinkering for situations with particularly high-performance demands.

This approach has an important consequence: since it is implemented in the compiler, we can empower *old* programs with retrofitted parallelism. This means that the advances of this programming language technology not only affect new programs but can also improve the performance of old code.

There have been successful advances in this field of implicit parallelism, but there is still a lot of room for further progression. For example, inside the Glasgow Haskell Compiler there are mechanisms for inferring parallelism in certain circumstances. We further develop this implementation, together with related functionality, to reach wall-clock performance speedups.

2

Background

This thesis is focused on certain internals of GHC [1], the Glasgow Haskell Compiler, a complex piece of software itself written in Haskell [2]. This chapter aims to explain important concepts from Haskell the language, elaborate on the parts of the compiler where our main contributions are focused and what its current shortcomings are, as well as showcase an execution library that can lever our improvements for better wall-clock performance, `Haxl`.

2.1 General knowledge of Haskell

This thesis dwells quite deeply into Haskell as a language. We try to assume as little knowledge as possible and explain terminology as it appears, but we do expect knowledge of the language corresponding to a basic course and some general insight into programming language technology and functional programming concepts.

For supporting literature we can recommend Miran Lipovaca's excellent book *Learn You a Haskell for Great Good!* [3], which is available freely on the web or in a physical edition. Another great resource is Graham Hutton's *Programming in Haskell* [4].

2.1.1 The `Functor`, `Applicative` and `Monad` abstractions

The `Functor`, `Applicative` and `Monad` abstractions are central to Haskell, and also this thesis. We dedicate this section to explaining them, how they relate, and in what way they represent certain properties that are important for our contributions. As a rule of thumb, less strict constraints on a function are good since it becomes more polymorphic. Therefore, a `Monad` constraint where the weaker `Applicative`, or more so `Functor`, would have been sufficient can be considered a code smell.

2.1.1.1 The `Functor` class

The Haskell `Functor` class is an abstraction for a data structure with a way of applying a function over its inhabitants, instantiated by implementing the function `fmap`. For example, a list is a `Functor` defined in such a way that applying a function over a list applies it to all of its elements. Below, we show the type class definition and instance for lists.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  f `fmap` []     = []
  f `fmap` (a:as) = f a : f `fmap` as

> fmap (+1) [1,2,3]
[2,3,4]
```

The type of `fmap` tells us that it takes a function of type `a -> b` and a data structure with values of type `a`, and transforms it into a data structure with values of type `b`. `fmap` is also available as the infix operator `<$>`

For the `Maybe` type, representing possible absence of values, applying a function to `Nothing` does no computation at all, while on a `Just a`, `a` is applied to the function.

```
instance Functor Maybe where
  f `fmap` Nothing  = Nothing
  f `fmap` (Just a) = Just $ f a
```

For the context of this thesis, `Functor` is important as it is a basic computation of lesser constraint compared to the stronger classes `Applicative` and `Monad`. This means that a function only needing the `Functor` constraint works over strictly more values which is good in a polymorphic sense.

2.1.1.2 The `Applicative` class

Since the first argument in `fmap` has to be a single, pure function that is *not* contained in the functor, `fmap` cannot be used for repeated application to functions with multiple arguments:

```
binFunc :: a -> a -> a
> binFunc `fmap` Just 1 :: Num a => Maybe (a -> a)
-- Cannot apply second argument using fmap
```

By applying the first argument to the binary function, the partially applied function will be returned in a `Maybe`, as shown in the example above. The next argument can therefore not be applied using `fmap`, since it is not an ordinary function type any more. To be able to apply more arguments, the stronger `Applicative` constraint is required, allowing us to do partial application with data structures. Let us look at the `Applicative` type class definition and the instance declaration for lists:

```haskell
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative [] where
  pure x = [x]
  fs <*> as = [f a | f <- fs, a <- as]
```

The function `pure` specifies how to bring something into the context of the data structure. For this particular case, this means putting it in a singleton list.

The application operator `<*>`, or *splat*, specifies what it means to apply an instance of the data structure with values, to an instance filled with functions. For lists, this means applying each of the values to each of the functions:

```haskell
> let fs = [(++ "1"), (++ "2")]
> let as = ["a", "b", "c"]
> fs <*> as
["a1","b1","c1","a2","b2","c2"]
```

For the `Maybe` type, the semantics are again different. If any step of the computation yields `Nothing`, subsequent computations will be short-cut to `Nothing`:

```haskell
instance Applicative Maybe where
 pure   = Just
 Nothing <*> _ = Nothing
 Just f <*> as = fmap f as
```

While possibly tricky to see from the types, this works well with functions of several arguments. If we give `b` the type `b -> c` of `<*>`, we get this:

```haskell
(<*>) :: f (a -> (b -> c)) -> f a -> f (b -> c)
```

If we then supply an argument to the functions in the first argument with `<*>`, we are ready for another one right away:

```
> (pure (+)) <*> Just 1 <*> Just 2
Just (1 +) <*> Just 2 -- Just 1 applied
Just 3 -- Just 2 applied
```

It is noteworthy that `a`, `b` and `f` are independent and could be evaluated in parallel.

2.1.1.3 The `Monad` class

The strength and weakness of the `Applicative` class is the independence of operations. This expresses parallelism, but cannot allow us to choose what operations to perform later in the computations depending on the value of the result from earlier.

Monads are best explained through examples, so we look at the class declaration for `Monad` and the instance for `Maybe`:

```
class (Applicative m) => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

The *bind* operator (`>>=`)allows us to connect two computations, just like `<*>`. However, the second argument is allowed to be dependent on the results from the first computation. This makes it possible to take a decision on what to do next depending on the value returned from earlier computations.

```
instance Monad Maybe where
  Nothing >>= _ = Nothing
  Just a  >>= Nothing = Nothing
  Just a  >>= f = f a
```

This looks deceivingly similar to the `Applicative` instance, but the difference lies hidden in the type of the continuation function: `f :: a -> m b`. With this, we can define more intricate computations working with intermediate results:

```
> let a = Just 3
> let sqr x = Just (x*x)

> a >>= \root -> sqr root >>= \square -> pure (root, square)
Just (1,2)
```

This syntax can quickly get complicated and hard to read; to make this easier Haskell supports `do`-notation. The same function can be written as:

```
> let a = Just 3
> let sqr x = Just (x*x)

do
  root   <- a
  square <- sqr root
  pure (root, square)
```

Every line in this notation is called a *statement*, and together they are referred to as a `do`-block. This notation is purely syntactic sugar, and in the compiler it is converted to using bind with simple rules:

```
f = do
  a <- something
  somethingElse a 5
  pure a

-- Is desugared into
f = something >>= \a ->
    somethingElse a 5 -> \_ ->
    pure a
```

The possibility of chaining of computations is a powerful property of monads and can result in nested monadic computations which can be flattened using the function `join`. The `join` function removes one layer of monadic structure, by merging the two outer layers:

```
join :: Monad m => m (m a) -> m a
join x =  x >>= id
```

With `join`, we can use functions that give results wrapped in a monad. Because `Maybe` is an instance of `Monad`, `join` can be used:

```
let f x = Just (x+1)
let b = Just 1

> pure f <*> b
Just (Just 2)
> join $ pure f <*> b
Just 2
```

A few monad laws need to be fulfilled to make sure monads behave as expected [5] These are not enforced by the compiler so the developer of a monad has that responsibility. These laws are:

- Left identity: `return a >>= f` $\equiv$ `f a`
- Right identity: `m >>= return` $\equiv$ `m`
- Associativity: `(m >>= f) >>= g` $\equiv$ `m >>= (\x -> f x >>= g)`

Since monads extend applicatives, for a monad to behave as expected when used as an applicative, these laws must also be true:

```haskell
pure = return
(<*>) = ap

ap :: (Monad m) => m (a -> b) -> m a -> m b
ap m1 m2 = do
    x1 <- m1
    x2 <- m2
    return (x1 x2)
```

This constraint is not always wanted and there can be good reason to break it, with the loss of the behaviour guarantees. One example could be performance gains in form of parallelism; this is the case with `Haxl`, which we will see in Section 2.4.

2.1.2 GHC language extensions

To enable compatibility between different Haskell compilers there is a language standard (e.g. Haskell98 or Haskell2010), keeping the language stable for longer periods of time. To enable the GHC development team to add new and interesting functionality between these updates, standard-breaking, non-backwards-compatible, experimental, or controversial features are activated using language extensions. These range from complex modifications to simple fixes and many are commonly used. They appear in this thesis, and some of the more common ones are elaborated upon in Appendix A.

2.2 Flexible desugaring of `do`-notation

As shown in the monad introduction, the `do`-notation is simply syntactic sugar for the monadic bind operator (`>>=`). This section explains recent developments of GHC that have generalised this translation into a more flexible system, allowing `do`-notation to be used without necessarily demanding a monadic, and hence sequential, constraint.

The paper *Desugaring Haskell's do-Notation into Applicative Operations* [6] describes extension of `do`-notation that liberates it from being exclusive to monads, to being available for applicatives as well. The functionality is implemented in GHC as a language extension called `ApplicativeDo`, which turns on an algorithm in the compiler with the same name. This makes it possible to take advantage of the less dependent relation between arguments of the applicative operators, thus allowing parallelism to be expressed inside `do`-notation.

While the standard translation of `do`-notation to monadic binds is straightforward, the generalised `do`-notation translates into a combination of monadic binds and the applicative operator `<*>`. There can be numerous valid translations and they are translated in different ways depending on how the variables in the block are used, making it a complex task. The goal of the algorithm is to analyse the decencies between statements and return an expression with as little forced sequentiality as possible. An example of the `ApplicativeDo` translation can be found in Listing 1; this is explained in further detail in Section 2.2.1.

How to find this optimal translation is a big part of the paper. If done successfully, few or no unnecessary dependencies are present in the desugared code, and libraries modifying execution order, such as `Haxl`[1] (described in Section 2.4), can make use of the independence in applicative expressions for running statements in parallel.

```haskell
main = do
    x1 <- a
    x2 <- b x1
    x3 <- c
    return (x2,x3)

-- Standard do-desugaring
main = a >>= \x1 ->
         b x1 >>= \x2 ->
           c >>= \x3 ->
             return (x2, x3)

-- Expected output from ApplicativeDo, with no forced sequentiality
-- between evaluation of the expressions (a >>= b) and c
main = (,) <$> (a >>= b) <*> c
```

Listing 1: An example of how a function containing a do-block with dependencies desugars. The standard desugaring result forces sequential execution through the `>>=` operator. The `ApplicativeDo` algorithm however only forces sequentiality between `a` and `b`.

[1] `https://github.com/facebook/Haxl`

2.2.1 The `ApplicativeDo` algorithm

The `ApplicativeDo` algorithm works in two stages: rearrangement and desugaring. Rearrangement collects statements in groups so that it enables as much parallelism as possible; it builds an intermediate tree structure of groups, where groups with dependencies between them are connected sequentially, and independent groups in parallel with each other. These groups does not re-order any statements; flattening the tree of groups will result in the original code.

The second stage *desugars* this intermediate tree structure into a proper Haskell expression using `<*>`, `<$>`,`>>=` and `join`.

2.2.1.1 Rearrangement

Consider the statements of a *do*-block, excluding the final statement which will be saved for the next stage. Each statement may or may not depend on any of the earlier statements in the block. The dependency structure of the example in Listing 1 is shown in Figure 2.1.

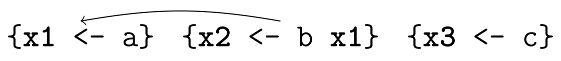


Figure 2.1: Dependency graph of the code in Listing 1, showing the dependencies between statements.

From a parallelism perspective, it would be ideal if all the statements could be segmented into groups without intermediate dependencies, meaning that they can be evaluated completely in parallel. Here however, there is a dependency from `{x2 <- b x1}` to `{x1 <- a}` which prevents that. `{x3 <- c}`, however, do not depend on anything; this statement can be evaluated independently of the other two. The `ApplicativeDo` algorithm looks for independence between statements, like between `{x3 <- c}` and `{x2 <- b x1}`. In general, if there is a splitting point in between statements which do not break any dependencies, the statements are split at that point into two groups. The split in the *do*-block shown in Listing 1 is shown in Listing 2.

```
    x1 <- a
    x2 <- b x1
----------------- Split
    x3 <- c
```

Listing 2: The only way to split the statements in the Listing 1 do-block without breaking any dependencies. `b` uses the result `x1` from computation `a`; this is a statement dependency.

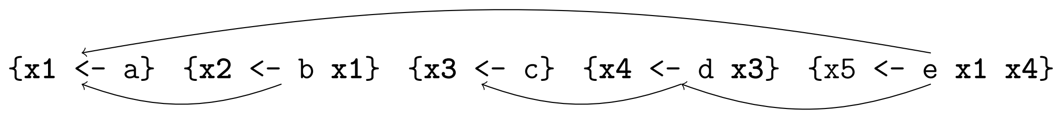
If it is possible to do a split without breaking any dependencies, as shown in the example above, the algorithm recursively rearrange each resulting group.

However, if it is not possible, the `do`-block has to be sequentially split. Let us revisit a practical example of how the `ApplicativeDo` algorithm works from the original paper. Consider this piece of code:

```haskell
do  x1 <- a
    x2 <- b x1
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4
    return (x2,x4,x5)
```

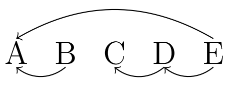
Listing 3: Example `do`-block from the *Desugaring Haskell's do-Notation into Applicative Operations* paper.

These statements are not all independent, since some use the results from others; this dependency structure is shown in Figure 2.2. Since this graph is a bit convoluted, a simplified version is shown in Figure 2.3. Here, the letters refer to the right side of each statement. This representation is used from here on.



```
{x1 <- a} {x2 <- b x1} {x3 <- c} {x4 <- d x3} {x5 <- e x1 x4}
```

Figure 2.2: Dependency structure of the code in Listing 3, showing the dependencies between statements.



$$\begin{tikzcd}
A & B \arrow[l, bend left] & C & D \arrow[l, bend left] & E \arrow[l, bend left] \arrow[llll, bend right]
\end{tikzcd}$$

Figure 2.3: Simplified dependency graph of the code snippet above, which shows the dependencies between statements written as letters referring to the right side of a statement. For example, the arrow from E to A shows that E uses the result of A; this of course means A has to be run before E.

The dependency from E to A prevents any parallel segmentation without breaking dependencies. Therefore the algorithm proceeds to split the sequence in two blocks, which will run sequentially to each other. Each of these parts is then recursively rearranged by checking for possible parallel segmentation and splitting if necessary. Trying all possible splits between statements and then segmenting gives us the following candidate solutions:

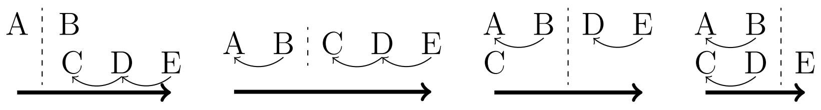


Figure 2.4: Execution graphs of candidate solutions after a single sequential split, shown as a dotted line. Segments on different lines are independent, so in the first, solution B can be run in parallel with C-D-E.

The `ApplicativeDo` algorithm will then determine which of these rearrangements that has the lowest total cost. Again, the algorithm assumes unit cost for all of the statements. Here a split after D (the rightmost rearrangement in Figure 2.4) seems to be the one with the lowest total cost of 3, by first evaluating A and B in parallel with C and D, then finally evaluating E, and is therefore chosen.

To find the optimal placement of these sequential splits, the algorithm simply tries all the possibilities and picks the best. The algorithm must know what *best* is, and for that we need a cost model. We denote each sequential enforcement with `;` and each independent split with `|`. Using this syntax, we can encode the rearranged code in Listing 2 and Listing 3 as:

```
({x1 <- a} ; {x2 <- b x1}) | {x3 <- c}
({x1 <- a} ; {x2 <- b x1}) | ({x3 <- c} | {x4 <- c x3}) ; {x5 <- e x1 x4 }
```

Listing 4: Encoded rearrangement of Listing 2 and Listing 3, with each sequential enforcement with `;` and each independent split with `|`.

From this syntax a simple cost model is defined:

```
cost(E) = 1
cost({x <- E}) = cost(E)
cost(A | B) = max(cost(A) , cost(B))
cost(A ; B) = cost(A) + cost(B)
```

Listing 5: Simple parallel unit cost model from the paper.

This cost model builds on the assumption that each statement has the same cost, which is of course a simplification, since in practice this is seldom true. The cost of groups running in parallel is defined as the cost of the most expensive group, while the cost of groups running in sequence is naturally defined as the combined cost of the groups.

An optimal or heuristic rearrangement can be used in the algorithm. The optimal will recursively test all possibilities, possibly giving a better solution at the expense of an increased compilation time. The heuristic version, which is the default, abandons the exhaustive search of arrangements and instead makes a local decision when faced with a group of statements that can't be split without breaking any dependencies. The algorithm then enforces sequentiality between the longest initial subsequence of mutually-independent statements and the rest of the statements, then continuing the same search for the next possible non-dependency breaking splitting point.

2.2.1.2 Desugaring

After rearrangement, we have a structure for what needs to be sequential and what can be parallel. This needs to be desugared into a proper, runnable Haskell expression.

Desugaring builds the expression from the rearranged statements, including the return statement, using the `<*>`, `<$>`,`>>=` and `join` functions. It does this by connecting parallel blocks using `<*>`, and sequential blocks with `>>=`. The simplified rules for applying these functions are seen in Listing 6.

```haskell
desugar ({b1} | {b2} | .. | {bn}) =
        f <$> desugar b1 <*> desugar b2 <*> .. <*> desugar bn
desugar ({b1} ; {b2} ; .. ; {bn}) =
        desugar b1 >>= \x1 -> desugar b2 >>=
        \x2 -> .. -> desugar bn >>= \xn -> f ..
```

Listing 6: The simplified rules for the desugar stage, which connects independent groups with `<*>` and dependent groups with `>>=`. Here, `f` is a function for the returning statement.

Looking at the example in Listing 7, the rearrangement will be `'({x1 <- a} | {x2 <- b}) ; ({x3 <- c x1} | {x4 <- d x2})'`. This is to be read as follows: `'{x1 <- a}'` and `'{x2 <- b}'` can be run in parallel, sequentially prior to, running `'{x3 <- c x1}'` and `'{x4 <- d x2}'` in parallel.

Combining this rearrangement with the desugaring rules will yield the code of `main1` in Listing 8. The first solution in Listing 8 is correct, but requires an unnecessary construction of a pair which has a minor performance overhead. To overcome this, the results of the first block (containing `a` and `b`) are supplied to the second (containing `c x1` and `d x2`) as arguments. This will result in a nested applicative computation, a computation returning another computation; this can be flattened using the `join` function.

As stated, the rules shown are simplified. The real algorithm is more intricate, considering more cases and using clever shortcuts to optimise the translation. The details on how the desugaring works are not important to understand since they merely simplify the generated expressionst. For more detailed information on this, *Desugaring Haskell's do-Notation into Applicative Operations* [6] is a worthy read.

```haskell
main = do
    x1 <- a
    x2 <- b
    x3 <- c x1
    x4 <- d x2
    return (x3,x4)
```

Listing 7: An example of a function containing a do-block with statement dependencies.

```haskell
main1 = ((,) <$> a <*> b)
        >>= \(x1,x2) ->
        (,) <$> c x1 <*> d x2

-- Nesting the applicatives to avoid the extra the pair constructor
main2 = join (
        (\x1 x2 ->
        (,) <$> c x1 <*> d x2)
        <$> a <*> b
       )
```

Listing 8: Two different ways to desugar the code in Listing 7. The first constructs an extra pair, `(,)`, which creates unnecessary overhead. The better solution uses nested computations and `join` to flatten the final answer, neatly avoiding this issue.

2.3 Possible improvements to ApplicativeDo

The paper *Desugaring Haskell's do-Notation into Applicative Operations* [6] introduces a basic unit cost model. This of course leads to the phrasing *optimal* in the algorithm being somewhat misleading. With better compile time information at hand, we could consider costs better representing the actual execution cost. The paper states:

> We can sometimes do better if we have more knowledge about the exact cost of statements, but in general that knowledge is not available.

It is from this statement we derive our research questions:

- Can we extend the `ApplicativeDo` algorithm to make better decisions, given information about statement evaluation cost?
- How can we provide the `ApplicativeDo` algorithm with this cost information with minimal programmer overhead?

2.3.1 Flaws under variable evaluation time

Here we show how the current cost model leads to imperfect decisions when considering variable statement execution time. To do this, let us revisit the example from Section 2.2.1, with its candidate solutions:

```haskell
do  x1 <- a
    x2 <- b x1
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4
    return (x2,x4,x5)
```

Listing 9: Example `do`-block from the *Desugaring Haskell's do-Notation into Applicative Operations* paper.

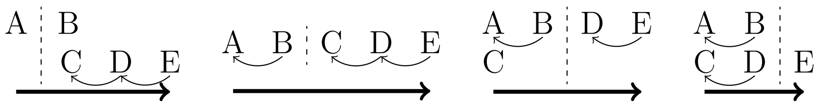


Figure 2.5: Execution graphs of candidate solutions for the above example.

This is the main issue: the current cost model assumes that each statement takes the same amount of time to evaluate. However, the statements B and E may, for example, in reality take four and two times as long as the other statements, respectively! If we assign these statements the corresponding costs, as shown in Figure 2.6, the execution graphs change. The right graph, which with unit cost was the optimal solution, now gives a total cost of seven, while the left graph now has a total of five. This means that when considering varying evaluation times, the *ApplicativeDo* algorithm is no longer optimal.

This shows that the unit cost model is not a good fit for deciding how to rearrange variable cost statements; a better model indicating more realistic evaluation costs would allow us to make smarter decisions in the algorithm. The information could for example come from optional programmer weight annotations, which is also something this thesis will investigate.

Now it should be clear how `ApplicativeDo` allows automatic inference of available parallelism by clever analysis of `do`-blocks. However, nothing automatically executes these programs in parallel; we need to use a library that can lever the implicit parallelism of applicatives in how it schedules code for execution. We hope to see further development in this exciting field, but currently `Haxl` is unique in this capability to our knowledge.

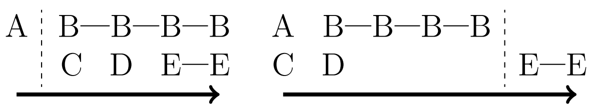


Figure 2.6: Execution graphs which show available parallelism for two different solutions, with B and E having real costs of 4 and 2 respectively. It it clear that the left graph is a better solution because it represents a more parallel evaluation.

2.4 The `Haxl` abstraction

The paper *There Is No Fork: an Abstraction for Efficient, Concurrent, and Concise Data Access* [7] describes a new programming abstraction using the implicit parallelism of the applicative operator `'<*>'`. This abstraction is implemented as the `Haxl` monad, and its primary target is to use implicit parallelism of applicative expressions, to make data fetching purely functional and concurrent.

The reason that `'<*>'` represents parallelism is that, as described in Section 2.1.1.2, its arguments cannot depend on each other. Without any dependencies between them, they could be performed in parallel. For example: in `'f <*> a <*> b'`, `'a'` and `'b'` cannot depend on each other.

`Haxl` is implemented as a monad and with it comes the `dataFetch` function, which is used to fetch data using IO:

```haskell
newtype GenHaxl u a

instance Functor (GenHaxl u) where ...
instance Applicative (GenHaxl u) where ...
instance Monad (GenHaxl u) where ...

dataFetch :: (DataSource u r, Request r a) => r a -> GenHaxl u a
```

Listing 10: `Haxl` monad

`Haxl` gains concurrency through the property of how its fetches are handled. Each time the evaluation is faced with a fetch, the request is noted and the evaluation continues elsewhere. Ideally, the fetches are taken care of simultaneously but it is up to the programmer to decide. Once all further evaluations are blocked waiting for a fetch, the evaluation progress halts and waits for the fetch result. The evaluation can continue when fetching completes. This property makes it so that multiple fetches used by non-dependent functions, such as `'<*>'`, can all be noted and performed independently from one another when needed. In the case of dependent fetches, such as when connected with a `'>>='`, the fetching needs to be done in sequence.

Since `Haxl` can make use of a combination of both dependent and non-dependent fetches, Haxl can translate non-dependencies in programs into parallelism. The real issue lies with the programmer needing to avoid unnecessary dependencies to make the most out of `Haxl`. Generally, building programs without unnecessary dependencies is hard and avoiding precision coding using `'<*>'` and `'»='` would thus be preferred. We would, therefore, like to be able to combine the more intuitive `do`-notation with `Haxl` without adding unnecessary dependencies and thus losing parallelism. `ApplicativeDo`, described in Section 2.2, does just that. The symbiosis of `Haxl` and `ApplicativeDo` makes it so that there is no need for explicit parallelism, but the extension infers it from dependencies in the `do`-blocks. This means that improvements to the `ApplicativeDo` algorithm will directly affect the runtime performance of `Haxl` programs written in `do`-notation, since it allows for more parallelism but never adds sequentiality.

2.4.1 Usage of `Haxl`

To use the `dataFetch` operations, requests have to be predefined and used by a data source. Consider the example in Listing 11, especially how requests are defined and then used by a data source.

The example shows a simplified program that fetches all values from a remote server. At first glance, the `do`-block on lines 11-14 looks entirely sequential, but beyond the non-dependent `<*>`, `Haxl` can also gain parallelism by other non-dependent functions such as `mapM`. Since `mapM` does not have any intermediate dependencies, each fetch can be taken care of separately from the evaluation of the rest of the code.

To complete the example above, the `DataSource` instance would need to be implemented, which at the current version is tailored to batch fetches. Batching is a crucial feature of how `Haxl` is used for data fetching; it utilises that multiple fetches can be sent over the network as a single request instead of individually. An in-development version of Haxl has a new mode which can prioritise parallelism over batching. This version has not yet been released at the time of writing but is described with examples in Section 3.2.2.3, and available at GitHub[2].

Even though `Haxl` is made for the specific purpose of data fetching, it can be used for a variety of tasks where implicit parallelism can be used, such as other remote or local data operations. In Section 3.2.2, we describe utilisation of `Haxl` in another context.

[2]Haxl 2.0 `https://github.com/facebook/Haxl/releases/tag/2.0.0.0`

```haskell
data MyRequest a where -- Define types of requests
  GetAllIds    :: MyRequest [Int]
  GetValueById :: Int -> MyRequest String

instance DataSource () MyRequest where    -- Implement the remote
    fetch state flags userEnv reqs = ... -- fetching procedure

main :: IO ()
main = do
 env <- initEnv initialState ()
 values <- runHaxl env $ do -- This do-block is a Haxl program
   x <- dataFetch GetAllIds
   y <- mapM (dataFetch . GetValueById) x
   return y
 putStrLn $ "Values" ++ show values


initialState :: StateStore
initialState = stateSet NoState stateEmpty

-- Required boilerplate instances below

deriving instance Show (MyRequest a)
deriving instance Eq   (MyRequest a)

instance StateKey MyRequest where
  data State MyRequest = NoState

instance Hashable (MyRequest a) where
  hashWithSalt salt _ = hashWithSalt salt ()

instance DataSourceName MyRequest where
  dataSourceName _ = "MyRequest"

instance ShowP MyRequest where
  showp _ = "MyRequest"
```

Listing 11: A simple `Haxl` example where ids are fetched from a remote data source followed by fetching those id's values.

3 Contributions

In this chapter we describe and document our contributions, together with the explanation and motivation of decisions. The sections can be summarised as follows:

- Section 3.1 describes extension of the `ApplicativeDo` algorithm with a more elaborate cost model.
- Section 3.2 describes an evaluation method of discerning whether our improvements to `ApplicativeDo` is better than the current implementation. What is a *better* parallelisation, and how can we measure improvements? We present both an analytic and empirical way of thought.
- Section 3.3 describes two ways of adding weight annotations. Section 3.3.1 describes a way of adding weight annotations with the use of changes in the Haskell lexer and parser; this is the final implementation. Section 3.3.2 describes the GHC annotation pragma, both the inner workings and our use for weight annotations, together with advantages and drawbacks. This is not part of the final implementation, but an interesting insight into relevant parts of the compiler nonetheless.
- Section 3.4 introduces Appendix E, documenting some of our experiences from developing on GHC: setting up a shared work environment, working with experimental patches not yet merged into the source control system, build configuration, et cetera. This is not central to the thesis results.
- Section 3.5 introduces Appendix D, describing the GHC compiler plugin system and our experience in working with it. Particularly, we show certain limitations, that in addition are not solved by a new proposal of extended plugin support for GHC.

This thesis also aims to be useful as a guide for future MSc thesis projects based on GHC; this is a particularly complex code base, and a lot of things are sparsely documented. Therefore, some exploration that is not used in the final result, but was difficult to figure out and can be useful for readers working in the field are also documented.

3.1 Extended cost model for `ApplicativeDo`

The `ApplicativeDo` algorithm is, as described in Section 2.2, implemented in two ways: an optimal and a heuristic version. The optimal version uses a simple cost model while the heuristic version uses none at all. The reason for that is that the cost model, described in Section 2.2.1, is used when choosing where to split a segment, through inserting a bind. The heuristic version never does this and instead takes the longest initial sub-sequence of statements with no intermediate dependencies, abandoning the cost model. We will, therefore, leave it as is to be used when low compile time overhead is so important that you can compromise runtime. On the other hand, the optimal version needs a way to compare each possible split to one another during the exhaustive search of options, with varying execution times.

As described in Section 2.3, weights would be beneficial to make more intelligent decisions when the `ApplicativeDo` algorithm desugars code. If auxiliary cost information is available for a statement it is used; otherwise, we default to a unit cost, as shown in Listing 12. This extension to the `ApplicativeDo` algorithm is completely agnostic to the *source* of weight information; later we will present how to get weights from optional programmer source annotations, but other paths are possible as well.

```
-- The cost of a statement is 1, or uses additional weight information
cost(E) = [auxiliary if available, else unit]
cost({x <- E}) = cost(E)

-- The model of calculating cost of parallel and sequential
-- operations is the same
cost(A | B) = max(cost(A) , cost(B))
cost(A ; B) = cost(A) + cost(B)
```

Listing 12: Parallel cost model for an optional auxiliary cost of statements.

Using this cost function, each expression in a do-statements is assigned a cost. Assuming we have a cost available, the extension of the `ApplicativeDo` algorithm is straightforward. Instead of the unit cost used in the original version of the algorithm, the cost is looked up using a function `getCurrentWeight`, as shown in Listing 13. Implementation details depend on the particular source of auxiliary costs, so now we just assume it returns a valid cost.

Costs in `ApplicativeDo` are only used in the split function, and the changes to the function are shown below. The task of the function `split` is, for a sequence $s_1 ... s_n$, to find the split that leads to the minimum sum of the left and right subsequences. It is not necessary to understand the function in detail.

The changes in the code are mainly a replacement of the encoding of a unit cost. The indices `hi` and `lo` is the start and end of an interval in the `do`-block. The function `getCurrentWeight` gives the weight/cost for a specific row, while `hiCost` and `loCost` are the costs for the start and end of the interval.

```haskell
-- Original code for creating statement trees with costs
split :: Int -> Int -> (ExprStmtTree, Cost)
split lo hi
      | hi == lo = (StmtTreeOne (stmt_arr ! lo), 1)
      | otherwise = (StmtTreeBind before after, c1+c2)
  where
  ((before,c1),(after,c2))
    | hi - lo == 1
    = ((StmtTreeOne (stmt_arr ! lo), 1),
       (StmtTreeOne (stmt_arr ! hi), 1))
    | left_cost < right_cost
    = ((left,left_cost), (StmtTreeOne (stmt_arr ! hi), 1))
    | left_cost > right_cost
    = ((StmtTreeOne (stmt_arr ! lo), 1), (right,right_cost))
    | otherwise = minimumBy (comparing cost) alternatives

-- Modified code for creating statement trees with weighted costs
split :: Int -> Int -> (ExprStmtTree, Cost)
split lo hi
      | hi == lo = (StmtTreeOne (stmt_arr ! lo), loCost)
      | otherwise = (StmtTreeBind before after, c1+c2)
  where
  ((before,c1),(after,c2))
    | hi - lo == 1
    = ((StmtTreeOne (stmt_arr ! lo), loCost),
     (StmtTreeOne (stmt_arr ! hi), hiCost))
    | cost0 == costn && left_cost < right_cost
    = ((left,left_cost), (StmtTreeOne (stmt_arr ! hi), hiCost))
    | cost0 == costn && left_cost > right_cost
    = ((StmtTreeOne (stmt_arr ! lo), loCost), (right,right_cost))
    | otherwise = minimumBy (comparing cost) alternatives
  loCost =  getCurrentWeight lo
  -- ^ Cost for statement on row 'lo'
  hiCost = getCurrentWeight hi
  -- ^ Cost for statement on row 'hi'
  cost0 = getCurrentWeight 0
  -- ^ Cost for statement on first row
  costn = getCurrentWeight n
  -- ^ Cost for statement on last row
```

Listing 13: Changes in the `ApplicativeDo` algorithm to make use of weights for each statement in a do-block.

3.1.1 Split optimisation shortcut

There is an noteworthy optimisation in the `ApplicativeDo` algorithm, located in the function `split`, which uses Theorem 4.3 from Marlow et al. [6]. The theorem says that if the statement subsequences $s_1 ... s_{n-1}$ and $s_2 ... s_n$ differ in cost, the optimal solution is the cheaper of the subsequences running sequentially to the remaining statement. Only when the subsequences have the same cost, an exhaustive search is needed to find the optimal rearrangement.

With the changes to the `ApplicativeDo` algorithm, the optimisation no longer functions as intended since it depends on the unit cost model. This simple counter example shows that the theorem do not hold for the changed context:

```
do
  x1 <- a  {-# Weight 3 #-}
  x2 <- b {-# Weight 1 #-}
  x3 <- c x1 x2 {-# Weight 1 #-}
```

Listing 14: Example where the optimal split is not the same as said in Theorem 4.3 in Marlow et al. [6]. Hence, the optimisation is not valid for the extended cost model.

Following the theorem, the subsequence $s_1 ... s_{n-1}$ will be `{x1 <- a}`, `{x2 <- b}`. Since it does not have any dependencies, it will have a cost of $Max(3, 1) = 3$. The other subsequence, `{x2 <- b}`, `{x3 <- c x1 x2}`, has a dependency and a cost of $1 + 1 = 2$. The theorem stated that the lowest of them will be optimal, which in this case is `{x2 <- b}`, `{x3 <- c x1 x2}`. Adding the final cost for `{x1 <- a}` the total cost is $2 + 3 = 5$. That is not optimal since a split after `{x2 <- b}` will result in a cost 4.

This optimisation can be adapted to the new cost model by verifying that the cost of the left and rightmost statements in a sequence are the same, then the same logic as presented in the paper applies. An adaptation to correct for this issue is shown in Listing 13, lines 27 and 29.

3.2 Evaluation method

To judge the effect of changes to the `ApplicativeDo` algorithm, we need a way to compare whether the resulting code after desugaring has better or worse characteristics. We approach this in two different ways: first through analytic evaluation to make sure the approach is sound, by evaluating and inspecting rearrangements of code by `ApplicativeDo`; second, through empirical experiments with `Haxl` running pure computations with different execution times in parallel, so not only remote data fetching operations.

3.2.1 Analytic evaluation

In `ApplicativeDo`, different rearrangements are often possible for a given dependency graph. Since each statement has a weight, annotated or defaulted to one, we can compute the total cost of the block using the parallel cost model previously shown. Depending on how the costs are distributed over the statements, the optimal rearrangement may differ.

Since the improved `ApplicativeDo` algorithm will give the rearrangement with the lowest cost, which in theory also is the optimal rearrangement. Finding the this rearrangement for a given program with weights is possible given all rearrangements and the cost model presented in Section 2.2.1. To derive all the rearrangements, the algorithm described in Section 2.2.1 can be used for a given `do`-block.

To evaluate the effect of our improvements, we compare the total cost of the rearrangement from the original algorithm and the changed algorithm, with the correct costs considered. If these costs correspond to the actual computational complexities, a lower total cost corresponds to a more parallel program. Section 4.3.1 presents the improvements achieved.

3.2.1.1 Evidence of optimality in improved algorithm

To ensure that the optimal rearrangement correspond to the result from the improved `ApplicativeDo`, we can inspect the code directly when it leaves the desugarer. Consider the rearrangements derived in Section 2.2.1:

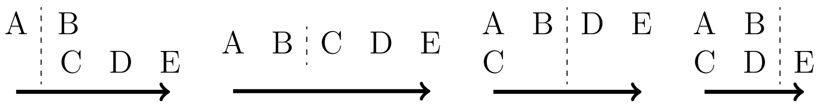


Figure 3.1: Execution graphs of candidate solutions after a single split, shown as a dotted line, with segmenting indicating available parallelism. Segments on different lines are independent, so in the first solution B can be run in parallel with C-D-E.

We want to verify if the algorithm gives the anticipated rearrangement. To do this we inspect the code directly when it leaves the desugarer. By passing the flag `-ddump-ds` to GHC, we get the fully desugared, mid-compilation code dumped from the compiler. It is not straightforward to evaluate the output by running it since it only contains implicit parallelism and no explicit threading logic. But we can do an analysis by looking at the dependency structure of the desugared code. Given this example `do`-block, we desugar and analyse the result:

```haskell
-- a, c, and d have equal evaluation cost.
-- b and e are 4 and 2 times as computationally heavy.
main = do
    x1 <- a
    x2 <- b x1    -- Known computational complexity factor of four
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4 -- Known computational complexity factor of two
    print (x2,x5)
```

Listing 15: Example do-block with statements of different computational complexity.

With the actual weights integrated with the new `ApplicativeDo` algorithm, the anticipated rearrangement is the leftmost in Figure 3.1. If no weights are used in `ApplicativeDo`, the anticipated rearrangement is the rightmost one.

We show the analytic method on the standard sequential desugaring of the `do`-block, before looking at the more complicated `ApplicativeDo` dumps. Without using `ApplicativeDo`, the desugaring will be a sequence of binds, which as we know are inherently sequential:

```haskell
main1 -- NORMAL, without ApplicativeDo
  = a >>=
        \x1 -> b x1 >>=
           \x2 -> c >>=
                \x3 -> d x3 >>=
                     \x4 -> e x1 x4 >>=
                          \x5 -> print (x2, x5)
```

Listing 16: Heavily simplified output from `-ddump-ds` of desugared code from Listing 15

We denote independent statements `(x | y)`, and a dependency with `(x ; y)`, meaning `y` is dependent on the result of `x`. The latter is what `(x >>= y)` expresses, so if we replace `>>=` with `;` in the snippet above, we get the piece of psuedocode shown in Listing 17. As expected, the dependency structure of the `do`-block without `ApplicativeDo` corresponds to a sequential chain of statements.

Now we want to look at the desugaring with `ApplicativeDo`, with and without weights, respectively. The expression `(x >>= y)` indicates a dependency, whereas `(x <$> y)` and `(x <*> y)` both have independent arguments. We can disregard the `join` function since it simply flattens the complete expression. Replacing these functions with appropriate dependency symbols, we can calculate a cost for the `do`-block with and without using weights in `ApplicativeDo`:

```haskell
main1 -- Vanilla desugaring, without ApplicativeDo
  = a ; \x1 -> b x1 ; \x2 -> c ; \x3 -> d x3 ;
    \x4 -> e x1 x4 ; \x5 -> print (x2, x5)
```

Listing 17: Dependency structure of the desugared code from Listing 15, without `ApplicativeDo`.

```haskell
-- Heavily simplified output from -ddump-ds of desugared code
main2
  = join ((fmap -- Partial function. x4 comes from <*>
            (\(x1, x2) x4 -> e x1 x4 >>= (\ x5 -> print (x2, x5)))
            ( a >>= \x1 -> fmap (\x2 -> (x1, x2)) (b x1))
          )
          <*> -- PARALLEL!
          ( c >>= \x3 -> fmap (\x4 -> x4) (d x3) )
      )

-- Creating dependency structure
main2
  = (
      (a ; \x1 -> b x1 ; \x2 -> (x1, x2))
      |
      (c ; \x3 -> d x3 ; \x4 -> x4)
    ) ; \(x1, x2) x4 -> e x1 x4 ; \x5 -> print (x2, x5)
```

Listing 18: Simplified intermediate dump, with dependency structure and without cost calculation from Listing 15. Using `ApplicativeDo`.

```
-- By passing the flag ddump-ds when compiling
main3
  = a >>=
        \x1 ->
         join ( (fmap (\x2 x5 -> print (x2, x5)) (b x1))
                <*> -- PARALLEL!
                (c >>=
                  (\x3 -> d x3 >>=
                    (\x4 -> fmap (\x5 -> x5) (e x1 x4))
                  )
                )
              )

-- Creating dependency structure
main3
  = a ; \x1 -> (
      (b x1 | c ; \x3 -> d x3 ; \x4 -> e x1 x4 ; \x5 -> x5)
      ;
      \x2 x5 -> print (x2, x5)
    )
```

Listing 19: Simplified intermediate dump, with dependency structure and cost calculation from Listing 15. Using `ApplicativeDo`.

Using `ApplicativeDo`, we get a desugaring with better implicit parallelism. We can clearly see that we get different solutions when using, or not using weights. By only looking at the `-ddump-ds` output, it is not trivial to distinguish which compilation corresponds to which rearrangement. However, at closer inspection we can see that Listing 18, which does not use weights, corresponds to the rightmost rearrangement in Figure 3.1 and Listing 19, which uses weights, corresponds to the leftmost. This is the expected output for the weights described in Listing 15.

Doing this for multiple examples of weights we get another verification that the optimal rearrangement is indeed returned from the improved `ApplicativeDo` algorithm. Worth noting is that this is with a "perfect" weighting, but we show that using weight information the algorithm *can* indeed perform better desugaring.

3.2.2 Evaluation using `Haxl`

`Haxl` is designed to run as many *data fetches* in parallel as possible. This is great in the setting it is designed for — systems with lots of external parts to query — which benefits greatly from batching requests to the same data source, and querying different sources in parallel.

We would like to use `Haxl` for parallelism in other contexts than just remote data fetching because it is, to our current knowledge, the *only* library that can automatically take advantage of implicit parallelism in applicative expressions. We can run any type of computation in parallel with `Haxl` by implementing them as data fetches; which we want both because it is both an interesting approach to automatic top-level parallelism and easier to use for evaluating the effects of modifications to the `ApplicativeDo` algorithm. This way of using `Haxl` is a bit contrived; hopefully we will see libraries for this purpose in the future.

3.2.2.1 Pure computation as data fetches

Arbitrary computation can be masqueraded as a data fetch in `Haxl`; this can be done by mocking a *data store*, where the implementation of the `fetch` function that is supposed to pull data remotely instead performs other arbitrary tasks. The tasks can be run in parallel with the use of a parallel execution library, such as the `Par` monad[1], and `Haxl` making sure only independent statements ever get run in parallel. An important note is that the relative order of these computations is arbitrary, but this is already a property of `Haxl`: the price we pay for implicit parallel execution is the loss of ordering other than what is demanded by statement dependency. For example, the relative order is unknown for the following `do`-block:

[1]`https://hackage.haskell.org/package/monad-par-0.3.4.8/docs/Control-Monad-Par.html`

```haskell
main = do
  x1 <- a
  x2 <- b
  return (x1,x2)
```

Listing 20: Example of a `do`-block in which the execution order of `a` and `b` are unknown in advance.

In the renaming phase, `ApplicativeDo` analyses the dependencies between statements in the computation, explores opportunities for applicative expressions for as much parallelism as possible. `Haxl` is hence not limited to only performing *data fetches* in parallel, but any computation *modelled* as a data fetch. A small example of how this can be done in practice can be found in Listing 21; this code is also available online.[2]

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE ApplicativeDo #-}

import Haxl.Core
import Data.Hashable (Hashable (..))
import Control.Monad.Par (runPar)
import Control.Monad.Par.Combinator (parMapM)
import Control.DeepSeq (rnf, NFData)
import System.IO.Unsafe (unsafePerformIO)
import Data.List (foldl')

data Heavy a where
  MockA :: Heavy Integer
  MockB :: Heavy Integer

-- Perform all requests to the data source in parallel using
-- the Par monad. Haxl will batch all independent requests to
-- this data source together in the variable reqs.
instance DataSource () Heavy where
  fetch _ _ _ reqs = SyncFetch $ runPar $ do
    parMapM (\(BlockedFetch req var) ->
      return $ runHeavy req var) reqs
    -- SyncFetch constructor requires a result of type IO (),
    -- but results are passed through an IORef in the runner
```

[2]Small Haxl example: `https://git.io/vNHTn`

```haskell
    pure (pure ())

-- By design of Haxl we don't care about the order our effects
-- occur, other than for the dependencies which the library
-- already does in batches. Therefore, it is not a problem to
-- perform our IO _now_, independently of the rest of the IO
-- operations.
instance NFData (IO ()) where
  rnf = unsafePerformIO


-- Defines the semantics of our two actual "data source"
-- operations.
runHeavy :: Heavy a -> ResultVar a -> IO ()
runHeavy MockA var = do
  let !n = foldl' (+) 0 [1..100000000]
  putSuccess var n
  putStrLn "MockA finished."
runHeavy MockB var = do
  let !n = foldl' (+) 0 [1..100000000]
  putSuccess var n
  putStrLn "MockB finished."

-- Initiates an (empty) state and runs a Haxl computation.
-- Thanks to ApplicativeDo, the two dataFetch calls desugars
-- to applicative operations because they have no dependency,
-- and are performed in parallel by the Haxl framework.
main :: IO ()
main = do
  env <- initEnv initialState ()
  summed <- runHaxl env $ do
    x <- dataFetch MockA
    y <- dataFetch MockB
    return $ x + y
  putStrLn $ "Result = " ++ show summed


initialState :: StateStore
initialState = stateSet NoState stateEmpty
```

Listing 21: Example of how `Haxl` data sources can be used to mock any computation. Some boilerplate instance declarations removed for brevity.

3.2.2.2 Limitations of `Haxl`

The `Haxl` library collects independent fetches into a single batch. It waits for the whole batch of fetches to finish before doing any more calculations, even though one of them might unlock more evaluation early. Sometimes this leads to unexpected run-time behaviour, as with the example in Listing 22.

```haskell
main = do
    x1 <- a
    x2 <- b x1   -- Weight 2
    x3 <- c      -- Weight 2
    return (x2,x3)

mainEx1 = a >>= x1 ->
    (,) <$> b x1 <*> c
mainEx2 = (,) <$> (a >>= b) <*> c
```

Listing 22: An example of a function containing a `do`-block with dependencies and different computational complexities. Two examples of how to desugar `main` with applicative syntax are shown as `mainEx1` and `mainEx2`, these are visualised in Figure 3.2 and Figure 3.3

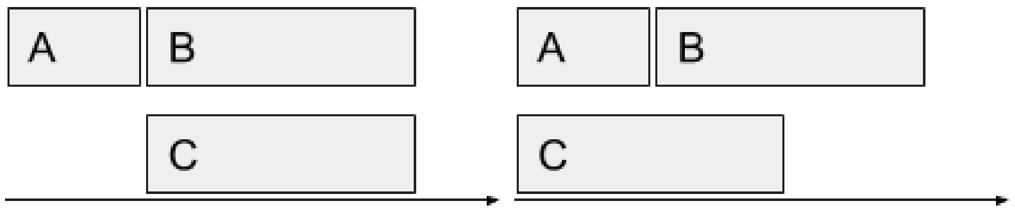


Figure 3.2: A visualisation of the the two ways to arrange the statements in Listing 22, with the left image being `mainEx1` and the right being `mainEx2`. Assuming weights as cost, they have the same original total cost 2, and the same weighted cost 3.

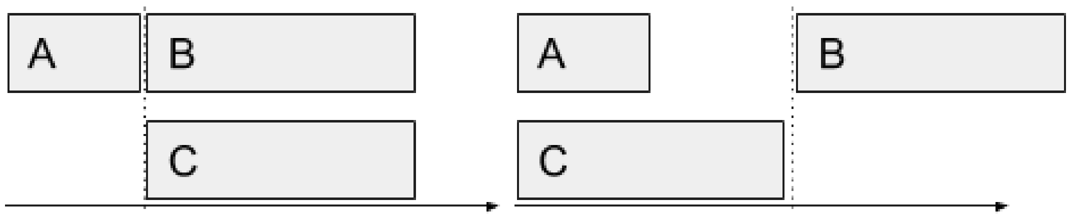


Figure 3.3: An execution visualisation of how `Haxl` runs the arrangements in Listing 22, with the left image being `mainEx1` and the right being `mainEx2`. In the right image, since Haxl waits for the batched call to finish, both `A` and `C` must finish before `B` can be fetched.

This simple example shows that two equally parallel desugarings of a `do`-block does not yield the same run-time behaviour when used by `Haxl`. `Haxl` collects as many blocking fetches as possible, sends them all on their way, and does nothing until they all return. Hence, one of the many fetches in a batch may be short and would unlock further evaluation but batching requires it to wait for the rest of the results before it can be evaluated.

The uncertainty of run-time behaviour will be present even for improvements for parallelism and is based on how the system manages batching of fetches. Unfortunately, this makes it impossible to improve on the performance in a reliable way from the perspective of `ApplicativeDo`. There is however a new version of `Haxl` under development that sidesteps this issue and allows prioritising parallelism over batching.

3.2.2.3 `Haxl` 2.0

The limitations presented in the previous section boils down to the fact that `Haxl` trades parallelism for sending fetches in batches. `Haxl` is extended in version 2.0 with an option to *fetch instantly* (not yet published at the time of writing, but is available at GitHub[3]). Instead of collecting fetches to prepare a batch, each will be performed as soon as possible, and its depending statements can begin evaluating as soon as the fetch returns, to fully utilise the available parallelism. Using the *fetch instantly* option will make `Haxl` execute the example code shown in Listing 22 as desired (rightmost illustration in Figure 3.2) instead of the batched version (rightmost in Figure 3.3).

The parallelism-improved version of `Haxl` makes `ApplicativeDo` algorithm improvements directly affect the execution time positively. It also makes reasoning about execution time much easier, since the cost model correlates stronger to execution time without the batching logic complicating things.

To activate the fetch instantly option, `SubmitImmediately` is passed as a scheduler hint in the `DataSource`, together with using `BackgroundFetch` instead of `SyncFetch` as shown in Listing 23. With only these changes, `Haxl` automatically executes some programs faster when the improved `ApplicativeDo` is used, as shown in Section 4.3.2.

```haskell
instance DataSource () Heavy where
  fetch _ _ _ = BackgroundFetch $ \(reqs :: [BlockedFetch Heavy]) ->
    mapM_ (\(BlockedFetch req var) ->
      forkIO $ runHeavy req var) reqs
  schedulerHint _ = SubmitImmediately
```

Listing 23: Changes to `DataSource` implementation to gain the full power of our improved implicit parallelism in `Haxl` 2.0.

[3]Haxl 2.0: `https://github.com/facebook/Haxl/releases`

3.3 Weights for `ApplicativeDo`

Greater knowledge of evaluation cost would be able to give a better cost function, and in return, performance could be improved. A simple way to gain this information is through optional programmer weight annotations. The option to annotate does require attention to believed evaluation time but does not require thinking about the evaluation order logic. A bit of conscious decision is already required for `ApplicativeDo` to perform optimally; as described in [6], the order of statements in a `do`-block will affect dependency structure, which in turn may affect the performance in the end. The `ApplicativeDo` algorithm can consider this additional weight information in the decision making, giving higher priority to parallelisation of heavily weighted statements.

3.3.1 Weights directly from the parser

This section describes the final implementation of adding programmer weight annotations to `do`-statements and getting them to the `ApplicativeDo` algorithm. Section 3.3.2 describes the initial implementation with the `ANN` pragma which works as a proof-of-concept, but also elaborates on some complications that makes it hard to work with in practice. This solution is arguably better for the programmer, but on the other hand requires more intrusive changes to Haskell, namely the lexical analysis and language grammar.

The main idea here is to modify the lexer and parser to enable adding dedicated weight pragmas to relevant types of statements in `do`-blocks, instead of hijacking the annotation pragma infrastructure. An example of how this looks in practice is shown in Listing 24. Further information on the practices of lexical analysis and parsing are out of scope in this thesis, but we recommend Aarne Ranta's excellent book *Implementing Programming Languages: An Introduction to Compilers and Interpreters* [8].

3.3.1.1 Lexer and parser modifications

First, we need to make the lexer aware of the fact that there are new tokens to consider when grouping the source code characters into chunks for the parser. These extensions are shown in code Listing 25.

We also need to extend the parser to get these tokens into the program structure, for later access down the compiler pipeline. The parser extensions needed for this are shown in code Listing 26.

In Listing 27, the parser state data is converted to a `Map` and put into the data type `HsParsedModule` in `HscMain.hscParse'`, which is the first internal representation of a parsed source file. `HsParsedModule` has a field for the type `ApiAnns` (used for other kinds of source annotation), which is extended to also keep the weights. `HsParsedModule` is then moved into the typechecking global environment (`TcGblEnv`) in `TcRnDriver.tcRnModuleTcRnM`, which makes for easy access in `ApplicativeDo`.

The main difference to the `ANN` pragma approach is that instead of having binder *names* to lookup for weights, source lines already have associated values of type `Maybe Weight`. The algorithm works with within-block statement indices, which point to an expression type with a source line reference, making lookups simpler than with the `ANN` pragma. Listing 28 shows how the weights are accessible in `rearrangeForApplicativeDom` ready to be used as described by the new cost model in Section 3.1.

```haskell
main = do
    x1 <- a         {-# Weight 4 #-}
    x2 <- b x1
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4 {-# Weight 2 #-}
    return (x2,x4,x5)

mainHaxl = do
  env <- initEnv initialState ()
  summed <- runHaxl env $ do
    x <- dataFetch MockA {-# Weight 2 #-}
    y <- dataFetch MockB
    return $ x + y
  putStrLn $ "Result = " ++ show summed
```

Listing 24: Using the weight annotations added in the language grammar. Can be compared to the same example with the `ANN` pragma approach, shown in Listing 29.

```
-- Weight data type, defined in compiler wide type
-- module BasicTypes.hs
+ data Weight = Weight Integer

-- Extension of the Token data type
data Token [...]
    | ITincoherent_prag   SourceText
+   | ITweight_prag       SourceText
    | [...]

-- Extension of the parser state to keep track of
-- collected weights associated with a source location
data PState = PState {
         [...]
         -- See note [Api annotations] in ApiAnnotation.hs
         annotations :: [(ApiAnnKey,[SrcSpan])],
         comment_q :: [Located AnnotationComment],
         annotations_comments :: [(SrcSpan,[Located AnnotationComment])],
+        weight_anns :: [(SrcSpan,Weight)]
      }

-- Initialisation of the parser state
mkPStatePure :: ParserFlags -> StringBuffer
             -> RealSrcLoc -> PState
mkPStatePure options buf loc =
       [...]
       annotations_comments = [],
+      weight_anns = []


-- Extension of the list of recognised one-word pragmas
oneWordPrags = Map.fromList [
      ("complete", strtoken (\s -> ITcomplete_prag (SourceText s))),
      ("column", columnPrag),
+     ("weight", strtoken (\s -> ITweight_prag (SourceText s)))
      ]
```

Listing 25: Extensions to the lexer module (`Lexer.x`) to accommodate tokens for the weight pragmas. Lines starting with a + mark our additions.

```
-- List of recognised tokens with corresponding
-- target data structure. 'L' is a constructor
-- for a located type, meaning it was an associated
-- source code position.
%token
    -- Haskell keywords
     '_'           { L _ ITunderscore }
    'case'         { L _ ITcase }
    [...]

    -- Haskell pragmas
    '{-# COMPLETE' { L _ (ITcomplete_prag _) }
+   '{-# WEIGHT'   { L _ (ITweight_prag _) }
    '#-}'          { L _ ITclose_prag
    [...]

-- Production for a do-statement. '$1' means
-- the first token, from 'qual' below.
stmt  :: { LStmt GhcPs (LHsExpr GhcPs) }
        : qual              { $1 }

        -- We do not consider recursive statements.
        | 'rec' stmtlist    { [...] }

-- Save weight statements for expression and binding
-- statements. Weights are produced by 'maybeweight',
-- and added to the parser state with 'aw'.
qual  :: { LStmt GhcPs (LHsExpr GhcPs) }
    -- sLL and mkBindStmt are not important and ams will be described below.
+   : bindpat '<-' exp maybeweight  {% (ams (sLL $1 $3 $ mkBindStmt $1 $3)
+                                          [mu AnnLarrow $2]) >>= aw $4 }

+   | exp maybeweight               {% aw $2 (sL1 $1 $ mkBodyStmt $1) }

    -- Do not consider 'let' binds because they are pure,
    -- and hence not part of the monad expression.
    | 'let' binds                 { [... ] }

-- Production for optional Weights.
maybeweight :: { Maybe Weight }
    -- Empty parse, return Nothing. Converted to
    -- a unit weight later.
    : {- empty -}                 { Nothing }

    -- Eat a 'Weight' pragma token, an associated
    -- integer literal, and save it in a Just.
    | '{-# WEIGHT' INTEGER '#-}'
        { Just (Weight $ il_value $ getINTEGER $2)

-- Getter for the source code associated with the weights.
-- This will be used for inferring weights of functions
-- defined elsewhere.
+getWEIGHT_PRAGs       (L _ (ITweight_prag       src)) = src


-- For reference, how other annotations are added to
-- the parser state
ams :: Located a -> [AddAnn] -> P (Located a)
ams a@(L l _) bs = addAnnsAt l bs >> return a

-- Add weights to the parser state, return the statement
-- untouched.
+aw :: Maybe Weight -> Located a -> P (Located a)
+aw Nothing a = return a
+aw (Just w) a@(L l _)  = addWeightAt l w >> return a

-- Add the weight to the parser state field.
+addWeightAt :: SrcSpan -> Weight -> P ()
+addWeightAt ss w = P $ \s -> POk s {
+    weight_anns = (ss,w) : weight_anns s
+  } ()
```

Listing 26: Extensions to the parser module (`Parser.y`) to parse the weight pragma tokens defined in the lexer. Lines starting with a + mark our additions.

```
-- In ApiAnnotation.hs, ApiAnns type is extended to
-- accomodate weights
type ApiAnns = ( Map.Map ApiAnnKey [SrcSpan]
               , Map.Map SrcSpan [Located AnnotationComment]
+              , Map.Map SrcSpan Weight )

-- In HscMain.hs, list of weights from parsing are
-- converted to a Map
case unP parseMod (mkPState dflags buf loc) of
    PFailed warnFn span err -> do [...]
    POk pst rdr_module -> do
        let res = HsParsedModule {
            hpm_module    = rdr_module,
            hpm_src_files = srcs2,
            hpm_annotations
              = (Map.fromListWith (++) $ annotations pst,
                 Map.fromList $ ((noSrcSpan,comment_q pst)
                                 :(annotations_comments pst)),
+                    Map.fromList $ weight_anns pst)
            }

-- In TcRnTypes.hs, TcGblEnv is extended to
-- accomodate weights
data TcGblEnv = TcGblEnv {
        [...],
+       tcg_ann_from_parser :: Map SrcSpan Weight
    }


-- In TcRnDriver.hs, the global typechecking env.
-- is updated with the weights from the parsed module
tcRnModuleTcRnM :: HscEnv
                -> HscSource
                -> HsParsedModule
                -> (Module, SrcSpan)
                -> TcRn TcGblEnv
tcRnModuleTcRnM hsc_env hsc_src
                (HsParsedModule {
                   hpm_module = [...],
                   hpm_src_files = src_files,
+                  hpm_annotations = (_,_,ssToWeights)
                })
                (this_mod, prel_imp_loc)
 = setSrcSpan loc $
   do { let { explicit_mod_hdr = isJust maybe_mod } ;
```

```
        tcg_env <- getGblEnv ;
        boot_info <- tcHiBootIface hsc_src this_mod ;
        setGblEnv (tcg_env { tcg_self_boot = boot_info ,
+                            tcg_ann_from_parser = ssToWeights }) $ do {
        [...]
```

Listing 27: Changes to `ApiAnnotation`, `HscMain`, `TcRnTypes`, and `TcRnDriver` to accommodate and move the weight annotations through the compiler.

3.3.2 Weights through `ANN` pragma

Here we describe the first working solution for weight annotations through the built-in `ANN` pragma[4]; it can be used in the way shown in Listing 29.

While the implementation works without doing any particularly intrusive changes to the compiler, the approach also has clear drawbacks. It is not possible to do annotations per statement, like with the parser approach, since the `ANN` pragma is syntactically limited to top-level binders, and not possible in nested expressions. This leads to the user having to do cumbersome refactoring, moving the content of statements to top-level functions just to annotate with a weight. Additionally, to annotate different calls to the same function requires it to be refactored to *two* functions with different annotations. It could also be argued that the implementation is a bit of a hostile takeover of a functionality meant for different things later in the compilation. The rest of this chapter will describe how this approach was implemented.

3.3.2.1 Implementation of the `ANN` pragma

To understand how we can use the pragma for our purpose, we take a look at the implementation of the `ANN` pragma in GHC. The GHC compilation pipeline is complex, and some information is only available in certain stages of the compiler; Appendix C contains a visualisation of this pipeline and may be a useful reference in this section.

An annotation is represented with the record type `Annotation`, containing fields for the name of the binder that is annotated and the *payload*, which is what we attach to the binder. In our case, this is the weight expression. The relevant data types are constructed as shown in Listing 30.

After the rename stage in the pipeline comes typecheck, where a global environment record gets populated with a lot of things, including the content of the pragmas. The environment, `TcGblEnv` (see Listing 31), is a big record where two fields are of particular relevance since they contain the information from the pragmas. The field `tcg_anns` is a list of all annotations, represented in the form shown in Listing 30.

The `tcg_ann_env` field contains a mapping from unique keys to a list of annotation payloads in the form of `UniqFM`, a particularly fast type of map for values of types implementing `Unique`. This is a type class making an uniquely identifying integer key available with `getUnique` even if we want to look up different types of keys. In practice this makes it possible to use a fast `IntMap`, even when we do not *really* want to look up integers.

[4]`https://ghc.readthedocs.io/en/master/extending_ghc.html#source-annotations`

```
rearrangeForApplicativeDo
  :: HsStmtContext Name
  -> [(ExprLStmt GhcRn, FreeVars)]
  -> RnM ([ExprLStmt GhcRn], FreeVars)
rearrangeForApplicativeDo _ [] = return ([], emptyNameSet)
rearrangeForApplicativeDo _ [(one,_)] = return ([one], emptyNameSet)
rearrangeForApplicativeDo ctxt stmts0 = do
+ hscAnns <- tcg_ann_from_parser <$> getGblEnv
  optimal_ado <- goptM Opt_OptimalApplicativeDo
+ let stmt_tree | optimal_ado = mkStmtTreeOptimal stmts hscAnns
                | otherwise = mkStmtTreeHeuristic stmts


-- The optimal algorithm now instead takes a weight map,
-- and getting the weight for a certain statement is
-- very simple.
mkStmtTreeOptimal :: [(ExprLStmt GhcRn, FreeVars)] ->
+                    Map.Map SrcSpan Weight ->
                     ExprStmtTree
mkStmtTreeOptimal stmts weightMap =
    [...]
    -- First arg is the stmt index, returns corresponding
    -- weight. If none annotated, default to unit weight.
    getCurrentWeight :: Int -> Int
    getCurrentWeight sIx = case stmts !! sIx of
         (L ss _, _) -> checkUnit $ Map.lookup ss weightMap
       where
         checkUnit :: Maybe Weight -> Int
         checkUnit Nothing = 1
         checkUnit (Just (Weight w)) = fromIntegral w
    [...]
```

Listing 28: The weights are now readily available inside the ApplicativeDo code, through the global environment.

```haskell
{-# ANN a (Weight 4) #-}
a = ...

{-# ANN e (Weight 2) #-}
e = ...

main = do
    x1 <- a
    x2 <- b x1
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4
    return (x2,x4,x5)

-- Can't just annotate dataFetch, since evaluation cost
-- can depend heavily on the argument.
{-# ANN dataFetchMockA (Weight 2) #-}
dataFetchMockA :: GenHaxl () Integer
dataFetchMockA = dataFetch MockA

mainHaxl = do
  env <- initEnv initialState ()
  summed <- runHaxl env $ do
    x <- dataFetchMockA
    y <- dataFetch MockB
    return $ x + y
  putStrLn $ "Result = " ++ show summed
```

Listing 29: Using the weight annotation system built with the `ANN` pragma. Forces quite annoying refactoring for the sake of annotation. Can be compared to the other approach, shown in Listing 24.

```haskell
data Annotation = Annotation {
  ann_target :: CoreAnnTarget
  ann_value  :: AnnPayload
}

type CoreAnnTarget = AnnTarget Name

data AnnTarget = NamedTarget name     -- General binder
               | ModuleTarget Module -- Not relevant, for annotating
                                     -- a whole module

-- Unique naming information
data Name = Name { ... }

-- Anything serialisable, i.e. writeable to disk
type AnnPayload = Serialized
```

Listing 30: Data types for representing content from `ANN` pragmas.

```haskell
data TcGblEnv = TcGblEnv {
  [...]
  tcg_anns    :: [Annotation]
  tcg_ann_env :: AnnEnv
  [...]
}

newtype AnnEnv = MkAnnEnv (UniqFM [AnnPayload])
```

Listing 31: The fields of the type checking environment tracking content of `ANN` pragmas.

The `tcg_ann_env` field is readily available for `ApplicativeDo`, but that resides in the rename part of the pipeline, and the annotations are not present when we need them since they are curated in the typecheck part of the compiler. The architecture of the compiler is very complex; moving the management of annotations to an earlier stage, if it is even possible, would be a major refactoring with a high probability of leading to cascading problems that need to be fixed. Hence we deem it as an unviable path. We can, however, add some specialised code that handles some of these annotations early, looking for weights added with the pragma to top-level declarations directly within `ApplicativeDo`. This is what we mean with the approach being a bit abusive of the existing GHC code, `ANN` pragmas are not meant to be used in this way in the current implementation.

3.3.2.2 Routing annotations to `ApplicativeDo`

The collection the annotations for the `ApplicativeDo` algorithm, explained in high-level, begins with a linear search of the top-level declarations before the renaming phase, placing the collected annotated binders and associated payloads in a data structure made available to the algorithm. We use the existing `TcGblEnv` and scrape the AST for relevant annotations ourselves.

Specifically, scraping for annotations is done in the function `tcRnModuleTcRnM`, in the `TcRnDriver` module, as displayed in (the heavily stripped and simplified) code listings below. `TcRnDriver` is a module residing in the Typecheck part of the pipeline, and `tcRnModuleTcRnM` is very close to the entry point of the type checker, which in an early phase applies the renamer. The functions `slurpTopLvlAnns` and `stripAndFilterAnn` traverses the top-level expressions of the AST, picks the data constructors apart and looks for annotation data. This is not as computationally heavy as it sounds since we do not recurse into expressions but only consider top-level binders.

First, we extend the global typechecker environment record, `TcGblEnv`, with a field for collected annotations:

```haskell
data TcGblEnv = TcGblEnv {
    [...],
    tcg_ann_from_parser :: [(AnnProvenance RdrName, HsExpr GhcPs)]
    }
```

Listing 32: Extension of the global typechecker environment type, `TcGblEnv`.

Then, in `tcRnModuleTcRnM`, we implement the scraping and run it before the environment is passed to the renamer:

```haskell
-- AnnProvenance is a representation of the binder name
-- HsExpr represents the annotation payload
type StrippedAnnD = (AnnProvenance RdrName, HsExpr GhcPs)
```

```haskell

-- | Traverses the top level declarations in the module, finds
-- annotations and returns the annotated binding together with
-- the payload expression.
slurpTopLvlAnn :: HsModule GhcPs -> [StrippedAnnD]
slurpTopLvlAnn hsModule = let lHsDecls = hsmodDecls hsModule in
  stripAndFilterAnn lHsDecls

-- | Strips location wrappers and collects content from ANN
-- declarations from the top level declarations; we do not
-- traverse deeper into the AST.
--
-- HsDecl has a constructor 'AnnD (AnnDecl name)', where
-- 'AnnDecl name = HsAnnotation (AnnProvenance name)'
--                              '(Located (HsExpr name))'
stripAndFilterAnn :: [Located (HsDecl GhcPs)]
                  -> [StrippedAnnD]
stripAndFilterAnn = foldr unwrapAnnD []
  where
    unwrapAnnD :: Located (HsDecl GhcPs) -> [StrippedAnnD]
               -> [StrippedAnnD]
    unwrapAnnD (L _ (AnnD (HsAnnotation _ annProv lHsExpr)))
                  annDecls = let L _ hsExpr = lHsExpr in
                    (annProv, hsExpr):annDecls
    unwrapAnnD _ annDecls = annDecls
    -- ^ Not interested in anything else, so we skip any other
    -- declaration

tcRnModuleTcRnM :: HscEnv -> HscSource -> HsParsedModule
                -> (Module, SrcSpan) -> TcRn TcGblEnv
tcRnModuleTcRnM hsc_env hsc_src
                (HsParsedModule {
                   hpm_module = L loc hmod,
                   hpm_src_files = src_files
                })
                (this_mod, prel_imp_loc)
 = setSrcSpan loc $
   do { [...]
        tcg_env <- getGblEnv ;
        setGblEnv (tcg_env {
          tcg_self_boot = boot_info ,
          tcg_ann_from_parser = slurpTopLvlAnn hmod
          }) $ do [...]
```

Listing 33: Scraping and storage of top level annotations for the typechecker environment.

Now that the annotations are available as a field in `TcGblEnv`, they are within reach from the main function of `ApplicativeDo`:

```haskell
1  rearrangeForApplicativeDo
2    :: HsStmtContext Name
3    -> [(ExprLStmt GhcRn, FreeVars)]
4    -> RnM ([ExprLStmt GhcRn], FreeVars)
5  rearrangeForApplicativeDo _ [] =
6    return ([], emptyNameSet)
7  rearrangeForApplicativeDo _ [(one,_)] =
8    return ([one], emptyNameSet)
9  rearrangeForApplicativeDo ctxt stmts0 = do
10   hscAnns <- tcg_ann_from_parser <$> getGblEnv
11   -- ^ Get the annotations from the environment
12   [...]
```

Listing 34: Easy access to scraped annotations in `ApplicativeDo` algorithm, collected from a `TcGblEnv` instance at line 10.

3.3.2.3 Looking up weights in `ApplicativeDo`

The content of the `ANN` pragmas needs to always refer to *renamed* binders; this is a result of a transformation taking place in Rename, a part of the compiler that resolves local binder names to fully qualified ones. This is needed because many of the optimisations in the simplifier need to know exactly which module binders point to. One example of where this knowledge required is cross-module and cross-package inlining, a process where GHC substitutes function calls with the implementing code, a central optimisation technique putting a dent in the overhead of function calls in tight loops.

There are different kinds of names in the compiler, each with their own purpose.

- `OccName` is the simplest form of a name, and simply represent names as a string and from which namespace that it came. Namespaces could be for example value, type constructors or data constructors.
- `RdrName` comes directly from the parser and contains a `OccName`, together with an optional module qualifier. It is not yet processed by the renamer and could therefore be ambiguous with other `RdrNames`.
- `Name` is unique and have both scope and binding resolved. They also contain a `OccName`, which are not unique.

The data structure of weights and names in the `ApplicativeDo` algorithm is a simple list of pairs of function name together with individual weight expression. The list contains values of type `RdrName`, meaning the names are not unique, but as the list only contains top-level annotations from within the same module, this is not a problem since multiple definitions are anyway illegal. For faster lookups, the list is converted to a `Map` with the `OccName` from the `RdrName` as keys, and weight expressions as values.

The function that uses the weights in `ApplicativeDo` is `mkStmtTreeOptimal`; it needs to be able to lookup possible weight information for the statements it treats. Since the `ANN` pragmas are only used for top-level annotation, a statement in a `do`-block cannot be annotated. Therefore we only consider the top-most identifier on the right side of a bind or in a bind-less statement, called *body statement*. An identifier could either be a function application or a variable. The function `getCurrentName` looks up the name of the statement currently handled:

```haskell
-- Gets the name of the current statement
getCurrentName :: Int -> Maybe Name
getCurrentName stmIndex = getStmNameMaybe (stmt_arr ! stmIndex)
 where
  getStmNameMaybe :: (ExprLStmt GhcRn, FreeVars) -> Maybe Name
  getStmNameMaybe (L _ (stmtLR), _) = case stmtLR of
      -- stmtLR :: StmtLR GhcRn GhcRn (LHsExpr GhcRn)
      -- data StmtLR is defined in HsExpr.hs
      (BodyStmt (L _ expr) _ _ _)   -> getExpNameMaybe expr
      (BindStmt _ (L _ expr) _ _ _) -> getExpNameMaybe expr
      _ -> Nothing
  getExpNameMaybe :: HsExpr GhcRn -> Maybe Name
  getExpNameMaybe (HsApp (L _ expr) _) = getExpNameMaybe expr
  getExpNameMaybe (HsVar (L _ name))   = Just name
  getExpNameMaybe _ = Nothing
```

Listing 35: Finding a top-level name for a statement in a do-block

Since both `Name` and `RdrName` contains an `OccName` field, their `OccName`s are suitable for a comparison between the two data types. Therefore, a weight map can be constructed with the key in the form of an `OccName` with the data of the weight of an expression. After checking that it is, in fact, a valid weight annotation, the annotation payload is converted into an integer value which can be used by the `ApplicativeDo` algorithm. The implementation of how these weights are accessed for a certain statement is shown in Listing 36, and can be compared to the ease of use in the other approach (shown in Listing 28).

```haskell
-- The map with weights from the annotations is in scope as
-- weightMap :: Map (AnnProvenance RdrName) (HsExpr HcsPs)

getCurrentWeight :: Int -> Maybe Integer
getCurrentWeight i = join $ getWeightFromWeightExpr <$> getWeightExpr i
  where
    -- Does a lookup in the weight-map with the name,
    -- converted as a OccName, of the current statement
    getWeightExpr :: Int -> Maybe (HsExpr GhcPs)
    getWeightExpr i = join $ (flip Map.lookup weightMap . nameOccName)
                        <$> getCurrentName i
    -- Converts a weight expression into a weight integer value
    getWeightFromWeightExpr :: HsExpr GhcPs -> Maybe Integer
    getWeightFromWeightExpr (HsPar (L _ exp)) = getWeightFromWeightExpr exp
    -- ^ Remove parentheses
    getWeightFromWeightExpr (HsApp (L _ varExp) (L _ valExp))
      -- Extracts the ANN data type and check that it is a Weight
      | maybe False isValidWeightRdrName $ extractRdrName varExp
                     = getWeightFromWeightExpr valExp
                     -- ^ Get the integer value
      -- If the ANN data type is something else nothing is returned
      | otherwise    = Nothing
    getWeightFromWeightExpr (HsOverLit overLit) =
      case ol_val overLit of -- Convert to integer
        HsIntegral i -> Just $ il_value i
        _            -> Nothing
    getWeightFromWeightExpr _   = Nothing

    extractRdrName :: HsExpr GhcPs -> Maybe RdrName
    extractRdrName (HsPar (L _ e))     = extractRdrName e
    -- ^ Remove parentheses
    extractRdrName (HsVar (L _ name)) = Just name
    extractRdrName _                  = Nothing
```

Listing 36: Finding the binder name for the right side of a statement found in a do-block.

3.4 Hacking on GHC

The Glasgow Haskell Compiler is a complex beast and setting up a development environment can be a bit tricky. In Appendix E we document a well working setup for upstream tracking source control, as well as setting up a properly optimised build environment locally. This was not entirely straightforward and the information was deemed useful by us for future theses focusing on GHC.

3.5 Compiler plugins

The GHC plugin system at first seemed particularly suitable for collecting data from source annotations. Collecting data from source annotations via a GHC plugin would have been a nice way to avoid having to make many changes to the compiler. Unfortunately, the plugin system is too constraining, even with the big extension proposal[5] that is currently under consideration.

This led to the conclusion that refactoring out things to a plugin, while still having to do GHC changes specific to that same plugin, is simply not viable. Hence, the implementation was moved inside the compiler. To document these apparent shortcomings we elaborate on this approach in Appendix D, even though the final solution does not make use of the plugin system. The appendix is useful as a working example of how the extended plugin system can be used since there are no other examples or tutorials to be found at the time of writing (and are sparse even for the vanilla plugin system).

The issues were raised with the GHC developers for insight to the development of the proposal at GitHub[6], and taken up for further discussion on Phabricator[7].

[5]`https://ghc.haskell.org/trac/ghc/wiki/ExtendedPluginsProposal`

[6]`https://github.com/ghc-proposals/ghc-proposals/pull/107#issuecomment-380806941`

[7]`https://phabricator.haskell.org/D4342#128964`

4 Results

This thesis documents improvements to a certain model of implicit parallelism inference in GHC. A typical use of the system is shown in a working context in Section 4.2, followed by evaluation and performance tests in Section 4.3. The following is a summary of our contributions:

- An extension of the `ApplicativeDo` algorithm to work with non-unit statement costs in `do`-blocks, for more efficient parallelisation of monadic code.
- Two possible solutions on how to get costs specified by optional weight annotations in the source code to the relevant place in the compiler, each with their own advantages and drawbacks.
- A description of how to use `Haxl` for general parallel computation instead of only remote data fetching. This shows our improvements are applicable to more than just remote data fetching.
- Documentation of research practicalities on GHC, for the sake of future theses based on the compiler.

4.1 Contribution results

This section elaborates on the actual contributions of this thesis. The accompanying code can be found at GitHub [1]. Our commits are interleaved with merged upstream changes from the GHC developer team, but a branch comparison shows our changes by themselves [2]. There is also a Docker container available for easy experimentation[3], with a pre-compiled GHC sporting our changes and a pre-configured Cabal package skeleton. In the root directory of the container there is a `README.txt` explaining how to work with the package.

[1]Project code repository: `https://github.com/m0ar/ghc/tree/weighted-do-stmts`

[2]Summarized GHC changes: `https://github.com/tweag/ghc/compare/5b63240f9822507a33bca6c5c05462832a9f13ab...m0ar:weighted-do-stmts`

[3]Project container: `https://hub.docker.com/r/thune/ghc-weighted-do-stmts/`

4.1.1 `ApplicativeDo` improvements

Our `ApplicativeDo` algorithm with an extended cost model successfully generates expressions with more implicit parallelism, when accurate cost information is supplied. Considering representative costs of statements, the algorithm makes better decisions and returns programs with more tactical dependencies that prioritise placing heavy computations independent of each other instead of in sequence, where possible. This increased implicit parallelism leads to faster execution of some `Haxl` programs, as shown in Section 4.3.2.

Our changes to the compiler never (with the small caveat explained in Section 6.1) affect the program performance negatively, nor does it have any measurable impact on compile time compared to the original `ApplicativeDo` with the compilation flag `-foptimal-applicative-do`.

Particularly noteworthy is that our implementation is a general improvement, and agnostic to the *source* of costs; they could come from any source, but we have evaluated the path of optional programmer annotations.

4.1.2 Costs from programmer annotations

Two different approaches of getting cost information from source code annotations to the `ApplicativeDo` algorithm have been presented. The solution utilising the `ANN` pragma is least intrusive to Haskell, but is slightly abusive of a GHC feature that is actually used for other things later in the pipeline. It also allows the addition of weight annotations where they have no effect, and refactoring of statements out to top-level is required.

The other approach, which is based on the parser, is less hacky, but requires changes to the language grammar and lexical analysis. On the other hand, it allows a smoother implementation of the rest of the functionality. Additionally, weight annotations are only syntactically correct on relevant `do`-statements and can be placed there directly without any refactoring.

Neither can infer the weight from a function call; this is a limitation and a direction for further work.

4.1.3 Parallelisation with Haxl

We show how to use `Haxl` for getting automatic parallel execution of other things than data fetching; this style of coding is a bit contrived, but shows that our improvements are not only applicable to remote data fetching, but also for scheduling of top-level parallelism for other types of computation as well.

The currently published version of `Haxl` does not allow the user to prioritise parallelism over batching behaviour. Improvements in implicit parallelism are therefore not always used by `Haxl`; in some cases it can even worsen performance. However, the in-development version makes prioritisation of parallelism possible, allowing our improvements to shine.

4.1.4 Documentation of GHC research practicalities

As stated earlier, this thesis is also meant to be useful as a guide for future MSc theses based on GHC, since the official documentation on internals can be a bit confusing and scattered. The appendix therefore includes documentation of:

- How to set up and work with collaborative GHC Git forks.
- Details on the Phabricator development tools and experimental branches, in combination with external git hosting.
- The use of compiler plugins, including the largely undocumented novel plugin system.
- The build system, with possible configurations and useful tricks.
- A mixed bag of details on the compiler pipeline.

4.2 System demo

Here we will look at a practical example of how the system is used. Consider this `Haxl` program; it does not perform as well as expected and some of the operations are known or suspected to be more computationally heavy than others.

```
do  x1 <- a
    x2 <- b x1
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4
    return (x2,x4,x5)
```

Listing 37: Example `do`-block from Marlow et al. [6].

Without having to specify the execution order manually, annotations can be added to the heavy computations to give the compiler more information to work with for how to parallelise the expression, as shown in Listing 38. This can yield a differently structured expression from `ApplicativeDo`, prioritising on isolating heavy

```haskell
do  x1 <- a
    x2 <- b x1    {-# Weight 4 #-}
    x3 <- c
    x4 <- d x3
    x5 <- e x1 x4 {-# Weight 2 #-}
    return (x2,x4,x5)
```

Listing 38: Example `do`-block from Marlow et al. [6], with added weight annotations.

statements from others with applicative operations, opening up for more efficient parallel runtime evaluation. How this affects the code generated within the compiler is shown in Section 4.3.1, together with performance improvements using `Haxl` in Section 4.3.2. It is possible to get *worse* performance by using non-representative weights, but ease of use encourages experimentation.

The running time of this particular example is 4.56 seconds without `ApplicativeDo`, improved to 3.67 seconds with the original implementation (24% speedup), and again improved to 2.75 seconds with correct costs. This is an improvement of 66% from not using `ApplicativeDo` at all, and of 33% compared to the original `ApplicativeDo`.

4.3 Evaluation

In this section, we show the results for analytic and empirical evaluations, as described in Section 3.2. The analytic is a theoretical evaluation where the algorithm is performed manually and the total costs for different weight configuration are scrutinised. The empirical evaluations consist of running programs using `Haxl` over independent runs, each with different weight configuration of statements, comparing the new and old `ApplicativeDo` algorithms.

4.3.1 Analytic evaluation

As described in Section 3.2, given a `do`-block, numerous valid rearrangements can be derived without breaking any dependencies. The method for deriving the rearrangements was described in Section 2.2.1, in which an example of a weight distribution is used to find an optimal rearrangement. In that specific example we can see a distinct improvement factor, but this is not always the case; therefore, an analytic analysis to get a better understanding is performed.

For a given program, each statement is given either weight 1, 2 or 4 and all combinations are tested exhaustively. To see how much improvement is gained compared to the original algorithm, we compare the optimal rearrangement for the weighed case to the rearrangement from the original `ApplicativeDo`. The weights correspond to actual run-time, this is because we evaluate the improvements to `ApplicativeDo`, not the weight annotation method which may can indeed be inaccurate. As stated in Section 3.1, the improved `ApplicativeDo` is agnostic to the *source* of costs, and programmer annotations is just one way to supply them.

To find which of the rearrangements the original `ApplicativeDo` returns, we compare the order in which each statement is running to the rearrangements. This process is time-consuming, and hence we do this analysis for two test programs only.

4.3.1.1 Program one

This program is the `do`-block from Marlow et al. [6], which we have used throughout this thesis.

```
    do  x1 <- a
        x2 <- b x1
        x3 <- c
        x4 <- d x3
        x5 <- e x1 x4
        return (x2,x4,x5)
```

Listing 39: Example `do`-block from Marlow et al. [6].

The `do`-block in Listing 39 gives the candidate solutions in Figure 4.1. These four rearrangements are the only valid derivable solutions of the program in Listing 39. Further splitting will give equal or less parallelism and is therefore not shown. The solution selected by the original `ApplicativeDo` algorithm is the rightmost figure. By exhaustively testing the different weight configurations we get the improvement factor distribution that is shown in Figure 4.2.

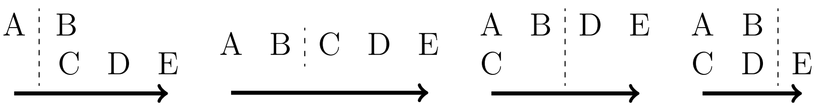


Figure 4.1: Execution graphs of candidate solutions after a single split, shown as a dotted line, with columns indicating available parallelism. Segments on different lines are independent, so in the first, solution B can be run in parallel with C-D-E.

In 11% of the cases we can see an improvement over the original `ApplicativeDo`, and in the remaining 89% we keep the original rearrangement. Hence, performance is never worse given that the weights correspond to execution time.

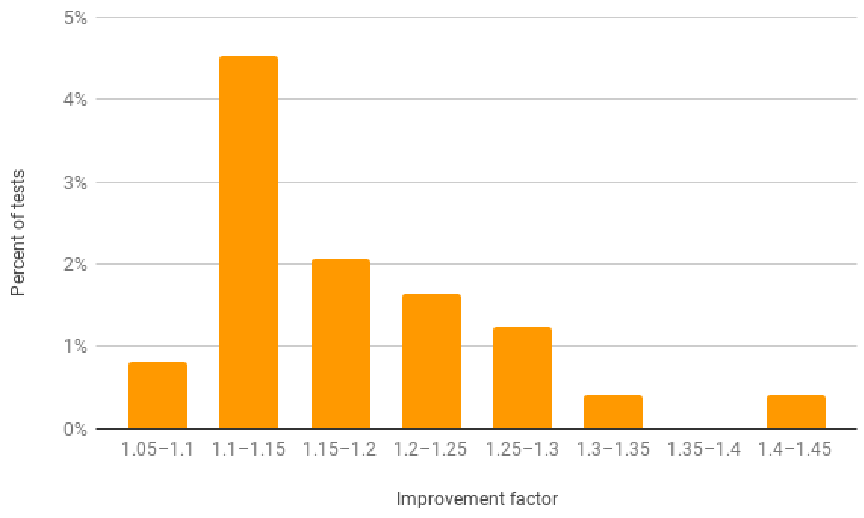


Figure 4.2: In 11% of the cases, our algorithm outperformed `ApplicativeDo`. This is the distribution of the improvements.

4.3.1.2 Program two

This program, shown in Listing 40, is slightly bigger and has a more complex dependency structure than the first.

```
    do  x1 <- a
        x2 <- d x1
        x3 <- b x1
        x4 <- f x2
        x5 <- e x1 x3
        x6 <- g x1 x3 x5
        return (x4,x6)
```

Listing 40: Example `do`-block for analytic evaluation.

The five rearrangements in Figure 4.3 are the only valid derivable solutions of the program in Listing 40. Further splitting will give less parallelism and are therefore not shown. The rearrangement returned by the original `ApplicativeDo` algorithm is the rightmost solution. By exhaustively testing the different weight configurations we get the improvement factor distribution shown in Figure 4.4.

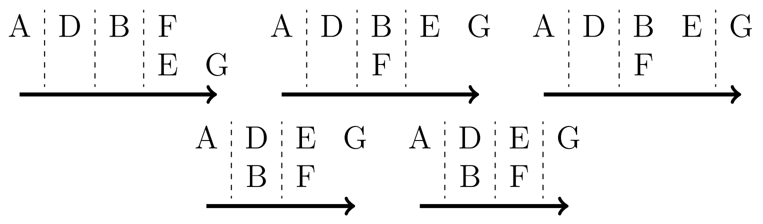


Figure 4.3: Execution graphs of candidate solutions after multiple splits, shown as a dotted lines, with columns indicating available parallelism. Segments on different lines are independent, so in the top left execution graph, solution F can be run in parallel with E-G.

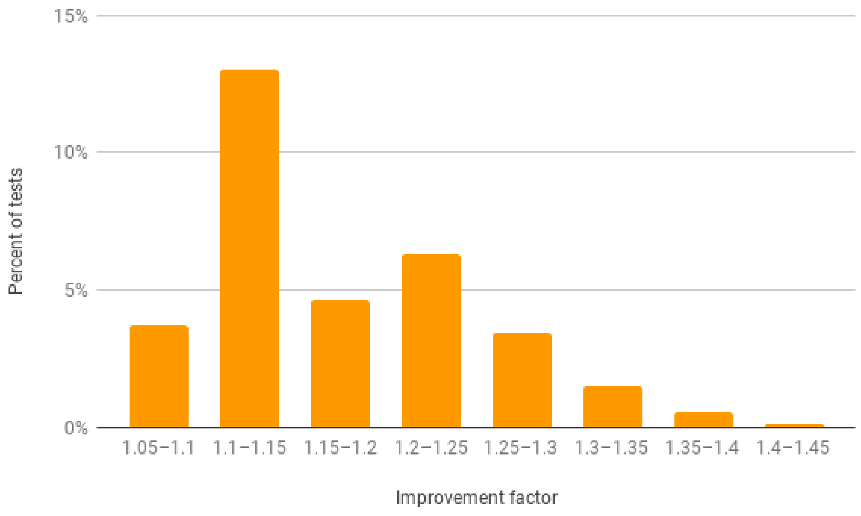


Figure 4.4: In 33% of the cases, our algorithm outperformed `ApplicativeDo`. This is the distribution of the improvements.

Here, in 33% of the cases give an improvement. The remaining 67% keeps the original rearrangement and thus performance is never worse than the original algorithm, given that the weights correspond to execution time.

4.3.1.3 Effects of incorrect costs

The previous evaluation worked under the assumption that the cost and evaluation time for an expression matched. Since performance is affected by faulty costs, an evaluation of the maximal decrease in performance is described in this section. The source of these costs is arbitrary, and different techniques for getting them may have different accuracy.

As in the previous evaluation, for a given program, each statement is given either weight 1, 2 or 4 and all combinations are tested exhaustively. To see how much the performance can decrease compared to the original algorithm, the worst possible rearrangement for the weighed case is compared to the original `ApplicativeDo` rearrangement.

Figures 4.5 and 4.6 show the maximal decrease factor for program one and two, respectively.

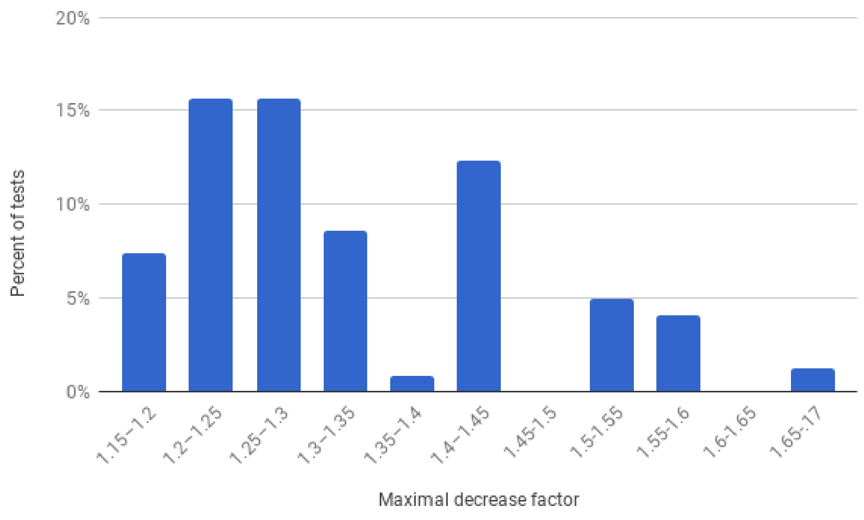


Figure 4.5: For the same weight distributions tested earlier, this figure shows the performance *decrease* factors for the *worst* possible desugarings of program one.

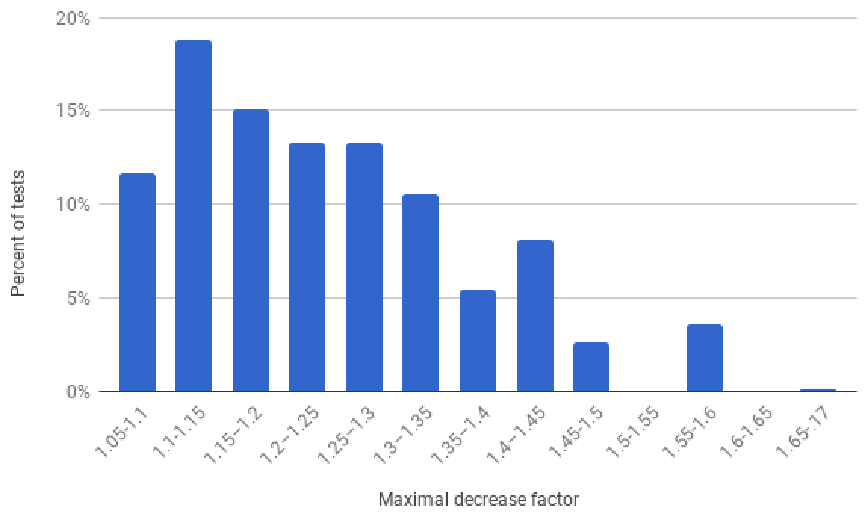


Figure 4.6: For the same weight distributions tested earlier, this figure shows the performance *decrease* factors for the *worst* possible desugarings of program two.

4.3.2 Empirical evaluation

As described in Section 3.2.2, `Haxl` can be used for arbitrary computation, not only remote data fetching. We define pure benchmarking functions with a fixed evaluation time; these functions are combined in `do`-blocks and then desugared by the improved `ApplicativeDo`. A series of experiments will test the performance of a `Haxl` program containing functions with scalable complexity, so each test instance has a different computational complexity of each statement. Each program is compiled using both our improved and the original `ApplicativeDo`, and the runtime performance is compared in 15 random statement weight distributions on a 2-core CPU (Intel Core i5-4210U @ 1.70GHz). For reproduction purposes, the evaluation time factors in the 15 tests are listed in Table 4.1.

4.3.2.1 Tests for program one

This program is the `do`-block from the Marlow et al. [6], which is a simple `do`-block used throughout this thesis.

```
    do  x1 <- a
        x2 <- b x1
        x3 <- c
        x4 <- d x3
        x5 <- e x1 x4
        return (x2,x4,x5)
```

Listing 41: Example `do`-block from Marlow et al. [6].

Compiling and running the program using the 15 different weight distributions with and without the improvements in `ApplicativeDo`, we get the following result:

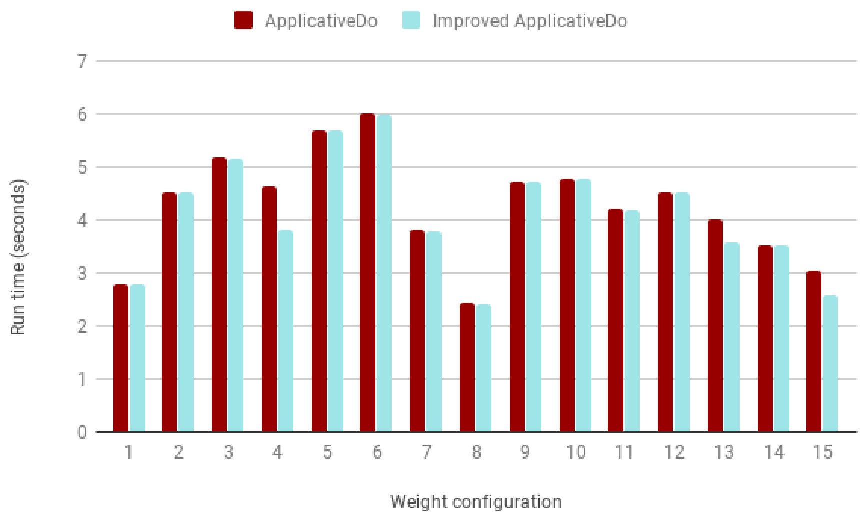


Figure 4.7: Test 4, 13 and 15 gains significant improvements compared to the original.

4.3.2.2 Tests for program two

This program, shown in Listing 40, is slightly bigger and has a more complex dependency structure than the first.

```
do  x1 <- a
    x2 <- d x1
    x3 <- b x1
    x4 <- f x2
    x5 <- e x1 x3
    x6 <- g x1 x3 x5
    return (x4,x6)
```

Listing 42: Example `do`-block for analytic evaluation.

Compiling and running the program using the 15 different weight distributions with and without the improvements in `ApplicativeDo`, we get the following result:

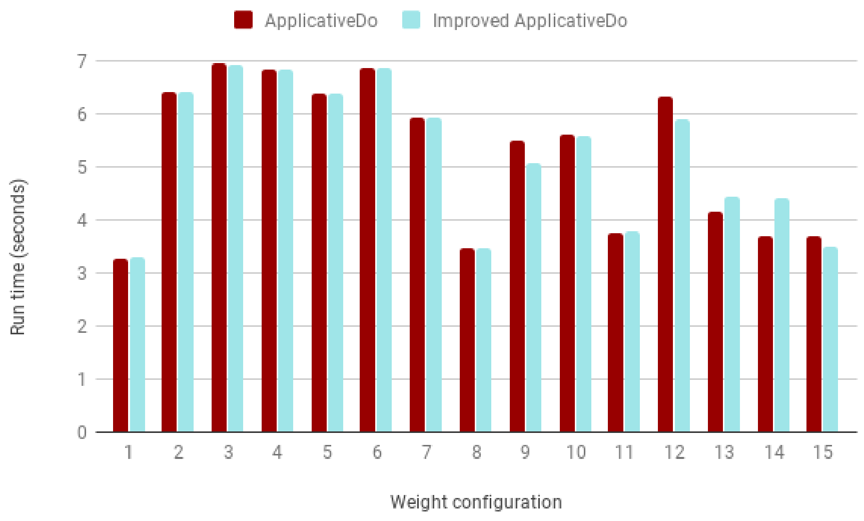


Figure 4.8: Both tests 9 and 12 have a significant improvement in runtime while tests 13 and 14 have significantly worse performance. The reason for this is elaborated upon in Section 6.1.

4.3.2.3 Tests for program three

This program is a permutation of 14 functions each named *a*-*n*. The dependencies for each function is randomly selected from the previously bound variables. This results in a large and, from our part, unbiased program well suited for a test.

```
    do  x1  <- c
        x2  <- b x1
        x3  <- d x2
        x4  <- n x1 x2 x3
        x5  <- a
        x6  <- k x1 x3 x5
        x7  <- l x1 x3 x6
        x8  <- f x5
        x9  <- m x6
        x10 <- e x2 x5
        x11 <- i x4 x10
        x12 <- g x1 x5 x8
        x13 <- h x5
        x14 <- j x4 x5 x10 x13
        return (x7, x9, x11, x12, x14)
```

Listing 43: A permutation of functions a-n with randomised dependencies.

Compiling and running the program using the 15 different weight distributions with and without the improvements in `ApplicativeDo`, we get the following result:

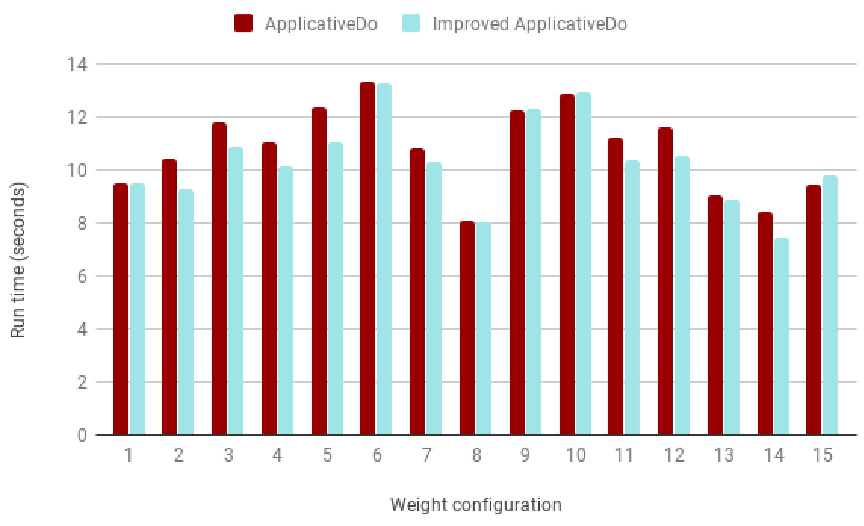


Figure 4.9: Tests 2, 3, 4, 5, 11, 12 and 14 have a significant improvement in runtime. Test 15 has worse performance; the reason for this is elaborated upon in Section 6.1.

Test	a	b	c	d	e	f	g	h	i	j	k	l	m	n
1	1	2	1	2	2	2	2	1	4	4	4	4	1	1
2	1	4	2	2	4	1	4	2	1	1	2	1	1	4
3	4	4	4	4	2	1	4	4	1	1	1	1	1	1
4	4	4	1	1	2	1	4	1	4	4	4	1	1	2
5	4	4	4	2	4	1	1	2	1	4	2	1	2	1
6	4	4	4	1	4	2	2	4	2	2	4	4	2	2
7	2	2	2	2	4	1	4	2	4	1	1	4	4	4
8	2	1	1	2	2	1	1	2	1	2	4	2	2	1
9	2	2	4	4	2	4	2	4	1	1	2	4	4	2
10	2	4	4	4	2	2	4	2	1	1	4	1	4	2
11	1	1	4	1	4	4	2	4	4	4	2	1	1	2
12	4	4	4	1	1	4	2	1	1	2	4	2	1	2
13	2	4	1	2	2	1	1	4	4	1	1	4	1	2
14	2	1	2	1	4	2	1	1	2	2	4	2	1	1
15	1	4	1	2	1	2	1	4	4	2	2	1	2	2

Table 4.1: Evaluation time factor for functions a,b...n

5
Related Work

5.1 Inferring type size

The paper *Automating Sized-Type Inference and Complexity Analysis* [9] gives a model for inferring a size of a computation in its type. A common example of such a sized type is a vector with type-level size, as found in the `fixed-vector` package.[1] The paper constructs a new type system with size analysis capabilities and type inference rules for the same, but in particular a *ticking monadic transformation* is further developed; it was first introduced in the paper *Denotational Cost Semantics for Functional Languages with Inductive Types* [10]. The idea is to use an *indexed monad*, where the types allow for an additional parameter that can be incremented every time the value is involved in a computation.

This can be used as an approximation of the run time complexity of a program, which is of relevance to this project. This estimate is not directly translatable to the wall-clock time requirement of a computation but does give a complexity *indication*; this could be useful as inferred weights for the desugaring in `ApplicativeDo`.

Avanzini and Lago [9] implements this in an abstract language, but Gissurarson started some work on a Haskell implementation during his MSc thesis, where this basic ticking transformation is applied to monadic computations [11]. However, limitations in the Haskell type system prevented this to mesh neatly with ordinary monads, as described by him in the GHC proposal issue at GitHub.[2]

5.2 Thunk-based implicit parallelism

In the paper *Feedback Directed Implicit Parallelism* [12], Harris and Singh develop a system for automatic parallelism in Haskell programs by investigating GHC thunking behaviour during profiling. The idea is to have a pool consisting of a low-priority worker-thread per physical core instead of the original GHC thunking behaviour.

[1] fixed-vector: `https://hackage.haskell.org/package/fixed-vector-1.1.0.0`
[2] OverloadedDo proposal: `https://github.com/ghc-proposals/ghc-proposals/pull/78`

These can steal thunks from other threads, using core idle time for automatic parallel evaluation of expressions.

These thunks, however, are of high value to evaluate in parallel only if they are heavier than the overhead of book-keeping the whole process. To decide on whether or not a thunk might be worth evaluating in parallel, they track the CPU cycle counter for thunk allocation sites in a profiling run, selecting series of thunks with a high mean evaluation time for speculative parallelism. This two-fold approach is what the paper title refers to with *feedback directed*.

The paper presents both a simulated limit study with real-world constraints such as the number of cores, cache and memory behaviour, et cetera ignored to find an upper limit to what the approach can theoretically allow, as well as actual empirical comparisons with the GHC 6.6 spark implementation using `par`. The limit study shows great promise. In the practical setting the method in the paper has a great deal of overhead, so for small thunk sizes, it is easily outperformed by the original thunk sparking mechanism. After a certain average thunk size, however, feedback directed implicit parallelism reaches a near-optimal speedup.

The reasons for the increased overhead is that with work stealing there are problems that need to be mitigated: preventing IO to be executed in a different order than intended, managing the speculative worker threads, organising the pool of sparked thunks, and handling thunk locks to prevent duplicate evaluation. The last of which is a particularly interesting improvement to the vanilla sparking behaviour since it minimises the risk of fizzled sparks. Both approaches work well between certain ranges of thunk sizes, so the paper concludes that a combination probably would work well in most cases.

Our work prioritises structuring code generated from `do`-blocks, if the evaluation of a resulting `<*>` operator then has more cost-balanced arguments, we believe this could positively impact the effect of the system described in the paper.

5.3 Schemik: an implicitly parallel language

Petr Krajča and Vilém Vychodil have through a series of papers investigated a language evaluation model for implicit parallelism. In *Data Parallel Dialect of Scheme: Outline of The Formal Model, Implementation, Performance* [13] we find a description of Schemik, a strict, stack based, implicitly parallel functional language with an execution model based on a pushdown automaton.

During interpretation of a program, an independent scheduler entity sifts through the execution stack for things deemed suitable for parallel evaluation, spawning lightweight threads for this purpose. There is an accompanying implementation showing the feasibility of the project available on the project home page at Sourceforge[3]. The representation of the evaluation semantics by an automaton allows proving determinism of programs, independently of scheduler parallelisation.

In *Software Transactional Memory for Implicitly Parallel Functional Language* [14], this model is further extended with software transactional memory (STM) to allow parallelisation of programs with side-effects. STM does not have to be used directly by the programmer but is used in the backend during evaluation. This is used to make sure each parallel evaluator has consistent memory during execution: when a parallel job is done executing, memory changes are committed to the program-global memory through STM, which through a case-analysis handles possible collisions. The most severe such collision is when a parallel thread has written to something during reads from a later thread. This is resolved by re-running the later computation with a new STM snapshot of the state, since the *real* evaluation order is specified by the order of arguments. The paper does no extensive analysis of the time consumption of this approach, but a small handful of benchmarks show positive indications.

In *Incremental JIT Compiler for Implicitly Parallel Functional Language* [15], a Just-In-Time (JIT) compiler is added to the model. The basis of doing so is that the interpreted nature of Schemik has heavy overhead, and by doing on-the-fly compilation of assumed costly expressions, great speedups are gained. This evaluation of compilation-worthiness is not described to any bigger extent, but is said to be based on: *lists, or lists fulfilling further criteria, such as containing sub-lists or user-defined functions.*

5.4 Data parallel languages

The area of data parallel languages is growing, and related to this thesis in the way they enable working with parallel computation with low programmer overhead. However, these languages can seldomly be referred to as widely-adopted, general purpose languages in contrast to Hasklell. The NESL [16] language was one of the pioneering ones that allow nested data parallelism primitively and a similar approach has been taken by the more recent developments of Futhark [17], which has extended the possibilities in an impressive fashion.

Data Parallel Haskell [18] is a project aiming to enable nested data parallelism for multi-core CPU computation in Haskell. There is basic support, but the project has since run out of funding; hopefully, community efforts allow this development to continue. Accelerate [19] brings data parallel GPU computation to Haskell through high-level array operations but does not support nested parallelism, additionally it requires explicit use of parallel computation functions.

NVIDIA has also done advances in this field with the functional, data parallel lan-

[3]Shemik project page: `http://schemik.sourceforge.net/`

guage NOVA [20]. NOVA shows impressive performance for both multi-core CPUs as well as GPUs but is always compared to CUDA C and parallel C, in the setting of raw computer graphic algorithms.

6
Discussion and conclusion

Benchmarks in the evaluation section show that the extension of the `ApplicativeDo` cost model indeed allows for greater performance in many cases. In general, it seems our enhancements give better results with more complex `do`-blocks; more statements and dependencies means more possible solutions to decide between.

The analytic tests of programs in sections 4.3.1.1 and 4.3.1.2 both show improvements, but the longer program outperforms `ApplicativeDo` in three times as many cases. The empirical evaluation also seems to support this hypothesis; the shorter programs in sections 4.3.2.1 and 4.3.2.2 show significant improvements to execution time in 20% and 13% of the cases, respectively, while in the longer program evaluated in Section 4.3.2.3 we see it in 53% of the cases.

6.1 Peculiar data points in evaluation

There are a few places of particular interest in the empirical data; here we will discuss these cases.

Test program 2 Running the program in Section 4.3.2.2 sometimes show *worse* performance when desugared by our algorithm, specifically in cases 13 and 14. We scrutinised the generated code in these cases, and found that the two algorithms return different rearrangements that represent the same cost. We get different rearrangements because of the split optimisation shortcut described in Section 3.1.1; the shortcut does not trigger in our algorithm in these cases.

The reason that the vanilla `ApplicativeDo` code then runs faster in this case is that it happens to trigger a performance optimisation later in the simplifier; the shortcut in `ApplicativeDo` does not affect runtime by itself and is just a way to speed up the desugaring algorithm.

It is possible that this effect is hidden in more of the test cases, creating some noise in the data; there is however no reason to suspect that this is not equally likely to happen the other way around. For the particular cases above this explanation was verified by turning off GHC optimisations, but re-running *all* of the tests would be too time-consuming so this argument will have to suffice.

Test program 3 In test case 15 we see a slight decrease in performance with the code desugared by our algorithm. This program is more complex, and the desugarer generates a segment with three statements connected by applicatives. `Haxl` schedules this in three threads, and our test machine only has two cores. This slight worsening is therefore explained by threading overhead.

6.2 Weight annotations

The optional programmer weight annotations work well in practice. Particularly the implementation based on the GHC lexer and parser is neat to use. This part of the thesis was investigated for two purposes:

- Enabling a way to provide correct costs for evaluation of the new and improved `ApplicativeDo` algorithm.
- Exploring the practicality of using programmer annotations to affect an underlying model for implicit parallelism.

The first purpose was fulfilled nicely; by defining functions with known relative computational costs and annotating them accordingly, a fair evaluation of the improvements of `ApplicativeDo` could be done.

For the second purpose, the conclusion is that annotating is a nice method of affecting the implicit parallelism from the desugarer, with very little programmer overhead. There are however some usability issues. Firstly, estimation of evaluation times of Haskell expressions is notoriously difficult for a number of reasons. For one, lazy evaluation makes it hard to figure out exactly when evaluations take place, if at all. Another reason is that there are a lot of performance optimisations in the compiler which are difficult to determine if they will trigger, and if so, how they are going to affect the results. These are some examples of such automatic optimisations that are difficult to predict:

- Fusion, the removal of intermediate data structures. This is also referred to as *deforestation*.
- Common subexpression elimination, restructuring of code to prevent unnecessary evaluation of shared expressions.
- Inlining, preventing overhead from function calls by substituting the implementation.
- Strictness analysis, trying to figure out which expressions are definitely going to be needed to bypass certain laziness overhead.

6.2.1 Using GHC profiling for weight insights

To help mitigate the difficulty of estimating runtime requirements, the programmer can make use of the GHC profiling tools[1]. This way one can do a test run of the program without weights, inspect the relative evaluation times of the different statements, add corresponding weights and see if it makes a difference.

Using the `SCC` pragma (abbreviation for "Set Cost Centre"), the programmer can track the evaluation time of particular expressions.[2] This pragma can take an additional name parameter, making identification in the profiling dump easy.

6.3 When we see improvements

It seems that the extension of `ApplicativeDo` is more likely to give desired improvements over the original when desugaring a bit longer functions with several possible rearrangements. From inspecting which particular weight configurations indeed reach a lower cost in our algorithm in the analytic tests, we conclude that no effect is seen when the weight distribution is close to uniform, because this works similarly to the unit cost model. We can also see that in all of the cases that we *do* see improvements, the weight distributions are uneven. This makes sense because the bigger the relative difference between statements weights, the less accurate is the unit cost model.

So we know `do`-blocks need to be somewhat complex for the improvements to shine, but how does typical `Haxl` code look? `Haxl` is very new and there are no large open-source projects using it at time of writing. However, mPowered Solutions[3] use `Haxl` in one of their products, and have been kind enough to let us analyse parts of their code base. Unfortunately, this project was not written in such a way that it will be affected at all by `ApplicativeDo`. All data fetches are independent and explicitly connected with applicative operations in big chunks. There is no need for `do`-notation in the implementation, and they already get good performance from `Haxl` because the fetches are always independent.

Facebook is most likely the biggest user of `Haxl` and the library was developed by them. Unfortunately it was not possible to get access to production code, but in the paper *There Is No Fork* [7] some statistics about the `Haxl` code base at Facebook is presented; Figure 6.1 shows some of these results. The data comes from running a random sample of 10000 requests from the production system Sigma, a rule engine used to detect malicious content.

[1]GHC profiling: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html`

[2]Cost centre profiling: `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html#inserting-cost-centres-by-hand`

[3]mPowered Solutions: `https://mpowered.co.za/`

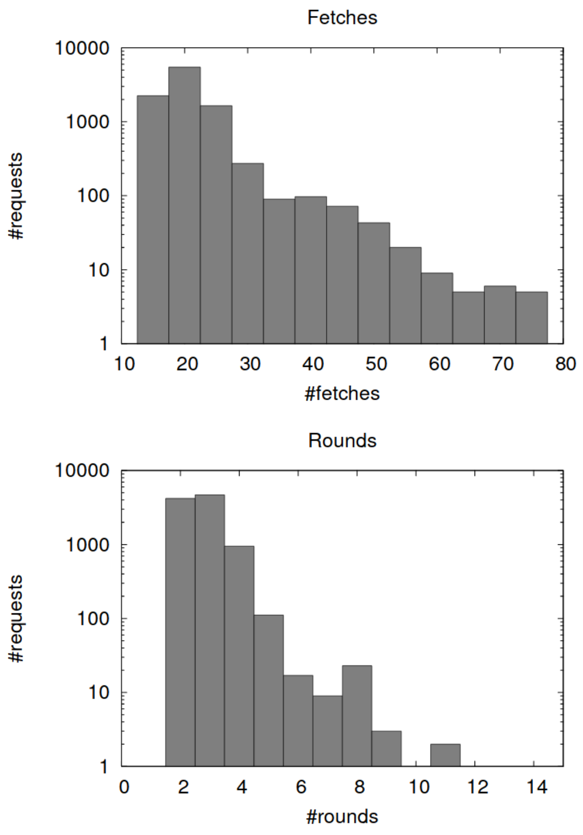


Figure 6.1: `Haxl` statistics from Facebook. Re-published from *There Is No Fork* [7] with author permission.

We can see that they all perform *at least* 10 fetches, but sometimes up towards 80. The fetches are always performed in at least 2 rounds, but sometimes up to 11; this indicates that there is indeed a lot of dependencies in the code, since otherwise they could all be performed in the same round. The paper discusses relatively small differences in latency between these requests, even though there are heavy outliers. However, Marlow references use of `Haxl` in a build system [21], which indeed seems like a case with heavily differing time requirements for each operation.

From this we conclude that this industry application of `Haxl` probably could benefit from our improved `ApplicativeDo`.

6.4 Future work

In this section we will expand on ideas that would further explore the neighbourhood of our work. Both ways to expand the annotative approach, as well as other sources for costs, are proposed. In particular, benchmarking on actual production code would be insightful, or at least on more parallel machines. `Haxl` is still very new, so there are unfortunately not much open-source code available.

6.4.1 Function weight inference

As our implementation of weight annotation currently works, calling a function in a do-statement cannot infer the weight from annotations in the definition of that function. This would be a worthwhile extension of the current implementation, but one thing is still unclear. Consider the code in Listing 44; since the weight annotation used in `subfunction` are relative to its other statements, we are not sure of how to use that total cost of 3, in relation to the statements in `mainFunction`.

```haskell
subfunction :: GenHaxl a
subfunction = do
  a <- something {-# Weight 2 #-}
  somethingLighter a

mainFunction :: GenHaxl a
mainFunction = do
  a <- subfunction
  computeResult a
```

Listing 44: When inferring the weight of a function call in `mainFunction`, whether it is fair to use the total cost of three from `subfunction` depends on the relative weight of `computeResult`.

A possible path for inferring the weight from a function definition, that we did not have time to explore, is through source-to-source transformations. The packages `haskell-src-exts`[4] and `haskell-names`[5] can be used to parse and rename the module while doing the additional analysis.

6.4.2 Type-based weight inference

The method of annotating as a basis for the cost model can, as described in Section 6.2, be prone to error since a wrong estimate of run time can lead to a worsening of `ApplicativeDo`. Therefore, other means of getting the cost information into the `ApplicativeDo` algorithm would be an interesting research subject.

One way of getting weight information could be using sized typed functions; Krajča and Vychodil saw good results when deciding upon speculative parallelism based on the type of value [13]. Size information like that, found in `Vector` and similar data structures, could help decide if parallel evaluation would be worth the effort. Worth noting is that both laziness and infinite data structures complicate things in the Haskell setting.

[4]`https://hackage.haskell.org/package/haskell-src-exts`
[5]`https://hackage.haskell.org/package/haskell-names`

Another way could be through size type inference, as explained in Section 5.1, where a function with a greater inferred size may be given a greater cost and vice versa. This is a very new, but nevertheless interesting path.

6.4.3 Investigating interoperation with thunk-based automatic parallelism

The paper *Feedback Directed Implicit Parallelism* [12] presents a promising take on implicit parallelism which may be affected by our improvements, as described in Section 5.2. The system is implemented for GHC 6.6, before `ApplicativeDo` was conceived. This is unfortunate because it is not possible to investigate how they work together without porting the implementation to a more recent GHC version, so further work is needed.

6.4.4 A more general library for executing implicit parallelism

Currently, `Haxl` is, to our knowledge, the only library that can make use of the implicit parallelism of applicative expressions. Unfortunately, `Haxl` can only make use of implicit parallelism between operations encoded as data fetches with its primitives. Since `ApplicativeDo` generates applicative expressions, new means of using implicit parallelism could increase the usability for `ApplicativeDo` and would be a step closer to fundamental automatic parallelism.

The task to make automatic parallelism is complicated even though the parallelism is there for the taking, because of the well-known problems of overhead that occur when parallel scheduling becomes too fine-grained. Perhaps a way to handle it in this specific case could be through using a low priority thread per core together with work stealing, in a similar way as described by Harris and Singh [12].

6.5 Conclusion

Our research questions, as presented in the introduction of Section 2.3, are:

- Can we extend the `ApplicativeDo` algorithm to make better decisions, given information about statement evaluation cost?
- How can we provide the `ApplicativeDo` algorithm with this cost information, with minimal programmer overhead?

For the former, we have shown how the cost model can be extended to work with non-unit statement costs. The results show that we indeed see improvements, both under analytic reasoning and empirical, wall-clock runtime evaluation using `Haxl`.

For the latter, we have investigated the path of optional programmer weight annotations on `do`-statements. It is a simple way to get cost information to the compiler and `ApplicativeDo` algorithm, without having to do anything else than specify the relative cost for the statement with a short pragma. The main issue with this approach seems to be that estimating the actual evaluation time can be difficult in Haskell, but experimentation is easy and the GHC profiling tools can provide assistance.

As further work, we suggest both extensions to our annotative approach as well as a direction using type analysis; this has been shown to be a promising approximation in related research. To mitigate the risk of depending too much on a single solution that may have its drawbacks, a suitable way forward may be a combination of the two. We also hope to see further developments in parallel execution of independent expressions like in `Haxl`, but in a more general setting. Perhaps interoperation with thunk-based automatic parallelism is a particularly interesting path.

Bibliography

[1] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler, "The glasgow haskell compiler: a technical overview," in *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, vol. 93, 1993.

[2] S. P. Jones, *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[3] M. Lipovaca, *Learn you a haskell for great good!: a beginner's guide.* No Starch Press, 2011.

[4] G. Hutton, *Programming in Haskell.* Cambridge University Press, 2016.

[5] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, (New York, NY, USA), pp. 1–14, ACM, 1992.

[6] S. Marlow, S. Peyton Jones, E. Kmett, and A. Mokhov, "Desugaring haskell's do-notation into applicative operations," in *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pp. 92–104, 2016.

[7] S. Marlow, L. Brandy, J. Coens, and J. Purdy, "There is no fork: An abstraction for efficient, concurrent, and concise data access," in *ACM SIGPLAN Notices*, vol. 49, pp. 325–337, ACM, 2014.

[8] A. Ranta, *Implementing programming languages. An introduction to compilers and interpreters.* College Publications, 2012.

[9] M. Avanzini and U. D. Lago, "Automated sized-type inference and complexity analysis," in *Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017*, pp. 7–16, 2017.

[10] N. Danner, D. R. Licata, and R. Ramyaa, "Denotational cost semantics for functional languages with inductive types," in *ACM SIGPLAN Notices*, vol. 50, pp. 140–151, ACM, 2015.

[11] M. Gissurarson, 2017-11-23. Private communication.

[12] T. Harris and S. Singh, "Feedback directed implicit parallelism," in *ACM SIGPLAN Notices*, vol. 42, pp. 251–264, ACM, 2007.

[13] P. Krajca and V. Vychodil, "Data parallel dialect of scheme: outline of the formal model, implementation, performance," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1938–1939, ACM, 2009.

[14] P. Krajca and V. Vychodil, "Software transactional memory for implicitly parallel functional language," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2123–2130, ACM, 2010.

[15] P. Krajca, "Incremental jit compiler for implicitly parallel functional language," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pp. 1511–1518, IEEE, 2013.

[16] G. E. Blelloch, "Nesl: A nested data-parallel language.(version 3.1)," tech. rep., Carnegie Mellon School of Computer Science, 1995.

[17] T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: purely functional gpu-programming with nested parallelism and inplace array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 556–571, ACM, 2017.

[18] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data parallel haskell: A status report," in *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, (New York, NY, USA), pp. 10–18, ACM, 2007.

[19] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pp. 3–14, ACM, 2011.

[20] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea, "Nova: A functional language for data parallelism," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, p. 8, ACM, 2014.

[21] S. Marlow, "Haxl: A big hammer for concurrency," 2017. `https://www.youtube.com/watch?v=sT6VJkkhy0o`, visited 2018-05-25.

[22] "Picture of the glasgow haskell compiler pipeline." `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscPipe`. Visited May 7, 2018.

A

Haskell language extensions

Haskell language extensions are used for enabling new features that are not present in the base language. The GHC extension system is used for adding new functionality without breaking the language specification[1], a system put in place to solidify the language API for longer periods of time to make it possible for several compilers to coexist! This way GHC can still add novel features while still by default adhering to the Haskell language specification. A compiler extension is enabled either by passing a flag to the compiler:

```
ghc -XApplicativeDo -XGADTs
```

Listing 45: GHC flags.

or on a per-module basis with source code pragmas in the top of the file:

```
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE GADTs #-}
```

Listing 46: Examples of source code pragmas.

In this appendix, we explain the language extensions we use in this project, or which are used in modules we work with in the GHC code base, in alphabetical order.

A.1 `ApplicativeDo`

The `ApplicativeDo` extension is explained in particular detail in Section 3.1.

[1] `https://wiki.haskell.org/Haskell_2010`

A.2 BangPatterns

This extension enables *strict* pattern matching, in contrast to Haskell's per-default lazy evaluation. This means that instead of the usual behaviour where the evaluation of a pattern match is deferred until the result is needed, the matched expression is forced to evaluate to *weak head normal form*[2] — often the first constructor. The difference in evaluation strategy can be seen in this example:

```haskell
lazy :: a -> Int
lazy x = 5

strict :: a -> Int
strict !x = 5

> lazy undefined
5

> strict undefined
*** Exception: Prelude.undefined
```

Listing 47: The difference in evaluation semantics when using bang patterns.

Strictness is useful to avoid function evaluation building big lazy computations before actually evaluating them. Introducing some strictness can speed up execution and lower the memory usage in some cases, but also make sure things actually do happen when the function is executed and not until the function results are needed. The latter is important when working with concurrent execution.

This extension is, in general, harmless but some care might be needed in some cases not to force evaluation of an infinite computation where a lazy evaluation might yield the expected result.

A.3 FlexibleInstances

This extension allows more liberal type class instance declarations, where the default is very conservative. Natively, GHC only allows type class instance declarations of this form:

```haskell
instance Class (T a1 ... an) where
```

Listing 48: `a1` through `an` are *type variables* and can only appear *once* in the instance head

[2]https://wiki.haskell.org/Weak_head_normal_form

The consequence is that the following instances do not pass typechecking — the first uses nested constructors, the second additionally uses the same type variable twice — but the `FlexibleInstances` extension makes both valid:

```haskell
instance Class [Int] where

instance Class (SomeType a (IORef a))
```

Listing 49: Two type class instance declarations not possible without the `FlexibleInstances` extension.

This extension is frequently and safely used; it cannot introduce ambiguity or overlapping of instance declaration that would confuse the type checker.

A.4 GADTs

This extension enables *Generalised Algebraic Data Types*, a very commonly used feature of the type system. It allows constructors of an algebraic data type to yield different types. The following kind of data type declaration is very practical, but not allowed without `GADTs` since the type variable `a` is instantiated to different types depending on the constructor:

```haskell
data MyType a where
    MyInt  :: Int  -> MyType Int
    MyBool :: Bool -> MyType Bool
```

Listing 50: A generalised algebraic data type, where constructors can instantiate the type variable `a` differently.

This is one of the most heavily used language extensions and is perfectly safe to use.

A.5 MultiParamTypeClasses

This extension enables writing type classes with more than one type in the instance head. By default, Haskell only allows type classes with kind `* -> *`, meaning they need a type to construct an instance of its type. The extension loosens this to an arbitrary number of parameters. For example, in `Haxl` we represent a data source with the following class definition head:

```haskell
class (DataSourceName req, StateKey req, ShowP req) => DataSource u req where
```

Listing 51: A multi parameter type class, enabled by this extension. `DataSource` takes two type arguments, meaning it has the *kind* `DataSource :: * -> * -> *`.

As we can see, this type class requires *two* types to be instantiated: `u` is the type of environment for global variables, and `req` is the type of request. This type of declaration depending on several types would not be possible without the extension.

`MultiParamTypeClasses` is a problematic language extension that can easily introduce problems in the type system in the form of ambiguity — when the type checker cannot decide which instance to use — or undecidability, where the type checker is no longer guaranteed to terminate when checking our types. These issues can be addressed in several ways, including type families[3], functional dependencies[4], and straight up allowing undecidable instances[5].

A.6 MultiWayIf

This purely syntactic extension adds a new way to write compact pattern-matching conditional expressions, instead of a chain of if-then-else expressions that are limited to a single branch.

With `MultiWayIf` activated, we can write code with pattern-match predicates basically anywhere, like this:

```haskell
f :: Int -> Bool -> String
f i False = "No result requested"
f i True  = show i ++ ": " ++
    if | i < 10 -> "yes"
       | i > 20 -> "no"
       | otherwise -> "maybe"
```

A.7 OverloadedStrings

Dealing with the different types of string representations in Haskell can be quite annoying since string literals *always* have the fixed type `[Char]`. This is different to how integer (and other) literals are treated, with a polymorphic type: `Num a => a`.

[3]https://wiki.haskell.org/GHC/Type_families
[4]https://wiki.haskell.org/Functional_dependencies
[5]http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#ghc-flag--XUndecidableInstances

An integer literal can be used wherever anything from the `Num` type class is expected, and this includes dozens of instances.

For strings, we sometimes want to use `Text`, `ByteString`, or other text data types. Instead of using explicit casting functions from `String` everywhere, we can instead use this extension to give literals the more general type `Data.String.IsString a => a`.

This extension is very common and safe. In certain circumstances, it can cause issues with ambiguity, but in practice, the surrounding types define the target instance precisely.

A.8 StandaloneDeriving

This extension allows deriving type class instances automatically, even if they are not in the same file as the type class declaration, which is one of the constraints otherwise. It also enables more complex derivations, particularly deriving instances for `GADTs`, which is another limitation of standard Haskell instance derivation. It is also possible to make instances with additional constraints not present in the data type; this is impossible with the standard `deriving` keyword:

```haskell
data MyType a = Constr a

deriving instance Eq a => Eq (MyType a)
```

Listing 52: An example of a stand-alone deriving clause for the `Eq` type class. The instance is derived if the type wrapped has an `Eq` instance; this "conditional instance" is impossible without the extension.

What we get from this is a free `Eq (MyType a)` instance for `MyType` whenever it contains something that itself has an `Eq` instance. This extension is safe and cannot introduce any particular problems.

A.9 TypeFamilies

Type families are a very powerful concept that vastly extends the Haskell type system; covering the full power of the extension is out of scope for this section so we will look at one example.

The Haskell type class construct is versatile but can be further extended in a natural way with the use of *type families*. The examples below are lent from the excellent

24 days of GHC extensions advent calendar.[6]

By themselves, a type class can be used to associate a set of functions with a certain type:

```haskell
class IOStore store where
  newIO :: a -> IO (store a)
  getIO :: store a -> IO a
  putIO :: store a -> a -> IO ()
```

This is nice, and we can create instances for different types of IO storage (`MVar` and `IORef`, for example), making it possible to write code that is polymorphic over the specific type of store used:

```haskell
instance IOStore MVar where
  newIO = newMVar
  getIO = readMVar
  putIO mvar a = modifyMVar_ mvar (return . const a)

instance IOStore IORef where
  newIO = newIORef
  getIO = readIORef
  putIO ioref a = modifyIORef ioref (const a)
```

With this, we can create a function that works for different kinds of storage, using the function from the type class above:

```haskell
storeBooksIO :: IOStore store => [Book] -> IO (store [Book])
```

This is nice, but what if we would like to create an instance for `TVar`? Unfortunately, we have locked ourselves into working with stores in the `IO` monad whereas `TVar` lives in the `STM` monad, so this is not possible!

Fortunately, type families can help us by allowing us to associate a *type* with another *type*; in this case, something of kind `* -> *`, meaning a first order type constructor:

```haskell
class Store store where
  type StoreMonad store :: * -> *
  new :: a -> (StoreMonad store) (store a)
  get :: store a -> (StoreMonad store) a
  put :: store a -> a -> (StoreMonad store) ()
```

[6]Credit to Oscar Charles, published under a permissive Creative Commons license: `https://creativecommons.org/licenses/by-nc-nd/3.0/`

Now the type class also forces you to specify what type you want `StoreMonad` to be when creating an instance of it:

```haskell
instance Store IORef where
  type StoreMonad IORef = IO
  new = newIORef
  get = readIORef
  put ioref a = modifyIORef ioref (const a)

instance Store TVar where
  type StoreMonad TVar = STM
  new = newTVar
  get = readTVar
  put ioref a = modifyTVar ioref (const a)
```

Now our polymorphism is not limited over `IO` storages, but different storage in different *monads* too. We can use the same implementation as earlier with only a slight change to the type of the function:

```haskell
storeBooks :: (Store store, Monad (StoreMonad store))
           => [Book] -> (StoreMonad store) (store [Book])
```

B

Discovery of unexpected GHC behaviour

While working with the example shown in Listing 21 in Section 3.2.2.1, we tried different pure computations each with a heavy machine workload. We ran into a weird issue with a big strict left fold (`foldl’`), where there is drastically different memory consumption depending on whether or not there is an optional type annotation on the statement.

`foldl’` is characteristically used to avoid heavy memory load that can occur when using the regular `foldl`: the runtime building very large chains of unevaluated function applications, before applying the last argument and letting the whole thing reduce. Instead, `foldl’` *strictly* applies the function in between each step, yielding a new value that can be passed to the next function. This avoids the thunking behaviour which is so heavy on the RAM.

The following snippet shows the different evaluations steps of `foldl` versus `foldl’`; the intermediate expressions of `foldl` needs to be stored somewhere, while `foldl’` condenses into one integer that is passed along.

```
bad = foldl (+) 0 [1..5]
-> foldl (+) (0 + 1) [2..5]
-> foldl (+) (0 + 1 + 2) [3..5]
-> foldl (+) (0 +1 + 2 + 3) [4..5]
-> foldl (+) (0 + 1 + 2 + 3 + 4) [5]
-> foldl (+) (0 + 1 + 2 + 3 + 4 + 5) []
-> 0 + 1 + 2 + 3 + 4 + 5
-> 15

good = foldl' (+) 0 [1..5]
-> foldl' (+) 1 [2..5]
-> foldl' (+) 3 [3..5]
-> foldl' (+) 6 [4..5]
-> foldl' (+) 10 [5]
-> foldl' (+) 15 []
-> 15
```

However, when designing a prototype for mocking computations as `Haxl` data fetches we encountered very odd memory behaviour with `foldl'`. Extra expression type annotations are optionally available in Haskell, and depending on whether or not this was added to a certain statement in this snippet, the memory usage when running the compiled program was vastly different. The following snippet is a minimised example of where it occurs:

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE BangPatterns #-}

import Data.IORef
import Data.List

data Heavy a where
  Mock :: Heavy Integer

{-# NOINLINE runHeavy #-}
runHeavy :: Heavy a -> IORef a -> IO ()
runHeavy Mock var = do
  -- This runs as expected:
  let !n = foldl' (+) 0 [1..100000000] :: Integer

  -- Without the type annotation it eats RAM like crazy
  -- let !n = foldl' (+) 0 [1..100000000]
  writeIORef var n

main = do
  ref <- newIORef 0
  runHeavy Mock ref
  readIORef ref >>= print
```

The profiling information for running with and without the type annotation is shown below. Interestingly, the version without type signature actually runs a bit faster but the runtime allocates a whopping 5.6 GB of additional RAM.

```
-- With type annotation
3,200,051,784 bytes allocated in the heap
Total   time    2.328s  (  2.350s elapsed)

-- Without type annotation
8,800,051,816 bytes allocated in the heap
Total   time    2.148s  (  2.167s elapsed)
```

This is surprising when using a function that is supposed to be a particularly memory-friendly fold. It is unclear if this is "odd but intended optimisation" or a bug; the issue is raised with the GHC developers.

C

The GHC compilation pipeline

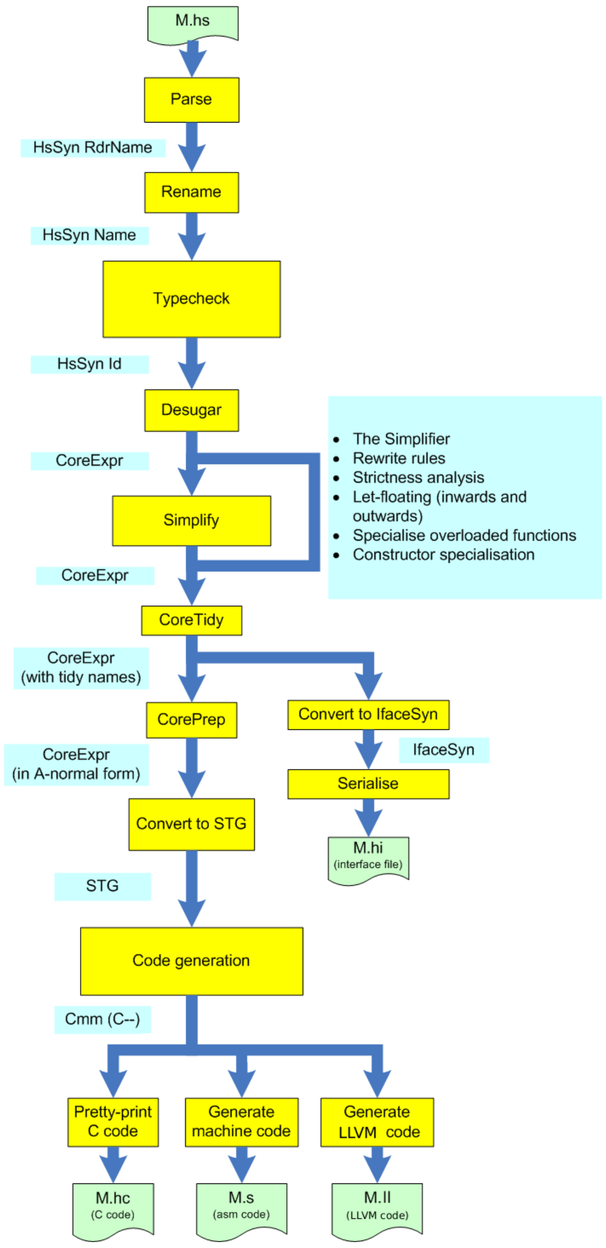


Figure C.1: A visualisation of the compilation pipeline, showing the order of compilation stages [22].

D

The GHC plugin system

GHC supports compiler plugins in the form of additional core-to-core passes, applying optimisations through repeatedly running transformations of the internal language representation (a tiny language called Core). This happens in the Simplify stage (see appendix C); in a compiler plugin the annotations are readily available, but it is long after `ApplicativeDo` has finished.

In addition to core plugins, it is also possible to write plugins for the type checker. This enables extensions to the constraint solving algorithm that is used internally to verify that types are indeed correct. Examples of type checker plugins include a Nat-solver for type level natural number constraints, and a type level implementation of units of measure, enabling the type checker to prevent you from adding a meter to a Newton, for example. The problem is the same with type checker plugins as with core; they are active too late in the compilation pipeline.

The (simplified) interface for writing a plugin is shown in code listing 53. A plugin should implement these functions, where some out of them may do nothing. The function `installCoreToDos` allows insertion of an external core pass (a `CoreToDo`) into a specific point of simplifier pipeline, and `tcPlugin` allows you to return a `TcPlugin` which can extend the constraint solving capabilities of the GHC type checker.

```haskell
data Plugin = Plugin {
    installCoreToDos :: [CoreToDo] -> CoreM [CoreToDo]
  , tcPlugin :: Maybe TcPlugin
  }
```

Listing 53: The interface of a GHC compiler plugin that allows extension of the simplifier and constraint solving capabilities of the type checker.

D.1 Usage of compiler plugins

A compiler plugin can be activated with a command line flag after it is built, made into a Cabal package and registered in the system (see Section D.3.1). The flag `-fplugin Plugin` activates the plugin provided by a module `Plugin`, and the flag `-fplugin-opt Plugin:arg` can be used to pass an argument `arg` to the plugin.

For the plugin developed in Section D.3, usage in compilation with a locally built compiler looks like this, from the location of the module we want to compile:

```
$ .../ghc/inplace/bin/ghc-stage2 -fplugin EarlyAnnPlugin Module.hs
```

D.2 Extended plugins proposal

There is a current GHC proposal called Extended plugins proposal[1,2] that aims to generalise the current plugin system to enable plugins for earlier stages than the type checker and simplifier. The (simplified) extended plugin interface is shown in code listing 54 below.

```haskell
data Plugin = Plugin {
    installCoreToDos :: [CoreToDo] -> CoreM [CoreToDo]
  , tcPlugin :: Maybe TcPlugin

  -- Proposed additions below
  , parsedResultAction :: HsParsedModule -> Hsc HsParsedModule
  , renamedResultAction :: Maybe(RenamedSource -> Hsc ())
  , typeCheckResultAction :: TcGblEnv -> Hsc TcGblEnv
  , spliceRunAction :: LHsExpr GhcTc -> TcM (LHsExpr GhcTc)
  , interfaceLoadAction :: ModIface  -> IfM lcl ModIface
  }
```

Listing 54: The proposed extension of the plugin interface.

The following is a high-level explanation of the capabilities of each added function:

- `parsedResultAction` allows access and modification to the source *before* the Rename stage. This can for example be used to print a little stack trace for failing `assert` checks.[3] This function is of particular interest to us for collection annotation data.

[1] `https://ghc.haskell.org/trac/ghc/wiki/ExtendedPluginsProposal`

[2] Development branch with extended plugins: `https://phabricator.haskell.org/D4342`

[3] Suggestions enabling debugging `assert`: `https://ghc.haskell.org/trac/ghc/ticket/14443`

- `renamedResultAction` enables inputting a read-only function that is run after the Rename stage. This enables source analysis tools to grab the renamed syntax before it is changed later in the pipeline.

- `typeCheckResultAction` allows modification of the type environment after type checking. This would allow all kinds of type hackery, but no applications are yet in sight.

- `spliceRunAction` enables modifications to Template Haskell[4] expressions before they are run. Basically, this allows metaprogramming of metaprogramming.

- `interfaceLoadAction` allows modification of a module interface when it is being loaded into another module. This is also useful for source analysis since the tool can know for example which identifiers are from which modules.

The function `parsedResultAction` gives us access to what we need for inspecting annotations *before* the Rename stage, and hence before `ApplicativeDo`. The constructor chain that leads to what we want is shown in code listing 55 below.

```haskell
parsedResultAction :: HsParsedModule -> Hsc HsParsedModule

-- RdrName is a representation of source code identifier names
data HsParsedModule = HsParsedModule { ... ,
    hpm_module :: Located (HsModule RdrName) }

data HsModule = HsModule { ... ,
    hsmodDecls :: [LHsDecl name] }

type LHsDecl id = Located (HsDecl id)

data HsDecl id = ... | AnnD (AnnDecl id) | ...

-- HsExp is any type of (in this case annotated)
-- Haskell expression
data AnnDecl name = HsAnnotation (AnnProvenance name)
                                 (Located (HsExpr name))


data AnnProvenance name = ValueAnnProvenance name | ...
```

Listing 55: The (stripped down) constructor chain that allows access to annotations from `parsedResultAction`. `Located` is a simple type that wraps information with source code position.

Through this series of data connections, we can dig up the annotation with a front

[4]Template Haskell: `https://wiki.haskell.org/Template_Haskell`

end plugin, manually populate the global type checking environment (`TcGblEnv`) with the information and thereby making it available in `ApplicativeDo`. The `AnnProvenance` data type contains a representation of the binder name we annotate in the source code and the `HsExpr` is the annotation payload.

D.3 Development of the compiler plugin for annotations

Using GHC patched with the implementation of the extended plugins proposal (how this is done is explained in Section E.1.1), we can build a plugin with the brand new functionality of the interface described in Section D.2. The plugin code is fairly straightforward and presented in code listing 56.

```
module EarlyAnnPlugin (plugin) where

  import GhcPlugins
  import HsDecls (HsDecl(..), AnnDecl(..), AnnProvenance(..))
  import HsSyn (hsmodDecls)
  import HsExpr (HsExpr(..))
  import HsExtension (IdP(..), GhcPs)
  import TcRnMonad (getTopEnv, updTopEnv)
  import HscTypes (hpm_ann_from_parser)
  import System.IO.Unsafe (unsafePerformIO)

  -- | A compiler plugin must export a value of this type. We
  -- take an "identity" plugin ('defaultPlugin'), and replace
  -- the 'parsedResultAction' field with our implementation:
  -- 'prepareAnnotations'.
  plugin :: Plugin
  plugin = defaultPlugin {
      parsedResultAction = slurpTopLvlAnn
    }

  -- | Extracts binders with payload from 'ANN' pragmas through
  -- 'HsParsedModule' and adds them back to a specialised field
  -- of the data type.
  slurpTopLvlAnn :: [CommandLineOption] -> ModSummary
                 -> HsParsedModule -> Hsc HsParsedModule
  slurpTopLvlAnn _ _ hpm = let annDecls = findAnnDecls hpm in
    return $ hpm { hpm_ann_from_parser = annDecls }

  type StrippedAnnD = (AnnProvenance RdrName, HsExpr GhcPs)

```

```haskell
-- | Traverses the top level declarations in the module, finds
-- annotations and returns the annotated binding together with
-- the payload expression.
findAnnDecls :: HsParsedModule -> [StrippedAnnD]
findAnnDecls hpm = let L _ hsModule = hpm_module hpm in
  let lHsDecls = hsmodDecls hsModule in
    stripAndFilter lHsDecls

-- | Strips location wrappers and collects content from ANN
-- declarations from the top level declarations; we do not
-- traverse deeper into the AST.
--
-- HsDecl has a constructor 'AnnD (AnnDecl name)', where
-- 'AnnDecl name = HsAnnotation (AnnProvenance name)'
--                                '(Located (HsExpr name))'
stripAndFilter :: [Located (HsDecl GhcPs)] -> [StrippedAnnD]
stripAndFilter = foldr unwrapAnnD []
  where
    unwrapAnnD :: Located (HsDecl GhcPs) -> [StrippedAnnD]
               -> [StrippedAnnD]
    unwrapAnnD (L _ (AnnD (HsAnnotation _ annProv lHsExpr)))
               annDecls = let L _ hsExpr = lHsExpr in
                          (annProv, hsExpr):annDecls
    unwrapAnnD _ annDecls = annDecls
    -- ^ Not interested in anything else, so we skip any other
    -- declaration
```

Listing 56: Implementation of the compiler plugin. The function `findAnnDecls` traverses the top level declarations of a parsed module, strips away wrapping constructors and collects binder names and annotation payloads.

D.3.1 Packaging and registration of the plugin

Cabal is the standard package management system for Haskell. When the plugin builds, we need to make a Cabal package and register it with the system. This makes it possible for the `ghc` binary to find the plugin when it is requested for use in the compilation.

We first need a standard `.cabal` configuration file for a library package, this is explained well in the Cabal documentation[5]. Unfortunately, Cabal uses the system GHC installation to build our package, and since we need to use our patched compiler it cannot proceed. Luckily, there is a newly developed build system for Cabal called

[5]`https://www.haskell.org/cabal/users-guide/developing-packages.html#example-a-package-containing-a-simple-library`

new-build. This enables a plethora of new configuration options, including one to direct the build to use a particular path to a compiler executable.

We instruct Cabal to use our patched compiler with the with-compiler setting in the global configuration; it is currently not possible to do this on a per-package basis. This file usually resides in /.cabal/config.

```
[...]
with-compiler: /path/to/ghc/inplace/bin/ghc-stage2
[...]
```

Then we can configure, build and register our package in our system so that ghc can find it:

```
cd /path/to/PluginModule
cabal configure
cabal install
```

Now the plugin can be run in the way specified in Section D.1.

D.3.2 Limitations in the extended plugins proposal

The extended plugin proposal API was found to be too constraining to allow persisting data in the compiler state for use later in the pipeline. The 'parseResultAction' plugin seemed like a good fit for implementing the catching of weights and stashing them in the compiler state for later retrieval in ApplicativeDo. The entry point of the algorithm works in the rename monad, RnM:

```
rearrangeForApplicativeDo
  :: HsStmtContext Name
  -> [(ExprLStmt GhcRn, FreeVars)]
  -> RnM ([ExprLStmt GhcRn], FreeVars
```

RnM has access to the global type checking state, TcGblEnv, which in turn contains the compilation state, HscEnv. The issue is that HscEnv is read-only within parsedResultAction, and we do not have the TcGblEnv in scope. A possible workaround for the issue is to put the collected data in a new field of the returned HsParsedModule, but this needs to be moved to the TcGblEnv elsewhere in the compiler. This requires plugin-specific changes to the compiler internals, which makes it hard to argue for extracting the functionality to a plugin in the first place.

As stated in Section 3.5, these issues were raised with the GHC developers for insight to the development of the proposal at GitHub[6], and taken up for further discussion on Phabricator[7].

[6]`https://github.com/ghc-proposals/ghc-proposals/pull/107#issuecomment-380806941`

[7]https://phabricator.haskell.org/D4342#128964

E

Hacking on GHC

The Glasgow Haskell Compiler is a complex beast and setting up a development environment can be a bit tricky. Here we document a well working setup for upstream tracking source control, as well as setting up a properly optimised build environment locally. This was not entirely straightforward, and the information was deemed useful for future theses working on GHC.

E.1 Setup up a GHC fork with Git

To enable working collaboratively and version controlled on the GHC code base, we set up a Git fork of the upstream GHC repository. This way we can stay in sync with the main development, while still adding our improvements. This can be stored at any git host, but we chose to fork the GHC repository on GitHub[1] and created a new branch to work on.

Then, we clone the GHC repository mirror from GitHub in a directory of our choice:

```
$ git clone https://github.com/ghc/ghc
```

Because of different naming conventions in GHC and GitHub we need to give git some extra pointers on where to find certain files. This updates global Git settings and needs to be done only once.

```
$ git config --global url."git://github.com/ghc/packages-".insteadOf
    git://github.com/ghc/packages/
$ git config --global url."http://github.com/ghc/packages-".insteadOf
    http://github.com/ghc/packages/
$ git config --global url."https://github.com/ghc/packages-".insteadOf
    https://github.com/ghc/packages/
$ git config --global url."ssh://git\@github.com/ghc/packages-".insteadOf
```

[1] https://github.com/

```
    ssh://git\@github.com/ghc/packages/
$ git config --global url."git\@github.com:/ghc/packages-".insteadOf
    git\@github.com:/ghc/packages/
```

GHC has a large number of project dependencies linked as Git *submodules*[2]. When first setting up, we need to populate the project with files from these dependent projects with the `git submodule` command.

```
$ cd ghc    # The directory of our newly cloned compiler
$ git submodule update --init
```

Now the GHC code base is all set up, but we do not have write access to the official GHC repository. Hence, we need to add our own repository on GitHub as a remote for Git.

```
$ git remote add {git username}
    https://github.com/{git username}/ghc.git
$ git fetch {git username} {development branch name}
$ git checkout {development branch name}
```

If the `git status` command then indicates changes to files in the submodules, there may have been version changes to the dependencies upstream. This is fixed by running `git submodule update`, which does a checkout of the correct snapshot of the relevant submodules.

Now, our local repository knows about *both* our own repository, as well as the official GHC one. This means we can stay in sync with upstream while storing our own changes somewhere else.

E.1.1 Working with Phabricator Differentials

GHC uses a software development platform called Phabricator[3] for things like Git hosting, planning, continuous integration, and particularly code review. Contributions that are above trivial in complexity has to be through Differential[4], a system for code review, discussion, and merging of code contributions.

The issue with our Git setup is that it is based on a fork of the GHC GitHub mirror, while the project is really hosted with Phabricator. This poses no issues until we need to work with experimental branches from this platform, as is the case with the Extended Plugins Proposal discussed in Appendix D. This in-development branch

[2]`https://git-scm.com/book/en/v2/Git-Tools-Submodules`
[3]The Phabricator platform: `https://www.phacility.com/phabricator/`
[4]The Differential module: `https://www.phacility.com/phabricator/differential/`

is not officially merged, and is hence absent from the main development branches, and therefore not mirrored to GitHub. Phabricator uses a tool called Arcanist[5] to pull these differentials into the local development setup, but this requires working in a repository having Arcanist setup; this is not the case in the GitHub mirror.

Fortunately, all differentials on Phabricator are available as a raw Git diff, a file containing instructions for Git on how to apply the changes locally to a code base. It specifies which rows have which changes, for each file affected by the differential. When the file is downloaded, applying the diff is simple from a clean Git directory:

```
$ git apply D4342.diff
$ git commit -am "Applying raw diff of D4342"
```

Now our local project has the proposed changes from the differential, and we build the patched GHC as described in E.2 with no differences. If we do not want the changes anymore, for example, if they get merged to master or they don't do what we had hoped, we can undo the patch by finding the commit with `git log`, and revert the commit. This is easily done if the patch commit message mentions the differential ID since it will be easy to find:

```
$ git log --grep "D4342"
commit b0462e864f719d9c7b141cdbb5c2c5513a1c247d
    (HEAD -> improved-ado, m0ar/improved-ado)
Author: Edvard Hübinette <edvard@hubinette.me>
Date:   Mon Feb 19 11:18:14 2018 +0100

    Apply raw diff of D4342 (ExtendedPluginsProposal)

$ git revert b0462e864f719d9c7b141cdbb5c2c5513a1c247d
```

E.2 Building GHC

Now the code is in place, and we would like to compile our compiler. This is a time-consuming process, but we can take some shortcuts to get our building done faster. In the `ghc/mk` directory there is a sample build file, `build.mk.sample`, that can edited and saved to `build.mk`. Inside, there are a plethora of options for the build process. The recommended full build mode is `quick`, which builds everything, including dependencies but excluding profiling versions of everything, which is unnecessary at this point.

When this is done, we prepare the build.

[5]The Arcanist tool: `https://www.phacility.com/phabricator/arcanist/`

```
$ ./boot
$ ./configure
```

Then run `make` to start the build process. We can speed it up further by telling `make` about extra cores so that it can do some work concurrently.

```
$ make -j{\# physical cores + 1}
```

When this finishes, the usable compiler can be invoked. It can be used both as a normal compiler, but also as `GHCi` with the `-interactive` flag.

```
$ ghc/inplace/bin/ghc-stage2 file.hs
$ ghc/inplace/bin/ghc-stage2 --interactive file.hs
```

E.2.1 GHC stages

GHC is built in *stages*, and depending on what you are trying to achieve you do not have to build all of them; it is possible to prevent the build system from compiling stages later than the one you need in `mk/build.mk`. This is useful to ensure a quick under-development build cycle, only spending time on necessities. The notation of stages is somewhat confusing, and are therefore explained here.

Stage 0 is the system GHC. You use the stage 0 compiler to make your GHC source code into the stage 1 compiler. The stage 1 compiler has your added changes. However, it is not *built* with a compiler sporting your changes; the features you added to GHC have not been used to improve the compiler itself. Stage 1 does not support `GHCi` or Template Haskell (a way of doing compile-time meta-programming with Haskell).

When using the stage 1 compiler to build the same source code again, the result is the stage 2 compiler: your compiler compiled with itself. Next is stage 3, where stage 2 is used to build GHC once again to verify that it is functioning properly. This stage is what is used for running all GHC tests, verifying that changes did not break anything.

E.2.2 Further build configuration

In `mk/build.mk` there are several settings to lock the build process to stop after the required stage. For example, if we are working on features that do not need to be present when building GHC itself, we can enable the `devel2` build profile. This locks the stage 1 compiler, meaning a rebuild only compiles the second stage; your changes will be present when compiling programs, but the compiler itself will not be built with them.

If we want to use the GHC profiler, for example, to get information on actual weights on statements of a program, we need to add the `BUILD_PROFS_LIBS = YES` flag to the build configuration. This is so GHC builds the `base` library with profiling enabled, a necessity for profiling of other programs to work.