Desugaring Haskell's do-Notation into Applicative Operations

Simon Marlow
Facebook, UK
smarlow@fb.com

Simon Peyton Jones
Microsoft Research, UK
simonpj@microsoft.com

Edward Kmett
S&P Global, USA
ekmett@mhfi.com

Andrey Mokhov
Newcastle University, UK
andrey.mokhov@ncl.ac.uk

Abstract

`Monad`s have taken the world by storm, and are supported by **`do`**-notation (at least in Haskell). Programmers are increasingly waking up to the usefulness and ubiquity of `Applicative`s, but they have so far been hampered by the absence of supporting notation. In this paper we show how to re-use the very same **`do`**-notation to work for `Applicative`s as well, providing efficiency benefits for some types that are both `Monad` and `Applicative`, and syntactic convenience for those that are merely `Applicative`. The result is fully implemented as an optional extension in GHC, and is in use at Facebook to make it easy to write highly-parallel queries in a distributed system.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) programming; D.3.3 [*Language Constructs and Features*]: Control structures

Keywords Haskell; monad; applicative; syntax

1. Introduction

Consider this Haskell function that calculates the number of common friends between two Facebook users:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y = do
  fx ← friendsOf x
  fy ← friendsOf y
  return (length (intersect fx fy))
```

Here `friendsOf` is an operation that makes a remote query to a database to fetch the list of friends of a user. Desugaring the monadic **`do`** expression according to the Haskell standard [13] yields this:

```
numCommonFriends x y =
  friendsOf x >>= λfx →
  friendsOf y >>= λfy →
  return (length (intersect fx fy))
```

where `>>=` and `return` are operations from the `Monad` class. This translation works fine, but it is inherently *sequential*: the second call to `friendsOf` cannot start until the first returns, because the result of the first call, namely `fx`, is in scope at the second call so in principle might be used by it. But, tantalisingly, `fx` manifestly *isn't* used by the second call, so we actually *could* run the two in parallel.

Marlow et. al. [14] showed how to exploit this parallelism by using McBride and Paterson's insight that between a `Functor` and a `Monad` lies an `Applicative` [16]. To be concrete, we can rewrite `numCommonFriends` using `Applicative` combinators like this:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y =
  (λfx fy → length (intersect fx fy))
    <$> friendsOf x
    <*> friendsOf y
```

The combinators `<$>` and `<*>` are defined in Figure 1, but for now we simply note that the two calls to `friendsOf` are now manifestly independent of one another. And indeed the implementation of the `Haxl` monad[1] *can* take advantage of that independence to perform the two `friendsOf` queries in parallel; in fact it collects them together and batches them into a single query.

But there is still a problem; programmers should not have to spot where they can use `<*>` to gain its advantages, because they are likely to miss some opportunities, especially when code is refactored. Moreover there are maintainability and comprehensibility benefits in using a single universal notation, namely **`do`** notation. In this paper we show how to have our cake and eat it too: the programmer writes **`do`** notation, and the compiler desugars it automatically into the efficient parallel code that uses `Applicative` combinators. We make these contributions:

- Rather than desugaring **`do`** notation uniformly into `Monad` combinators, we show how to take advantage of the program's dependency structure to selectively use `Applicative` combinators instead (Section 2.1). For some types that are both `Monad` and `Applicative`, this provides efficiency benefits at runtime without losing any maintainability or clarity in the source code. For types that are `Applicative` but not `Monad`, we gain access to the **`do`** notation, providing a syntactic convenience.
- The more we can use `Applicative` combinators, the better. But as we show in Section 2.4, there may be more than one way to desugar a **`do`**-expression into `Applicative` combinators, none of which is universally best. We propose a definition of optimality by fixing a set of assumptions.
- We present a detailed translation of Haskell's do-notation into `Applicative` operations (Section 3) using our definition of optimality (Section 4). This translation proceeds by way of an independently-interesting elaboration of the **`do`**-notation.
- We present an implementation of the described translation in the Glasgow Haskell Compiler (Section 5), and measure its effectiveness on existing widely-used open-source Haskell code, and a large codebase at scale.
- The `Haxl` monad is not the only abstraction where using `Applicative` combinators leads to more efficient code than the equivalent expression written using `Monad` combinators. We give some more examples in Section 6.

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

```
class Functor f where
  fmap :: (a → b) → f a → f b

class Functor f => Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b

class Applicative f => Monad f where
  return :: a → f a
  (>>=)  :: f a → (a → f b) → f b

<$>      :: Functor f => (a → b) → f a → f b
<$>      =  fmap

ap       :: (Monad m) => m (a → b) → m a → m b
ap mf mx =  mf >>= λf → mx >>= λx → return (f x)

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

Laws used in this paper

```
                  f <$> m   =   pure f <*> m
                      <*>   =   ap
                    pure    =   return
               pure r >>= f   =   f r
     m >>= (λx → k x >>= h)   =   (m >>= k) >>= h
```

Figure 1. Definitions of `Functor`, `Applicative`, `Monad`, `<$>`, `ap`, and `join`

2. The Main Idea

In 1992 Wadler suggested using *monads* as a programming abstraction [22], conveniently embodied as a type class `Monad` in Haskell. Monads took the world by storm, and have appeared in many other languages.

Sixteen years later, McBride and Paterson discovered another key abstraction, which they called *applicative functors* [16], embodied by the `Applicative` type class. The `Applicative` class sits between `Functor` and `Monad` in the class hierarchy; every `Monad` is an `Applicative` and every `Applicative` is a `Functor`, but the reverse of these is not necessarily true. Figure 1 gives the definitions of these classes for easy reference.

This paper is based on two observations. Firstly, it would be convenient to be able to write `Applicative` expressions using **`do`** notation. For example, given an effectful map written using **`do`** notation like this:

```
mapM []     = pure []
mapM (x:xs) = do  x' ← f x
                  xs' ← mapM f xs
                  pure (x' : xs')
```

we would like GHC to infer this type and desugaring for it:

```
mapM :: Applicative m => (a → m b) → [a] → m [b]
mapM []     = pure []
mapM (x:xs) = (:) <$> f x <*> mapM f xs
```

Notice that, despite the use of **`do`**-notation, the inferred type indicates that `mapM` works for any `Applicative`, not just for any `Monad`, and so will work for a significantly wider range of types.

The second, and more important, observation is that in some `Monad`s the `Applicative` `<*>` operation is more efficient than the equivalent `Monad` `ap` operation. Exploiting this performance difference currently requires the programmer to spot where they can use `<*>` and refactor their code to use it, but, like other compiler optimisations, we would prefer the compiler to automatically take advantage of `<*>` whenever it can. The approach we take is to have the compiler desugar **`do`** notation into uses of the `Applicative` operations where possible, falling back to `Monad` when necessary.

This second observation is the strongest motivator for this work: the `Haxl` monad (called `Fetch` in previous work [14]) provides parallelism between data-fetching operations when the `<*>` operator is used. But programmers should not have to think about where to use `<*>`. Indeed, we would prefer not to use `<*>` explicitly in our code at all, because it is sensitive to refactoring: introducing or removing dependencies between expressions affects where we can use `<*>`, and if the programmer is responsible for using `<*>` then not only do they have to spend time thinking about it, but they are likely to do an imperfect job. Thus we would like programmers to be able to use a simple universal syntax, so that they can focus on correctness while letting the compiler exploit parallelism as far as possible. The translation we present in this paper achieves this: `Haxl` programmers use **`do`** notation, and the compiler automatically extracts the available parallelism. This translation is used in a system at Facebook, and results in significant performance gains (Section 5.5).

2.1 The Challenge

The challenge is this: given an arbitrary expression in **`do`**-notation, we would like to translate it into an expression that, wherever possible, uses operations from the `Applicative` class rather than the `Monad` class.

For reference, the definitions[2] of the `Functor`, `Monad`, and `Applicative` type classes as provided in GHC 8.0.1[3] are given in Figure 1, along with the auxiliary functions `<$>` (an infix spelling of `fmap`), and `ap`. The Figure also gives the laws that are expected to hold for instances of `Monad` and `Applicative`. For example, in many monads `<*>` is defined to be `ap`; but even where it has a more efficient implementation the second law says that its semantics should be the same as `ap`. Nothing enforces these laws, *but our alternative desugaring is only semantics-preserving if these laws hold* for the relevant instances of `Functor`, `Applicative`, and `Monad`.

Before we give the translation scheme in full in Section 3, we will motivate our design through a series of examples. First, a straightforward example involving two independent statements:

```
do x1 ← A
   x2 ← B
   return (x1,x2)
```

where `A` and `B` are arbitrary expressions, and `B` does not mention `x1`. The normal desugaring of this expression, according to the Haskell 2010 Report, would yield this expression:

```
A >>= λx1 →
B >>= λx2 →
return (x1,x2)
```

Using `<$>` and `<*>` instead gives us:

```
(,) <$> A <*> B
```

This is semantically equivalent, as you can check for yourself using the laws given in Figure 1, plus the definition of `ap`.

[2] For simplicity we have omitted default definitions, and the operators `$>`, `>>`, `*>`, and `<*`.

[3] After extensive user debate, GHC has diverged from the Haskell 2010 specification by adding the new class `Applicative` as a superclass of `Monad`.

Next, let us modify the original expression so that the expression `B` mentions the variable `x1`:

```
do x1 ← A
   x2 ← B[x1]       -- An expression B mentioning x1
   return (x1,x2)
```

There is no way to desugar this expression into a use of the `<*>` operator as before, because there is now a *dependency* between `B` and `A`. We can see that from the types of `<*>` and `>>=`:

```
(<*>) :: Applicative f => f (a → b) → f a       → f b
(>>=) :: Monad f       => f a       → (a → f b) → f b
```

The type of `>>=` allows the second computation `(f b)` to depend on the result `a` of the first, whereas `<*>` does not. This is the essence of the difference between `Monad` and `Applicative`; `Monad` allows dependencies on previous results, whereas `Applicative` does not. So we must desugar the example to

```
A >>= λx1 → B[x1] >>= λx2 → return (x1,x2)
```

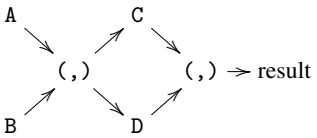
In short: whenever there is a dependency between two statements in a **`do`**-notation expression, our translation must use `>>=` somewhere.

2.2 Mixing it Up

However, it's not an either/or choice: we may be able to desugar in a way that uses `<*>` in some places and `>>=` in others. For example:

```
do x1 ← A
   x2 ← B
   x3 ← C[x1]
   x4 ← D[x2]
   return (x3,x4)
```

Here we have two pairs of statements, `A` and `B`, and `C` and `D`. The statements in each pair are independent, but `C` and `D` depend on the results of `A` and `B` respectively. So we can do `A` and `B` applicatively in parallel, gather the results with `>>=`, and then do `C` and `D` in parallel. Here's a picture to show what we mean:



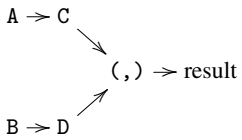
and we use the informal notation $(\texttt{A}|\texttt{B});(\texttt{C}|\texttt{D})$ to describe this structure. We can rewrite the expression using applicative combinators, following this structure, as follows:

```
((,) <$> A <*> B)
   >>=
λ(x1,x2) → (,) <$> C[x1] <*> D[x2]
```

The first line does `A` and `B` in parallel, building a result pair `(x1,x2)`; then comes a monadic bind; then we match the pair and do `C` and `D` in parallel. The important point is that we use the applicative `<*>` where possible, and the monadic `>>=` where necessary.

2.3 Accounting for Effects

Looking again at the example in the previous section, there is an alternative execution plan that would respect the data dependencies:



or, in our informal notation $(\texttt{A};\texttt{C}) \mid (\texttt{B};\texttt{D})$. After all, the data dependencies only require that `C` occurs after `A`, and `D` after `B`. Moreover, this appears to be a *better* plan than the one in Section 2.2, because it removes an apparently-unnecessary synchronisation point. To see why it is better, suppose `A` and `D` take two seconds each and `B` and `C` both take one second. Then the above plan takes three seconds, but the one in Section 2.2 takes four.

Alas we cannot use this more efficient execution plan, though, because it amounts to swapping the order of `B` and `C`. The corresponding applicative expression is this:

```
(,) <$> (A >>= λx1 → C[x1])
    <*> (B >>= λx2 → D[x2])
```

but this is *not* semantically equivalent to the original **`do`**-notation expression. Imagine executing it under a `State` monad, for example: the effects would appear in the order `A`, `C`, `B`, `D`, and the program may give different results.

Reordering the statements is only valid in a *commutative* monad, where the order of effects is not observable. The `Haxl` monad is not commutative, because it supports effects in the form of exceptions, so reordering statements can change which exceptions are reported. In our design, we therefore never reorder computations. We leave for future work the possibility of allowing reordering for commutative monads.

Even though our transformation does no automatic re-ordering, the programmer is free to do so manually, by writing:

```
do x1 ← A
   x3 ← C[x1]
   x2 ← B
   x4 ← D[x2]
   return (x3,x4)
```

Now our transformation will be able to produce the more efficient result.

2.4 There Is No Single Best Translation

Consider this example:

```
do x ← A
   y ← B
   z ← C[x]
   return (y+z)
```

There are two ways that we might consider implementing this, in our informal notation:

1. $(\texttt{A} \mid \texttt{B});\texttt{C}$
2. $\texttt{A};(\texttt{B} \mid \texttt{C})$

Which one is better? Alas, it depends on the relative execution times of `A`, `B`, and `C`. Imagine a parallel execution model where we can determine the overall execution time (which we will call "cost") by interpreting "|" as maximum and ";" as addition. so the cost of (1) is $max(\texttt{A},\texttt{B}) + \texttt{C}$, and the cost of (2) is $\texttt{A} + max(\texttt{B},\texttt{C})$. Now let's assign some example costs to `A`, `B`, `C`:

- $\texttt{A} = 1, \texttt{B} = 1, \texttt{C} = 1$: both alternatives have equal cost, 2.
- $\texttt{A} = 0, \texttt{B} = 1, \texttt{C} = 1$: (1) has cost 2, (2) has cost 1.
- $\texttt{A} = 1, \texttt{B} = 1, \texttt{C} = 0$: (1) has cost 1, (2) has cost 2.

It is easy to see that which translation is better depends on the relative cost of the terms.

We cannot have complete knowledge of the costs of the statements in a **`do`**, therefore it is not possible to find an optimal translation in general. Our scheme uses a conservative definition of "optimal" wherein each statement is assumed to have equal cost.

Expressions

$e \in Expr$	::=	v	Variable
	|	$e_1\ e_2$	
	|	$\lambda p \to e$	
	|	$(e_1, \ldots, e_n)$	$n >= 2$
	|	**do** $l\ e$	
	|	$\ldots$	

Patterns

$p \in Pat$	::=	v	
	|	$(p_1, \ldots, p_n)$	$n >= 2$
	|	$\ldots$	

Statement sequences

$l \in Stmts$	::=	$\{s_1; \ldots; s_n\}$	$n >= 1$

Statements

$s \in Stmt$	::=	$(l_1 \mid ... \mid l_n)$	$n >= 2$
	|	$p \leftarrow e$	
	|	e	
	|	**let** *bind* **in** e	

Figure 2. Syntax

$$\mathsf{desugar}_{\mathsf{std}}\ (\texttt{do}\ \{e\}) = e$$
$$\mathsf{desugar}_{\mathsf{std}}\ (\texttt{do}\ \{p \leftarrow e; s\}) = e\ \texttt{>>=}\ \lambda p \to \mathsf{desugar}_{\mathsf{std}}\ (\texttt{do}\ \{s\})$$

Figure 3. Haskell 2010 desugaring of `do`-notation

2.5 Optimising the Translation

In Section 2.2 we built a pair of results from `A` and `B`, used `>>=`, and pattern-matched the resulting pair. Here is an alternative and neater translation using the `join` combinator (see Figure 1):

```
join ((λx1 x2 → (,) <$> C[x1] <*> D[x2])
        <$> A
        <*> B)
```

By using `join` we avoid the intermediate pair `(x1,x2)`. One should think of `join` as a more flexible `>>=`, and in fact in our translation we shall be using `join` instead of `>>=` in this way.

3. The New Desugaring Algorithm

In this section we formalise our new desugaring algorithm for **`do`** notation. It proceeds in two stages:

- *Rearrangement* (Section 3.2). The first stage corresponds to our informal execution plan. It takes a sequence of statements $s_1, \ldots, s_n$ and groups them into *parallel blocks* (Section 3.1), thus building a tree. Rearrangement does not re-order the statements, merely groups them; flattening the tree returns the original statement sequence.
- *Desugaring* (Section 3.3). The second stage turns this tree of statements into an expression using `<*>`, `<$>`, `>>=`, and `join`.

Before presenting rearrangement and desugaring in detail, we first present an extended syntax for **`do`**-notation in Section 3.1. This syntax serves as a bridge between the two stages of the algorithm, capable of expressing the choices made by rearrangement without the noise introduced by desugaring.

For comparison, the standard Haskell 2010 desugaring for **`do`** expressions is given in Figure 3 (using the Haskell Report's abstract syntax which does not distinguish the final `return`, unlike ours). For simplicity we ignore refutable patterns for now, but we return to them in Section 3.7.

3.1 Parallel Blocks

In Section 2 we used an informal notation `(A | B); C` to describe our desired execution plan, and used that plan to desugar the **`do`** expression. In this section we formalise that notation as a simple and independently-useful extension of **`do`**-syntax.

Figure 2 gives the new (abstract) syntax. Note that:

- For expressions and patterns we omit everything except the forms we use in our translation; hence the "...".
- In our abstract syntax, an expression **`do`** $l\ e$ represents a **`do`** expression with statements l that ends in `return` e or `pure` e. If the original source **`do`** expression does not end in `return` e or `pure` e then we can transform it so that it does, by introducing a dummy variable. For example, **`do`** `{ x ← A; B }` would be represented as **`do`**$\{x \leftarrow$ `A`$; y \leftarrow$ `B`$\}\ y$ in our abstract syntax, where y is a fresh variable.
- A statement s is either a single statement (bind, expression, or **`let`**), or it is a *parallel block* $(l_1 \mid \ldots \mid l_n)$, where each l_i is again a sequence of statements.

These parallel blocks are not written by the programmer; rather, they are introduced by our rearrangement algorithm. Their meaning is simple: a block $(l_1 \mid \ldots \mid l_n)$ means the same as the statement sequence $l_1 +\!\!+ \ldots +\!\!+ l_n$, where $+\!\!+$ appends two sequences of statements. Thus, for example, these two mean the same thing:

```
do (a ← A | b ← B)          do a ← A
   c ← C                       b ← B
   (d ← D | e ← E)             c ← C
                               d ← D
                               e ← E
```

In short, flattening all parallel blocks does not change the meaning of the program.

As the syntax suggests, though, a parallel block requires that no result computed by l_i is required by any of the other blocks l_j. This is enforced by a simple scoping limitation: the variables bound in l_i are not in scope in l_j when $i \neq j$. In the above example, `a` is not in scope in `B`, nor vice versa. Similarly, `a`, `b`, and `c` are all in scope in `D` but `e` is not.

The independence of the l_i in a parallel block means that the desugaring algorithm is free to combine them with `Applicative` combinators. To be concrete, the parallel block $(l_1 \mid \ldots \mid l_n)$ is equivalent to the statement

$$(p_1, \ldots, p_n) \leftarrow (, \ldots,)\ \texttt{<\$>}\ \texttt{do}\ l_1\ p_1\ \texttt{<*>}\ \ldots\ \texttt{<*>}\ \texttt{do}\ l_n\ p_n$$

where $p_i = \mathsf{tuple}\ \mathsf{bv}(l_i)$ and $\mathsf{bv}(l_i)$ are the variables bound by l_i. By `<*> = ap`, this interpretation is equivalent to flattening the parallel block into a sequence.

3.2 Rearrangement

The algorithm for rearrangement is given in Figure 4. The function `rearrange` applies to the sequence of statements l in a **`do`** expression which contains no parallel forms, and it returns a new sequence in which the parallel form is used "as much as possible" (we will formalise this in Section 4). Let us consider this example:

```
do x1 ← A
   x2 ← B[x1]
   x3 ← C
   return (x2,x3)
```

Rearrangement ignores the final expression, `return (x2,x3)` in this case, and considers only the list of statements. The first step is

$$
\begin{array}{lll}
\mathsf{rearrange}\ \{s_1;\ldots;s_n\} & = \{s_1\}, & \text{if } n = 1 \\
& = \mathsf{split}\ g_1, & \text{if } k = 1 \\
& = \{(\mathsf{split}\ g_1 \mid \ldots \mid \mathsf{split}\ g_k)\}, & \text{otherwise}
\end{array}
$$

where

$g_1 \ldots g_k = \mathsf{segments}\ \{s_1;\ldots;s_n\}$

$$
\mathsf{segments}\ \{s_1;\ldots;s_n\} = \{s_1;\ldots;s_{i_1}\}\ldots\{s_{(i_k)+1};\ldots;s_n\}
$$

where

$$
i_1 \ldots i_k = \{\ i \in 1\ldots n \mid \mathsf{bv}\{s_1;\ldots;s_i\} \cap \mathsf{fv}\{s_{i+1};\ldots;s_n\} = \emptyset\ \}
$$

$$
\begin{array}{lll}
\mathsf{split}\{s_1;\ldots;s_n\} & = \{s_1\}, & \text{if } n = 1 \\
& = \mathsf{splitat}\ i_{opt}, & \text{otherwise}
\end{array}
$$

where

$\mathsf{splitat}\ i = \mathsf{rearrange}\ \{s_1;\ldots;s_i\} +\!\!+ \mathsf{rearrange}\ \{s_{i+1};\ldots;s_n\}$

$i_{opt} \in 1\ldots n$ **such that**

$\forall j\,.\,1 \le j < n.\ \mathsf{cost}_s\ (\mathsf{splitat}\ j) \ge \mathsf{cost}_s\ (\mathsf{splitat}\ i_{opt})$

$$
\mathsf{cost}_s\ \{s_1;\ldots;s_n\} = \Sigma\{\mathsf{cost}_a s_i \mid 1 \le i \le n\}
$$

$$
\begin{array}{ll}
\mathsf{cost}_a\ (\mathtt{p} \leftarrow \mathtt{e}) & = 1 \\
\mathsf{cost}_a\ (l_1 \mid \ldots \mid l_n) & = \mathsf{max}\{\mathsf{cost}_s\ l_i \mid 1 \le i \le n\}
\end{array}
$$

$\mathsf{fv}\ \{s_1;\ldots;s_n\} =$ *the free variables of* $s_1 \ldots s_n$

$\mathsf{bv}\ \{s_1;\ldots;s_n\} =$ *the variables bound by* $s_1 \ldots s_n$

Figure 4. Rearrangement: introduce parallel statements

$$
\mathsf{desugar}\ ::\ \mathit{Stmts} \to \mathit{Expr} \to \mathit{Expr}
$$

$$
\mathsf{desugar}\ \{\}\ e = \mathtt{pure}\ e \qquad \textit{(0)}
$$

$$
\begin{array}{ll}
\mathsf{desugar}\ \{p \leftarrow e\}\ e' & \textit{(1)} \\
\quad \mid p == e' & = e \\
\quad \mid \text{otherwise} & = (\lambda p \to e')\ \texttt{<\$>}\ e
\end{array}
$$

$$
\mathsf{desugar}\ \{p \leftarrow e;\ l\}\ e' = e\ \texttt{>>=}\ \lambda p \to \mathsf{desugar}\ l\ e' \qquad \textit{(2)}
$$

$$
\begin{array}{ll}
\mathsf{desugar}\ \{(l_1 \mid \ldots \mid l_n)\}\ e & \textit{(3)} \\
\quad = (\lambda p_1 \ldots p_n \to e)\ \texttt{<\$>}\ e_1\ \texttt{<*>}\ldots\texttt{<*>}\ e_n & \\
\quad \textbf{where}\ (p_i, e_i) = \mathsf{desugar}_{arg}\ l_i\ \mathsf{fv}(e) &
\end{array}
$$

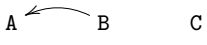
$$
\begin{array}{ll}
\mathsf{desugar}\ \{(l_1 \mid \ldots \mid l_n); s\}\ e & \textit{(4)} \\
\quad = \quad \mathtt{join}\ ((\lambda p_1 \ldots p_n \to e')\ \texttt{<\$>}\ e_1\ \texttt{<*>}\ldots\texttt{<*>}\ e_n) & \\
\quad \textbf{where} & \\
\qquad e' = \mathsf{desugar}\ s\ e & \\
\qquad (p_i, e_i) = \mathsf{desugar}_{arg}\ l_i\ \mathsf{fv}(e') &
\end{array}
$$

$$
\begin{array}{l}
\mathsf{desugar}_{arg}\ ::\ \mathit{Stmts} \to \mathit{Set}\ \mathit{Var} \to (\mathit{Pat}, \mathit{Expr}) \\
\mathsf{desugar}_{arg}\ \{p \leftarrow e\}\ vs = (p, e) \\
\mathsf{desugar}_{arg}\ l\ vs = ((v_1, .., v_k), \mathsf{desugar}\ l\ (v_1, ..., v_k)) \\
\quad \textbf{where} \\
\quad v_1 \ldots v_k = \mathsf{bv}(l)\ \cap\ vs
\end{array}
$$

Figure 5. Desugaring

to split the list into *segments*, as defined by the segments function in Figure 4. We define segments according to where their boundaries are: there is a segment boundary after statement i in the sequence whenever none of the variables defined by statements $s_1 \ldots s_i$ are used in the following statements, $s_{i+1} \ldots s_n$. Intuitively, we are looking for the places in the sequence that have no dependencies crossing them, which are exactly the places we can split the sequence to use the applicative <*> operator.

The dependencies of our example expression look like this:



A segment boundary is a place in the sequence that has no arrows crossing it. In our case there is only one such place: between the statements B and C. From the definition of rearrange this gives

$$
(\mathsf{split}\ \{\mathtt{x1} \leftarrow \mathtt{A}; \mathtt{x2} \leftarrow \mathtt{B[x1]}\} \mid \mathsf{split}\ \{\mathtt{x3} \leftarrow \mathtt{C}\})
$$

Next, split deals with a single segment. By the definition of segments we cannot split this segment into independent subsegments, so we have no alternative but to divide it into two subsequences and combine them with ";". The question is, at which point should we divide the sequence? There is no way to tell locally which is the best spot to split it, so we exhaustively test the possibilities and pick the best (or one of the best, since there might be more than one). Alternatives are evaluated using a simple cost function, which assumes a parallel execution model in which each statement has unit cost. Note that there is a more efficient implementation of this algorithm that we discuss in Section 4.3.

In our example, there is only one choice for the split boundary in the left segment, and the right segment has a single statement so is returned by split unchanged. Both recursive calls to rearrange are on single statements, which return the statement unchanged, leaving the final result:

$$
(\{\mathtt{x1} \leftarrow \mathtt{A}; \mathtt{x2} \leftarrow \mathtt{B[x1]}\} \mid \{\mathtt{x3} \leftarrow \mathtt{C}\})
$$

In Section 3.4 we will consider a more complex example where the search for an optimal split point in split comes into play.

3.3 Desugaring

The next stage is desugaring, where we turn our tree of statements into a concrete expression, using the operators from the Applicative and Monad classes.

Figure 5 gives the desugaring for a rearranged **do** expression. For an expression **do** l e, the call $(\mathsf{desugar}\ \ l\ \ e)$, produces an equivalent expression that does not use **do** at the outer level. In the call $(\mathsf{desugar}\ \ l\ \ e)$ we will call e the *continuation*; it is the expression that forms the return value after l has performed its effects and bound any variables mentioned in e.

There are five cases in desugar:

(0) handles an empty list of statements;

(1) translates a singleton bind, using <$>;

(2) handles the general case for bind, using >>=;

(3) translates a singleton parallel block, by building an applicative expression;

(4) handles the general case of an applicative block that is not the last statement. In this case we build an applicative expression with join.

The two cases that explicitly match on a singleton statement, (1) and (3), are required for building expressions that require only Functor or Applicative respectively. Without these two rules, desugar would still produce a valid result, but it would require a Monad constraint in some cases where one is unnecessary.

Our running example will help to illustrate the process of desugaring. Applying desugar to the expression after rearrangement:

$$
\mathsf{desugar}\{\{\mathtt{x1} \leftarrow \mathtt{A}; \mathtt{x2} \leftarrow \mathtt{B[x1]}\} \mid \{\mathtt{x3} \leftarrow \mathtt{C}\}\}\ (\mathtt{x2}, \mathtt{x3})
$$

requires rule (3), yielding the applicative expression

```
(λx2 x3 → (x2,x3))
   <$> desugar {x1 ← A; x2 ← B[x1]} x2
   <*> desugar {x3 ← C} x3
```

Each element of the parallel composition l_i becomes an argument of the applicative expression. For each l_i, the function $\mathsf{desugar}_{arg}$ returns a pair of (a) the pattern to use in the lambda, and (b) the expression to use in the argument position. For the pattern, we form a tuple of the variables that are both defined by l_i and used in the continuation. In our example, the first argument defines both `x1` and `x2`, but of these only `x2` is used in the continuation `(x2,x3)`, so the pattern becomes `x2` (a tuple of one term is the term itself). The second argument defines only `x3`, so that becomes the pattern. The function on the left of `<$>` is a lambda expression with patterns for each of the arguments (here `x2` and `x3`), and the body of the lambda is the continuation.

In the arguments of the applicative expression we now have recursive calls to desugar, so let's consider the first of those:

```
desugar {x1 ← A; x2 ← B[x1]} x2
```

This requires case (2), yielding

```
A >>= λx1 → desugar {x2 ← B[x1]} x2
```

Next, the inner desugar call hits rule (1), specifically the first case since `x2` = `x2`, yielding just `B[x1]`. This special case of rule (1) avoids leaving an unnecessary call to `<$>` in the output, which might be difficult to optimise away later.

The other recursive desugar call reduces in a similar way, leading to this overall result:

```
(λx2 x3 → (x2,x3))
   <$> (A >>= λx1 → B[x1])
   <*> C
```

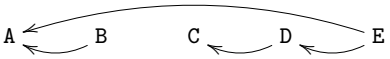
which is exactly what we wanted.

3.4 A Larger Example

Here is a more complicated example:

```
do x1 ← A
   x2 ← B[x1]
   x3 ← C
   x4 ← D[x3]
   x5 ← E[x1,x4]
   return (x2,x4,x5)
```

The statements have this dependency structure:



As before, we apply segments first. This time there are no segment boundaries, because the dependency from `E` to `A` spans the whole sequence. Thus we have a single segment, and we proceed to split.

In split, we must try all possibilities for a split and determine their costs. The four possibilities are enumerated below. For conciseness in the following discussion we will refer to the statements by their right hand sides (`A`, `B`, etc.):

1. Split after `A`, giving `A`; rearrange{`B`; `C`; `D`; `E`}. There are segments `B` and {`C`; `D`; `E`}, giving the final result `A`; (`B` | {`C`; `D`; `E`}) (cost 4).
2. Split after `B`, giving rearrange{`A`; `B`}; rearrange{`C`; `D`; `E`}, which reduces to `A`; `B`; `C`; `D`; `E` (cost 5).
3. Split after `C`, giving rearrange{`A`; `B`; `C`}; rearrange{`D`; `E`}. There is a segment boundary on the left after `B`, and we end up with ({`A`; `B`}|`C`); `D`; `E` (cost 4).
4. Split after `D`, giving rearrange{`A`; `B`; `C`; `D`}; `E`. There is a segment boundary after `B`, and the final result is ({`A`; `B`} | {`C`; `D`}); `E` (cost 3).

The rearrangement with the minimum cost was to split between `D` and `E`, which allowed us to put the two subsequences `A;B` and `C;D` in parallel with each other.

The full result of rearrangement is

```
({x1 ← A; x2 ← B[x1]} | {x3 ← C; x4 ← D[x3]});
{x5 ← E[x1,x4] }
```

Applying desugar results in:

```
join (λ(x1,x2) x4 →
      E[x1,x4] >>= λx5 → pure (x2,x4,x5))
  <$> (A >>= λx1 → B[x1] >>= λx2 → return (x1,x2))
  <$> (C >>= λx3 → D[x3])
```

Note that we determined that only `x4` needed to be returned from the sequence {`x3` ← `C`; `x4` ← `D[x3]`}, because $\mathsf{desugar}_{arg}$ takes the intersection of the variables defined by the sequence (`x3` and `x4` in this case) with the variables used in the continuation (`x2`, `x4`, and `x5`), which here is the singleton set containing `x4`.

3.5 `do` Expressions that Require `Functor` Only

A pleasant consequence of rule (1) in Figure 5 is that a simple **`do`** expression such as

```
do x ← ask
   return (filter (==x) list)
```

desugars to

```
(λx → filter (==x) list) <$> ask
```

This is a degenerate case of constructing an applicative expression, where we have only a single argument. With two or more independent statements the expression would use `<*>` and hence require an `Applicative` constraint, but here since we only use `<$>` the expression requires only `Functor`.

Interestingly, this translation may be more efficient than the standard Haskell desugaring, because `<$>` often has a more direct implementation than the combination of `>>=` and `return`. For example, consider the `Functor` and `Monad` definitions for lists:

```
instance Functor [] where
    fmap = map

instance Monad [] where
    return x = [x]
    xs >>= f = [y | x ← xs, y ← f x]
```

So `m >>= return . f` involves creating an intermediate singleton list `[x]` which is immediately deconstructed by `>>=`, whereas `fmap f m` does not have this intermediate singleton.

Note that by virtue of rules (1) and (3), *every* **`do`**-notation expression that ends with `return` or `pure` will be translated using `<$>`, effectively doing a little optimisation during desugaring, and leaving the optimiser with a little less work to do later.

3.6 `return` and `pure`

Our algorithm treats `pure` and `return` identically when they appear as the last statement of a **`do`** (see Section 3.1), and generates code that uses only `pure`. The latter is necessary so that we can generate code that requires only `Applicative` rather than `Monad`.

This might be surprising, because `do { return E }` turns into `pure E`. However, `return` is arguably a historical legacy, born before the discovery of applicative functors, and nowadays we should really be using `pure`. Indeed, there are those who argue that `return` should be removed from the `Monad` class and given the static definition `return = pure`.

A shortcoming of our design is that the check for `return` or `pure` is purely syntactic, and is easily defeated. For example, even `return $ x` or `let p = pure in p x` are not recognised. It seems hard to avoid this difficulty, given the constraints of fitting into an existing language design.

3.7 Refutable Patterns

A *refutable* pattern is one which may fail to match at runtime. Variables and patterns that match a single-constructor datatype (such as tuples) are irrefutable, because they cannot fail to match; patterns that refer to one constructor of a sum type (such as `x:xs`) are refutable.

The desugaring translation in Figure 5 needs an extra rule to handle refutable patterns:

```
desugar(do {p ← e; l} e')        (0.5) Handle refutable patterns
  | refutable p =
  let ok p = desugar l e'
      ok _ = fail "..."
  in
  e >>= ok
```

This rule takes precedence over Rules (1) and (2) when the pattern p is refutable. In particular this means that we cannot use `<$>` when the pattern is refutable, so a refutable pattern will entail a `Monad` constraint. Furthermore, future changes to Haskell are expected to remove `fail` from the `Monad` class into a separate `MonadFail` class, so this rule will result in a `MonadFail` constraint.

There is one more modification we need. The first clause of $\mathsf{desugar}_{arg}$ only applies when p is irrefutable:

$$\mathsf{desugar}_{arg}\ \{p \leftarrow e\}\ vs \mid \mathsf{not}(\mathsf{refutable}\ p) = (p, e)$$

and we fall back to the second clause, which will use Rule (0.5) above.

3.8 Extension to Other Statement Forms

Haskell's **do** notation has two additional statement forms that we have not dealt with yet: **let** statements and expression statements (Figure 2).

The **let** form is dealt with straightforwardly. First, the cost function treats a **let** as having zero cost:

$$\mathsf{cost}_a\ (\textbf{let}\ \mathit{decls}) = 0$$

A **let** should have zero cost because it can only do pure computation, and the goal of our translation is to achieve the maximal parallelisation of effects. Second, we must add a case to desugar:

```
desugar {let decls; l} e =        (5) Handle let
  let decls in desugar l e'
```

In our implementation we add one small refinement. We observe that there is no benefit in having **let** bindings placed in parallel with other statements, so in the result of segments if we have any segments that consist only of **let** bindings, we concatenate those bindings onto an adjacent segment. This results in slightly shorter desugared code with no loss in parallelism.

The expression form can be dealt with in two ways. The easiest way is to translate it into a bind statement with a wildcard pattern: `_ ← e`. That works, and yields the optimal parallelism, but it may be possible to achieve better efficiency in some cases (see Section 8).

3.9 Pitfalls

We encountered two related pitfalls when applying this translation to real code. In Haskell today it is possible to define `fmap` using **do** syntax, like this:

```
instance Functor T where
  fmap f m = do x ← m; return (f x)
```

If we apply our applicative desugaring this becomes

```
instance Functor T where
  fmap f m = f <$> m
```

and since `<$> = fmap`, the definition is now a loop. The fix is to define `fmap` without using **do**, as `fmap f m = m >>= return . f`.

A similar problem arises with `Applicative` instances:

```
instance Applicative T where
  mf <*> mx = do f ← mf; x ← mx; return (f x)
```

which turns into a self-recursive definition of `<*>`. The solution is to use `<*> = ap` (and ensure that the definition of `ap` itself does not fall into this trap!).

3.10 Expressing Applicatives Directly

Since the rearrangement algorithm of Section 3.2 is somewhat complex, one might worry that a minor change to the program might cause a different rearrangement, which in turn had very different behaviour (e.g. parallelism). In this respect applicative **do**-notation behaves like other compiler optimisations—moving the responsibility for optimisation from the programmer to the compiler is always a double-edged sword.

The usual solution is to allow the programmer to take control when they need to. Here, this means writing applicative code directly, which is certainly possible. However, an attractive alternative is to do *manual* rather than *automatic* rearrangement, by allowing the the programmer to write the rearranged program directly in the notation of Figure 2. That way, she can express the parallel structure in a high-level way, while the desugaring algorithm of Section 3.3 takes care of the tiresome plumbing. The syntactic extension is modest: just the parallel form of $Stmt$ in Figure 2. We have not yet tried this out in practice.

4. Optimality of Split

The optimality of our algorithm is relative to the cost function, which assumes a parallel execution model in which every statement has identical cost. Statements combined with "|" are assumed to run in parallel and thus we take the maximum of their costs, while statements combined with ";" run serially and so we add their costs. This corresponds closely to the execution model of the `Haxl` monad, and it is sufficient to give good results for other monads because it favours `<*>` over `>>=`. As we saw earlier (Section 2.1), we can sometimes do better if we have more knowledge about the exact cost of statements, but in general that knowledge is not available.

4.1 Optimality of Outer Parallelism

Our rearrangement algorithm exploits outer parallelism first, using the function segments in Figure 4. It is not immediately clear that this gives optimal results, so in this Section we formalise that claim.

Consider a sequence of n statements $s_1 \ldots s_n$. Let C_{ij} stand for the optimal cost of rearrangement of a subsequence $s_i \ldots s_j$. We make no assumption about the relative costs of individual statements, hence we let $C_{ii} = 1$. Furthermore, $C_{ij} \geq 1$ for all non-empty subsequences $1 \leq i \leq j \leq n$.

LEMMA 4.1 (Monotonicity). *Expanding a subsequence by one statement to the left or to the right cannot reduce the optimal cost, and can increase it by at most 1:*

$$C_{ij} \leq C_{(i-1)j} \leq C_{ij} + 1$$

$$C_{ij} \leq C_{i(j+1)} \leq C_{ij} + 1$$

Proof. The upper bound is achieved by sequentially composing the new statement with the original subsequence: s_{i-1} ; $(s_i \ldots s_j)$ or $(s_i \ldots s_j)$; s_{j+1}. The lower bound can be proved by induction on the length of subsequences. The base case $1 \leq C_{i(i+1)}$ trivially follows from $C_{ij} \geq 1$ (the case where we expand to the left follows by symmetry). For the inductive step we examine two cases:

1. The optimum in $C_{i(j+1)}$ is achieved by sequential composition $(s_i \ldots s_k); (s_{k+1} \ldots s_{j+1})$ for some $i \leq k \leq j$. Then,

   $C_{i(j+1)} = C_{ik} + C_{(k+1)(j+1)} \geq^{(*)} C_{ik} + C_{(k+1)j} \geq^{(**)} C_{ij}$,

   where (*) follows from $C_{(k+1)j} \leq C_{(k+1)(j+1)}$ (the inductive hypothesis), and (**) is due to the optimality of C_{ij}.
2. The optimum in $C_{i(j+1)}$ is achieved by parallel composition $(s_i \ldots s_k) \mid (s_{k+1} \ldots s_{j+1})$ for some $i \leq k \leq j$. Then,

   $C_{i(j+1)} = \max(C_{ik}, C_{(k+1)(j+1)}) \geq \max(C_{ik}, C_{(k+1)j}) \geq C_{ij}$,

   where the inequalities hold for the same reasons as in case 1. ■

THEOREM 4.2 (Parallelism is optimal). *If a subsequence $s_i \ldots s_j$ can be split into two segments $s_i \ldots s_k$ and $s_{k+1} \ldots s_j$ with no dependencies between them then $C_{ij} = \max(C_{ik}, C_{(k+1)j})$, and the optimum is achieved by combining the segments using parallel composition $(s_i \ldots s_k) \mid (s_{k+1} \ldots s_j)$.*

Proof. Thanks to the Monotonicity Lemma 4.1 one can see that $C_{ij} \geq C_{ik}$ and $C_{ij} \geq C_{(k+1)j}$, which can be combined into the following lower bound: $C_{ij} \geq \max(C_{ik}, C_{(k+1)j})$. The parallel composition achieves the lower bound and is therefore optimal. ■

4.2 Optimal Sequential Split

When a subsequence $s_i \ldots s_j$ has no outer parallelism, we have to use a sequential split $(s_i \ldots s_k)$; $(s_{k+1} \ldots s_j)$ instead. One can find the optimum k in linear time by examining all $j - i$ splits:

$$C_{ij} = \min_{i \leq k < j} \{C_{ik} + C_{(k+1)j}\}.$$

Since there are at most $O(n^2)$ different subsequences $s_i \ldots s_j$, the overall worst case complexity of the algorithm is $O(n^3)$. Fortunately, it is often possible to avoid iterating through all values of k, hence significantly improving the average case complexity.

Consider two splits $s_i; (s_{i+1} \ldots s_j)$ and $(s_i \ldots s_{j-1}); s_j$. Their costs are $L = C_{(i+1)j} + 1$ and $R = C_{i(j-1)} + 1$, respectively.

THEOREM 4.3 (Sequential split). *If $L \neq R$ then $C_{ij} = \min(L, R)$ and the optimum is achieved by the split with the lower cost.*

Proof. From the Monotonicity Lemma 4.1 we have:

$$L - 1 \leq C_{ij} \leq L \quad \wedge \quad R - 1 \leq C_{ij} \leq R$$

By combining the lower bounds we get $C_{ij} \geq \max(L, R) - 1$. We also know that $\min(L, R) + 1 \leq \max(L, R)$ since $L \neq R$. Hence:

$$C_{ij} \geq \max(L, R) - 1 \geq (\min(L, R) + 1) - 1 = \min(L, R).$$

Since $\min(L, R)$ achieves the lower bound it must be optimal. To construct a solution with such cost we choose one of the two extreme splits, namely $s_i; (s_{i+1} \ldots s_j)$ or $(s_i \ldots s_{j-1}); s_j$. ■

Theorem 4.3 reduces the complexity of the sequential split to $O(1)$ when $L \neq R$. See the example in Section 3.4, where this optimisation could have been used to avoid checking all 4 possible splits. The theorem is not applicable in the $L = R$ case, but we conjecture that this case can also be solved in $O(1)$ amortized time. We leave this for future work.

4.3 Optimising Rearrangement

The rearrangement algorithm in Figure 4 considers every partitioning of every segment, which means a naïve implementation would require time exponential in the length of the statement sequence. However, since subsequences are examined multiple times, we can apply dynamic programming. Caching the result for a subsequence makes the algorithm as a whole $O(n^3)$: we have $O(n)$ start points, $O(n)$ end points, and processing each subsequence is $O(n)$.

This algorithm is almost identical to the CockeYoungerKasami (CYK) parsing algorithm, which finds all the parses for a string of length n for a context-free grammar. It works bottom-up, by considering sequences of unit length, then sequences of length 2, and so on.

In our case, rather than finding all parses for each subsequence, we are only interested in the optimal parse (this does not affect the time complexity, only space). Furthermore, in practice, finding the top-level parallelism using `segments` tends to prune the search space considerably, and many subsequences need not be considered. Thus, rather than populating the matrix of possibilities bottom-up as in CYK parsing, it is better to use a lazy cache in which values for each subsequence are calculated if necessary and then cached. This is easily implemented in Haskell as a lazy array or map.

5. Implementation and Results

Our implementation of applicative **`do`**-notation is included in GHC 8.0.1. as the `ApplicativeDo` language extension. Language extensions are enabled explicitly in GHC, either by a declaration in the source file, or by a command-line option to the compiler.

5.1 Implementation Architecture

The implementation follows a slightly different pattern than the presentation in Section 3, although the overall result is the same. There are two competing concerns in the implementation:

- We want to perform our transformation before type inference, because it affects inferred types. Some **`do`** expressions require only `Functor` (see Section 3.5), some require only `Applicative`, and the rest require `Monad`.

  Furthermore, it is useful to do desugaring during the name-resolution phase (*renaming*) that comes before type inference, because information about free variables (which is required by both `rearrange` and `desugar`) is readily to hand during this stage.
- On the other hand, if there is a type error in the code, we want to present type errors to the user in terms of the original source code and not the rearranged code that our algorithm produces. For this reason we can't just apply `rearrange` and `desugar` before typechecking, because the shape of the original code is lost. GHC performs desugaring for all the other syntactic constructs *after* type inference for this very reason.

The solution we use is for rearrangement to *annotate* the abstract syntax tree with enough information that the type checker can infer the correct type, and so that the later desugaring phase can produce the correct applicative expressions. We have to be careful to strip the annotations when reporting code fragments back in the form of errors or warnings.

Type inference needs to infer the types of the operators used by **`do`** notation desugaring: `<$>`, `<*>`, and `>>=`, because GHC supports

a language extension called `RebindableSyntax`, in which operators needed during desugaring refer to whatever operators with these names are in scope, rather than the specific instances of these operators from the standard library. Even though the typechecker is inferring the types of these operators, it must be careful that any type errors in the code do not appear when inferring the types of these operators, and instead are reported against constructs in the original source code. This is rather delicate, but possible if careful attention is paid to the the order of unification when type-checking **`do`** expressions.

5.2 Optimality vs. Compile Time

The optimal algorithm we described earlier has complexity $O(n^3)$, which can have a severe impact on compile time for larger **`do`** expressions (we will give some figures in Section 5.4). Out of a desire for more predictable compile times, we also implemented a heuristic version of our algorithm that improves the complexity to $O(n^2)$ at the expense of optimality in some cases.

The heuristic version of the algorithm abandons the exhaustive search in split in favour of a local decision: we split the sequence after the longest initial subsequence of mutually-independent statements. Since we never examine a subsequence multiple times, this also avoids the need to use dynamic programming.

This policy was arrived at after considering examples that arose in the wild, and tends to do well: we achieve the optimal result in about 98% of cases (measurements will be presented in more detail in the next section). The heuristic algorithm is currently the default in GHC, while the optimal one is available as an option.

One could imagine alternative heuristics that might produce better results. For example, we could use the optimal split for short sequences but a local decision for larger ones. We leave for future work a more thorough investigation of alternatives here.

5.3 Results: How Often Does `ApplicativeDo` Apply?

We tested `ApplicativeDo` on two large codebases:

- 1188[4] Haskell packages from LTS Stackage 3.2[5]. In total the Haskell code in these packages contained 38,850 **`do`** expressions, of which 16,293 (41.9%) included at least one use of `<*>` when translated by `ApplicativeDo`. Furthermore, 10,899 (28.0%) were fully desugared into `Applicative` and `Functor` combinators and thus would not require a `Monad` constraint.

  The optimal algorithm found a better rearrangement than the heuristic algorithm in 226 cases, which is 0.6% of all **`do`** expressions, and 1.4% of those where `ApplicativeDo` introduced `<*>`.

- The Haxl codebase at Facebook. In here there were 28,273 **`do`** expressions, `ApplicativeDo` used at least one `<*>` in 5,498 cases (19.4%), and 7,600 (26.9%) were fully desugared into `Applicative` and/or `Functor`.

  The optimal algorithm found a better rearrangement in 141 cases, which is 0.4% of all **`do`** expressions, and 2.6% of those where `ApplicativeDo` introduced `<*>`.

Figure 6 is a histogram with our cost measure on the x axis and the number of **`do`** expressions with that cost on the logarithmic y axis, for the Stackage codebase. There are two data sets: first without applying `ApplicativeDo` (the dotted lines, where the heavier dotted line is a moving average of width 4) and after applying the heuristic `ApplicativeDo` (the solid red line). Without `ApplicativeDo`, the cost is equal to the number of statements in the sequence. We truncated the x axis at 50 to give a better view of the more common sizes to the left; in fact there were a few extreme outliers with costs over 300.

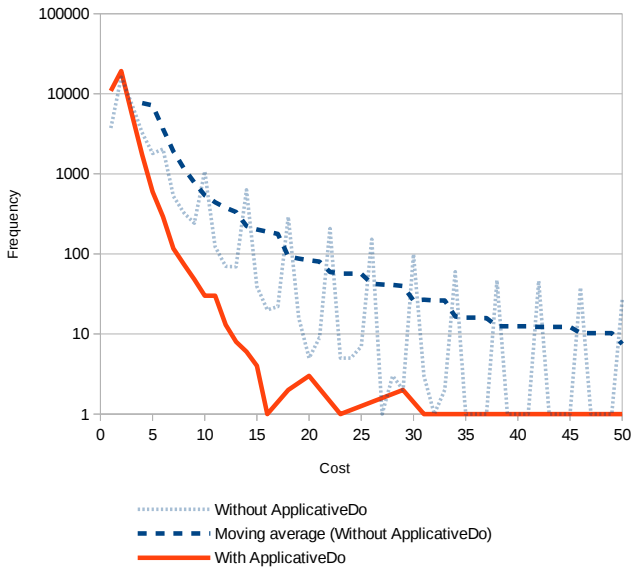


Figure 6. Frequency of **`do`** expression costs in Stackage packages, before and after `ApplicativeDo`

Without `ApplicativeDo`, the median cost is 2 and the 99th percentile is 30, and after `ApplitiveDo` the median cost is also 2, although the 99th percentile is 6. It is clear from these results that `ApplicativeDo` finds plenty of opportunity for parallelism in the **`do`** expressions that occur in typical Haskell code.

There is a strangely regular pattern of spikes in the pre-`ApplicativeDo` data. We investigated this, and it turned out to be due to derived instances of the `Read` class in automatically-generated code in the `amazonka` family of packages. Derived `Read` generates **`do`** expressions for parsing, and these packages contain a lot of data types with similar shapes.

5.4 Compile-Time Overhead

Worst case. We measured the compile time for a single file containing a **`do`** expression with 1000 statements in which each statement depends on the previous one, so that there are no segments. Compiling this file without optimisation:

without `ApplicativeDo`	1.22s
with `ApplicativeDo` (heuristic)	1.46s (20% slower)
with `ApplicativeDo` (optimal)	55.5s (4549% slower)

Note that in all cases the code being generated is the same, because there are no opportunities for `ApplicativeDo` to introduce the `<*>` operator, so the overhead is due purely to the `ApplicativeDo` algorithm itself.

This is only one data point, and we can make both versions of `ApplicativeDo` arbitrarily slow by using a large enough **`do`** expression. But 1000 statements is extremely rare (in LTS Stackage 3.2 the largest was 302), so our heuristic algorithm will not have a noticeable effect on compile-time. However, the optimal version of the algorithm can have a significant effect on compile time—at 300 statements it imposes a 400% overhead—which is why we left it as an option.

[4] About 160 packages failed to compile, mostly due to missing C library dependencies on the host platform.

[5] `https://www.stackage.org/lts-3.2`

Average case. We measured the compile-time overhead of both variants of `ApplicativeDo` for our Haxl codebase. We measure unoptimised compile-time, so as not to dilute the compile-time with the extra cost of optimisation. These measurements were the average of three complete compiles, and we give error bounds to 2 standard deviations:

without `ApplicativeDo`	450s +/- 2s
with `ApplicativeDo` (heuristic)	449s +/- 2s
with `ApplicativeDo` (optimal)	449s +/- 2s

There was essentially no measurable difference between the three modes. Neither the heuristic nor the optimal `ApplicativeDo` algorithms have any measurable impact on the compile time for this codebase.

We did not measure compile time for the Stackage codebase, because the build system performs a lot of activities that are not compiling Haskell files (configuration, installing packages, and so forth), so it was not possible to get a meaningful measurement.

5.5 Performance Improvement

Sigma is a general *detection system* at Facebook. Amongst other things, it classifies actions on Facebook to detect spam and other kinds of abuse. Sigma handles over one million requests per second using many machines across Facebook's different data centers.

Classification is performed by a set of rules, which are implemented in Haskell using the `Haxl` framework and a set of libraries developed for interacting with other back-end services. The rule code uses **`do`** notation, and the `ApplicativeDo` transformation ensures that this code exploits the `Applicative` operators that allow data-fetching requests to be batched and overlapped with the `Haxl` monad.

It is difficult to get an accurate measure of the benefit obtained from `ApplicativeDo`, because there are a huge number of variables. The effect we want to measure is the difference in concurrency when accessing external systems, which is inherently unpredictable: those other systems have their own varying performance characteristics due to caching and load differences. Moreover, the underlying data may change, so requests cannot be reliably replayed.

With these variables in mind, we measured Sigma performance as follows. We measured three common request types independently (Sigma handles hundreds of different requests), to eliminate differences in workload mix. For each request type, we took a sample of recent production requests, and measured the average latency of these requests with and without `ApplicativeDo`. We ran Sigma in single-threaded mode—normally Sigma runs with many threads processing requests in parallel, but for our purposes that would introduce more variables and obscure the latency difference we are trying to measure. Each separate test had to use a brand new sample of traffic, to mitigate the effects of external caching. We used a large enough traffic sample that the run lasted several minutes in each case, to mitigate the effects of differences in the samples.

- In request type 1 (typical latency around 150ms) there was a 44% improvement in average latency with `ApplicativeDo`.
- In request type 2 (typical latency around 125ms), there was a 34% improvement in average latency with `ApplicativeDo`.
- In request type 3 (typical latency around 12ms), there was a 22% improvement in average latency with `ApplicativeDo`.

6. Applications of Applicative do-Notation

Haskell's **`do`**-notation does not add new expressive power to the language; it is just syntactic sugar. But it is *powerful* syntactic sugar, and in practice **`do`**-notation is ubiquitous in Haskell programs. By extending **`do`**-notation to applicative functors we make two main gains. First, we can use **`do`**-notation for types that are `Applicative` but not `Monad`. Second, even where the type is a `Monad` there may be compelling efficiency reasons for wanting to use `Applicative` combinators wherever possible. Our main example, `Haxl`, gains parallelism thereby, but there are monads where the program is *asymptotically* more efficient if you use `Applicative` combinators. In this section we review examples of both these gains.

6.1 Parsing Command-Line Options

The `optparse-applicative` package is a library for parsing command-line options. It provides an `Applicative` (but not `Monad`) abstraction which serves two purposes: it builds the data structure representing the options, while at the same time specifying how to parse them. Here is how it looks without `AppliativeDo`:

```
data Options = Options
  { input :: FilePath
  , verbose :: Bool }

options :: Parser Options
options =
  (λi v → Options { input = i
                  , verbose = v })
    <$> strOption (  long "input"
                  <> help "Input file" )
    <*> switch    (  long "verbose"
                  <> help "Whether to be verbose" )
```

Here, `options` specifies a parser for two options, `--input` and `--verbose`, and a data structure, `Options`, to hold their values.

The problem is that we want to define the `Options` type using record syntax because it's more extensible, but using record syntax in the parser is cumbersome. We have to match the order of the arguments in the applicative expression with the order of the lambda-bound variables, which can become error prone when there are many options. For this reason people often abandon record syntax when building parsers for `optparse-applicative`, but that also sacrifices easy extensibility.

Using **`do`** notation with `ApplicativeDo` is cleaner and more extensible:

```
options :: Parser Options
options = do
  i ← strOption (  long "input"
                <> help "Input file" )
  v ← switch    (  long "verbose"
                <> help "Whether to be verbose" )
  return Options
    { input = i
    , verbose = v }
```

6.2 The `Seq` Data Type

In the `containers` package, `Seq` provides a length-annotated finger-tree [7] as a general-purpose catenable sequence type. The `>>=` operation for `Seq` behaves in the same way as lists: it applies the second argument for each element of the sequence returned by the first argument, and so has complexity $O(mn)$.

The `<*>` operation, on the other hand, can exploit the fact that it will be concatenating many trees of the same size, and by using lazy evaluation is able to provide access to a single element of the result in at most $O(m + \log n)$, even though accessing the whole of the result is still $O(mn)$. For example, with `ApplicativeDo`, this:

```
take 10 $ reverse $
   do { x ← a; y ← b; return (x+y) }
```

is instantaneous, but without `ApplicativeDo` it requires the full $O(mn)$ where m and n are the lengths of `a` and `b`. Of course we could write this explicitly using `<*>`, but the **`do`** notation is clearer and allows us to use real `Monad` bind when necessary too. `Seq` also provides an efficient `<$>`, which our new desugaring takes advantage of.

6.3 LL(1) Parsing

Swierstra and Duponcheel [20] described an non-monadic LL(1) parser that is guaranteed to parse a proper LL(1) grammar in linear time. It does so by tracking a FIRST set for each parser. It is also capable of checking if such a parser is really LL(1) or if it contains FIRST/FIRST or FIRST/FOLLOW set conflicts.

The FIRST set for a parser contains the set of terminals that this parser is able to accept as the first symbol of a successful parse and a flag to indicate whether or not an empty parse will be accepted.

Such a parser extends to a `Monad` at the cost of the linear time guarantee and ability to check a parser for FIRST/FIRST and FIRST/FOLLOW conflicts, while retaining this guarantee for the `Applicative` fragment.

6.4 Heap of Successes Parsing

We can modify Wadler's "List of Successes" parser [21] in two ways to allow for more efficient `Applicative` parsing in the presence of heavy non-determinism.

```
newtype Parser a = Parser (String → [(a, String)]
```

Borrowing the notion of an update monad from Ahman and Uustalu [1], instead of giving back the new `String`, we can give back how much of the string we've consumed, and between parse steps `drop` this many characters from the `String`. This costs us the ability to "push back" input we haven't actually consumed, but opens up the next option.

```
newtype UpdateParser a = Parser (String → [(a, Int)]
```

Next we can track a heap of successes rather than a list, sorted by length.

```
newtype HeapParser a = Parser(String → IntHeap [a])
```

Now, code written using `<*>` needs only execute the right hand parser once per distinct length, rather than once per distinct parse. By further augmenting such a structure, we could recover the original parse order.

6.5 Moore Machines

A possibly-infinite Moore machine with states labeled by `b` and transitions labeled by `a` can be represented with explicit state as the following GADT:

```
data Moore a b where
  Moore :: (r → b) → (r → a → r) → r → Moore a b

instance Applicative (Moore a) where
  pure a = Moore (const a) const ()
  Moore xf bxx xz <*> Moore ya byy yz = Moore
    (λ(x, y) → xf x $ ya y)
    (λ(x, y) b → (bxx x b, byy y b))
    (xz, yz)
```

The implementation of `<*>` for such a machine takes the product of the state spaces and builds a new machine.[6] Another way to think of such a machine is as a strict left fold [19], and `<*>` takes two independent folds and melds them in a single pass.

There even exists a `Monad` for this type, but it is grossly inefficient. It can be obtained by showing that `Moore a b` is naturally isomorphic to `[a] -> b`. To operate it has to record every value the machine is fed, and then feed each machine that labels our states the entire input seen thus far, just to take a single output from each machine.

With `ApplicativeDo`, we can work fairly naturally with such machines without incurring the horrible overhead of the `Monad`, whenever the passes are independent.

```
sum :: Num a => Moore a a
sum = Moore id (+) 0

length :: Moore a Int
length = Moore id (λx _ → x + 1) 0

mean :: Fractional a => Moore a a
mean = do
   a ← sum
   b ← length
   return (a / fromIntegral b)
```

That said, the `Monad` cannot be avoided entirely, as some computations simply require multiple passes over the data, such as computing robust statistics like median absolute deviation, which requires a pass to compute the median followed by another dependent pass to compute the median distance to the median we just identified.

7. Related Work

7.1 Extracting Parallelism

Extracting parallelism automatically from programs is a much studied problem. Two approaches dominate: extracting implicit parallelism from a program written in a largely-unmodified host language, or expressing parallelism explicitly with a domain-specific language or library, such as map/reduce [5], LINQ [8] or Accelerate [3].

Applicative **`do`**-notation combines features of both:

- The possibility of parallelism is signalled explicitly by the use of **`do`** notation, but
- A lexical dependency analysis is used to figure out exactly which statements can be run in parallel.

Moreover, in Haxl there is a runtime component too: a computation is only run in parallel if it initiates a remote data fetch.

7.2 Idiom Brackets

Idiom brackets [16] provide a concise syntax for writing applicative expressions, where $[\![f\ e_1 \ldots e_n]\!]$ is equivalent to

f `<$>` e_1 `<*>` $\ldots$ `<*>` e_n

The special syntactic form is used heavily in Idris [2] and is also implemented in the She Haskell preprocessor [15]. With ingenious use of overloading a similar syntax can be implemented in Haskell itself[7], and idiom brackets have also been implemented via GHC's quasi-quotation extension [12].

Compared with applicative **`do`**-notation, idiom bracket syntax only provides an abbreviation for applicative expressions. It doesn't allow for a mixture of applicative and monad operations, nor does it provide the flexibility of the **`do`** syntax when used with a pure applicative, as we described in Section 6.

[6] To avoid leaking memory a product type strict in both arguments really should be used instead of (,).

[7] `https://wiki.haskell.org/Idiom_brackets`

The F# language has an experimental extension (implemented in a research branch), that supports a parallel (applicative) binding form in F#'s computation expressions [17] (the equivalent of **do** notation). The use of applicative binding is fully explicit, rather than implicit as in our case.

7.3 Formlets

Formlets [4] describe an abstraction for defining parts of a web page. The abstraction is based on applicative functors, and the syntax for defining a formlet is essentially **do**-notation for applicative functors, albeit with the binding arrows reversed and embedded in HTML. The difference from our work is that the syntax only covers applicative functors, not monads.

7.4 Monad Comprehensions

Monad comprehensions [6] also includes a "`|`" operator in its syntax for statement sequences. In Monad Comprehensions, the "`|`" operator desugars into a call to `mzip` from the `MonadZip` class, which for lists is equal to `zip`, while for other monads such as `Maybe` it is equal to `liftM2 (,)`. For monads where `mzip = liftM2 (,)`, the "`|`" syntax of Monad Comprehensions can be used to write applicative expressions, since `liftM2 = liftA2`. Therefore, for *some* monads, Monad Comprehensions provides an explicit way to combine computations applicatively within a monad comprehension. However, this is somewhat accidental, since the intention of the "`|`" operator in Monad Comprehensions is to support zipping, and the `MonadZip` class was introduced as the natural generalisation to monads of zipping on lists.

It is not in general semantics-preserving to flatten the "`|`" operator of a monad comprehension to a sequence, unlike in our syntax, and thus monad comprehensions cannot automatically introduce "`|`" via a `rearrange` transformation.

While we have not done so yet, we believe it would be entirely possible to apply `ApplicativeDo` to monad comprehensions, and we do not anticipate any complications with doing that.

7.5 Other Related Work

The Arrow Calculus of Lindley, Wadler and Yallop [11] includes a form of let-binding which can be viewed as being like do-notation for applicative functors, arrows, and monads. Lindley later described a different calculus based on algebraic effects and call-by-push-value [10] that also gives rise to a form of do-notation for applicative functors, arrows, and monads.

8. Conclusion and Future Directions

Applicative **do**-notation has characteristics of both syntactic sugar and compiler optimisation, which is somewhat unusual. When used with a non-Monad, applicative **do**-notation behaves like syntactic sugar: it is obvious when it applies, and what effect it will have. But when used with a Monad, it is arguably not transparent to the programmer where the compiler will introduce applicative operators, and indeed the applicative structure that the compiler derives may change as the code is refactored. This is a feature, not a bug: the aim is to provide a notation that abstracts away from the applicative structure while still being able to take advantage of the applicative operators. Applicative **do**-notation works well for those situations where functionality and ease of refactoring take precedence over performance, but we're not prepared to give up on performance altogether.

That said, we do not claim that **do**-notation is universally better than explicit applicatives. Indeed, in many simple cases, using explicit applicative combinators is both shorter and more readable than **do**-notation; but our experience is that this becomes less true as the complexity of expressions increases. When a **do**-notation expression contains tens of statements, writing it with explicit applicatives is unwieldy to say the least, and doing it optimally is virtually impossible. Applicative **do**-notation works well in these cases.

In performance-critical situations where the programmer wants to be certain that they are achieving the desired applicative structure, they can still write explicit applicative code. Indeed, as we mentioned in Section 3.10, we believe that exposing the "`|`" operator from our abstract syntax at the source level (in some form) would be helpful in allowing programmers to be explicit about the applicative structure. This is a possible direction for future work.

A couple of other areas we plan to explore are:

- Our desugaring does not currently exploit the `*>` or `<*` combinators that `Applicative` provides, and in certain cases using these operators instead of `<*>` can result in performance benefits.
- It is possible, but unimplemented, to further reduce the search space for `split` by observing that our dynamic programming solution has the same structure as a minimax problem allowing us to exploit alpha-beta pruning [9], computing an alpha-beta bounded transposition table rather than a classic dynamic program.

  MTD(f) [18] is a particularly applicable pruning technique, because the length of a **do** expression acts as a conservative upper bound on our cost function, our result is an integer drawn from a very small range, and our transposition table is considerably smaller than that of most games to which it has been applied.

  MTD(f) would not improve the worst-case cost of computing an optimal solution, but based on limited experimentation, it should bring some extreme examples, such as the one in Section 5.4, back into line with heuristic compile times. In addition, with MTD(f), we could allow a parameterized early cut-off, to smoothly interpolate between the heuristic and optimal algorithms.

Acknowlegments

Andrey Mokhov conducted this work during a 6-month research visit to Microsoft Research Cambridge that was funded by Newcastle University, EPSRC (grant reference EP/K503885/1), and Microsoft Research.

References

[1] D. Ahman and T. Uustalu. Update monads: cointerpreting directed containers. In *Proc. of 19th Int. Conf. on Types for Proofs and Programs, TYPES*, volume 13, pages 1–23, 2014.

[2] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[3] 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*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. . URL `http://doi.acm.org/10.1145/1926354.1926358`.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. *The Essence of Form Abstraction*, pages 205–220. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. . URL `http://dx.doi.org/10.1007/978-3-540-89330-1_15`.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, Dec. 2004.

[6] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. In *Proceedings of the 4th ACM Sym-*

posium on Haskell, Haskell '11, pages 13–22, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. . URL http://doi.acm.org/10.1145/2034675.2034678.

[7] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(02):197–217, 2006.

[8] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 987–994, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. . URL http://doi.acm.org/10.1145/1559845.1559962.

[9] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1976.

[10] S. Lindley. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 47–58, New York, NY, USA, 2014. ACM. . URL http://doi.acm.org/10.1145/2633628.2633636.

[11] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20:51–69, 1 2010. ISSN 1469-7653. . URL http://journals.cambridge.org/article_S095679680999027X.

[12] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 73–82. ACM, 2007.

[13] S. Marlow. Haskell 2010 language report. 2010. URL https://www.haskell.org/onlinereport/haskell2010/.

[14] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337. ACM, 2014.

[15] C. McBride. The Strathclyde Haskell Enhancement (SHE). URL https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/.

[16] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

[17] T. Petricek and D. Syme. *The F# Computation Expression Zoo*, pages 33–48. Springer International Publishing, Cham, 2014. . URL http://dx.doi.org/10.1007/978-3-319-04132-2_3.

[18] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1):255–293, 1996.

[19] M. Rabkin. Beautiful folding. Nov. 2008. URL http://squing.blogspot.com/2008/11/beautiful-folding.html.

[20] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, pages 184–207. Springer-Verlag, 1996.

[21] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[22] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. . URL http://doi.acm.org/10.1145/91556.91592.