Compiling Agda to Haskell with fewer coercions

W. H. Kusee
ICA-3800296

Supervisors:
dr. W.S. Swierstra
dr. J. Hage

Department of Computing Science
University of Utrecht

December 7, 2017

Abstract

The Agda programming language is most often used as a theorem prover. Agda programs can also be compiled using the GHC backend, which translates an Agda program to a Haskell program that can be compiled by the GHC compiler. Because the Agda programming language has multiple features that are difficult to translate to Haskell automatically, the GHC backend creates simple Haskell programs that may contain type errors. Coercions (using the Haskell unsafeCoerce function) are then inserted to avoid these type errors.

This thesis changes the GHC backend so that it only inserts coercions where necessary, by utilizing the type errors that GHC reports and inserting coercions at those locations. To further lower the number of needed coercions, the translation of Agda data types is improved by retaining more type information in the generated Haskell data types.

Contents

1. Introduction

1.1. The Agda language

The Agda language is both an interactive system for developing constructive proofs, and a programming language with dependent types. Agda is mostly used as a theorem prover, where typechecking a program is more important than running it.

The Agda compiler, *Alonzo* [1], can also generate executable programs. The most used Agda backend, the GHC backend (formerly called *MAlonzo*), can be used to generate Haskell code, which is then compiled by the GHC compiler. However, the Agda language has some features that are hard to translate to Haskell, such as dependent types. The GHC backend solves this by simplifying the generated code, leaving out some parts of the Agda program such as type information. The generated code has some problems, however.

1.2. Translating Agda to Haskell

An often-used data type in Agda is the natural number, which is recursively defined as either zero, or the successor of a natural number:

Agda

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

While this data type can be straightforwardly translated to Haskell (`data Nat = Zero | Succ Nat`), not all Agda data types can be expressed in Haskell. Instead, the backend creates very simple constructors, which do not assume anything about its arguments by using parameters instead:

Haskell

```
data Nat a = Zero | Succ a
```

By keeping the data types very general, it becomes easier to translate more advanced Agda features to Haskell. But while this data definition works, it becomes tricky to translate functions, such as a function for addition:

Agda

```
_+_ : Nat → Nat → Nat
zero + m = m
succ n + m = succ (n + m)
```

When translating this function to Haskell, its type should be `Nat a → Nat a`. The problem lies in the recursion: the `Succ` constructor expects an expression of type `a`, but `n + m` will have type `Nat a`. There is not enough type information left to let the program pass the GHC typechecker.

To evade this limitation, the GHC backend opts to convince GHC that the program is correct, by telling GHC that the types are equivalent:

Haskell

```
-- Function used by the backend
coerce :: a → b
coerce = unsafeCoerce

-- Generated function
plus n m
  = case coerce n of
      Zero → coerce m
      Succ n' → coerce Succ (coerce plus n' m)
```

By inserting coercions before most Haskell expressions, Haskell's type system is essentially bypassed.

1.3. Contributions

The GHC backend of the Agda compiler generates well-typed Haskell programs by inserting coercions. There are some problems with this approach, however. Because coercions reduce the amount of type information, and thereby makes it harder to optimize the program by inlining expressions, GHC may be unable to optimize the program as much as it would be able to without the coercions. The unnatural translation of data types also makes it harder to interface with generated code from Haskell programs. To alleviate these problems, this thesis makes the following contributions:

- In chapter 2, we give a brief overview of the Agda compiler, and how it inserts coercions to generate Haskell programs.
- Rather than inserting coercions almost everywhere, we opt instead to insert them only where necessary. To achieve this, GHC will be called as a library, so that the backend can access the internal representation of the Haskell code in GHC. Chapter 3 introduces error-based coercion insertion: after the typechecking phase, coercions can be inserted at error locations, after which the program can be typechecked again. This process can then be repeated until there are just enough coercions for the program to be well-typed.
- The data type translation can be improved, so that fewer coercions are needed, without having to change the translation of functions and expressions (chapter 4). This is achieved by only translating parameters in data definitions, and keep more type information in constructors. As Haskell data definitions are essentially a subset of Agda's data definitions in terms of what they can express, data types that fall in that subset should be translated directly to Haskell. Some data type features only found in Agda, such as indices and levels, do not need to be present in the Haskell data definition. Other features, such as type-level functions, are very hard to translate to Haskell, and can be existentially quantified inside data constructors.
- We compare the performance of these changes with the old backend (chapter 5), to show how much of an impact the coercions have on both the compile-time and run-time of generated Haskell programs.

- Then, we give a brief outline of some related work on this topic (chapter 6), and conclude with some possible future work (chapter 7).

2. The Agda compiler

When compiling an Agda program using the GHC backend, the compiler goes through several stages before producing a Haskell program which GHC can compile. In figure 2.1 we show a conceptual overview of how the compiler translates an Agda program to an executable program.

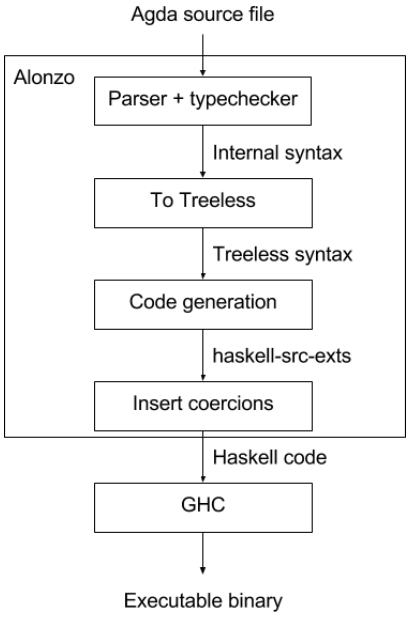

Figure 2.1.: Structure of the Agda compiler

To compile an Agda program to an executable, the compiler goes through the following stages:

- First, the Agda program is parsed and type-checked by the frontend of the Agda compiler. The typechecker works on *internal syntax*, which contains Agda terms with corresponding type information. This syntax is relatively complicated, and generally too bulky to be used by a compiler backend.
- To simplify the code representation, the *internal syntax* terms are translated to the so-called *treeless syntax*. This representation simplifies the syntax in several ways. Section 2.2 describes how this syntax is used.
- The GHC backend then takes the *Treeless syntax*, and transforms it to Haskell code. Because the syntax of these two are quite similar, this transformation is relatively straightforward. Section 2.3 gives a description of the code generation.

Agda

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat

_+_ : Nat → Nat → Nat
zero   + m = m
succ n + m = succ (n + m)

data Vec (A : Set) : Nat → Set where
  []  : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (succ n)

vappend : {A : Set}{n m : Nat}  → Vec A n → Vec A m → Vec A (n + m)
vappend [] ys = ys
vappend (x :: xs) ys = x :: vappend xs ys
```

Figure 2.2.: An example Agda program showing natural numbers with addition, and vectors that can be combined

- While the previous step produces Haskell code, it did so without taking the types into account. As such, the next step is to insert coercions in the Haskell terms so that the program can pass the GHC typechecker. Section 2.4 describes how this is done.
- At this point, the compiler simply writes the Haskell files to disk, and calls the Haskell compiler GHC to generate a binary.

2.1. Agda programs

The compilation steps used to compile an Agda program to a Haskell program will be explained using a pair of running examples: addition of natural numbers, and an append function for vectors, as shown in figure 2.2.

One of the most used data types in Agda is the natural number. However, simple functions like additions are similar to those in other programming languages, like Haskell. Notably, natural numbers do not make use of dependent types. Therefore, it should be possible to translate natural numbers, and functions over them, without using coercions. When demonstrating dependent types, a data type which is often used in Agda tutorials [10] is the *Vec* data type for vectors. Vectors are similar to lists in Haskell, but differ in that they are indexed by their length.

When writing a function which constructs such a data type, such as the *vappend* function which appends two vectors, the length of the vector has to be calculated in the type. In the type of *vappend*, this is done using the `_+_` function. Such type-level functions are difficult to translate to Haskell, making this a worthwhile example, even though it is more complex.

```
data TTerm = TVar Int
           | TPrim TPrim
           | TDef QName
           | TApp TTerm Args
           | TLam TTerm
           | TLit Literal
           | TCon QName
           | TLet TTerm TTerm
           | TCase Int CaseType TTerm [TAlt]
           | TErased
           | ...
```

Figure 2.3.: Treeless syntax

Treeless syntax

```
Vec._+_ =
  λ a b →
    case a of
      Vec.Nat.zero → b
      Vec.Nat.succ c → Vec.Nat.succ (Vec._+_ c b)

Vec.vappend =
  λ a b c d e →
    case d of
      Vec.Vec.[] → e
      Vec.Vec._::_ f g h →
        Vec.Vec._::_ (Vec._+_ f c) g (Vec.vappend _ f c h e)
```

Figure 2.4.: _+_ and `vappend` functions in treeless syntax

2.2. Treeless Syntax

The next step of the compilation process is to transform the Agda program to *treeless syntax*. One of the more important transformations done in this step is simplifying Agda's case trees, which are formed when *with* bindings are used, to nested case statements. This step also removes the type information associated in the terms, as most backends do not make use of the type information. Because this transformation eliminates these case trees, the syntax is called *treeless syntax*.

An overview of the treeless syntax can be found in figure 2.3. The syntax is similar to the λ-calculus, and uses de Bruijn indices to represent local variables (the innermost lambda binding is 1, the outer binding is 2, etc.). Let bindings and case statements can be straightforwardly translated to Haskell. The *TErased* construct is used as a dummy value for Agda terms which are not needed in the run-time representation. Finally, *TPrim* values denote primitive operations, such as integer addition and list concatenation.

This syntax is used by multiple backends of the Agda compiler, because it is easier to work with than the *internal syntax*, the main difference being that case expressions are simplified. Agda's case trees, which are formed when *with* bindings in Agda are used, are simplified to nested case statements, which can

be directly translated to Haskell.

2.3. Haskell code generation

The GHC backend uses the *treeless syntax* terms to directly generate Haskell code. The *Haskell-src-exts* Haskell library is used to directly create the terms, instead of having to pretty-print Haskell code.

Simple recursive datatypes, like natural numbers, have a straightforward translation. All constructor arguments are not specified by their type, but instead passed as an argument. The goal of the generated code is not to pass the typechecker, but to generate code that will have the correct result when run. The types can be unnecessarily general, as the next step will insert coercions to guarantee that typechecking succeeds.

As an example, take the following definition of natural number and addition:

Agda

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat

_+_ : Nat → Nat → Nat
zero + m = m
succ n + m = succ (n + m)
```

The backend simply generates the corresponding Haskell data type with the correct number of parameters, as follows:

Haskell

```
data Nat a = Zero
           | Succ a

plus n m
  = case n of
      Zero → m
      Succ n' → Succ (plus n' m)
```

This particular example will not compile as is, because the application of the *Succ* constructor triggers infinite types, which are not allowed in Haskell.

This translation also works for dependent types. An often used example is the vector append function:

Agda

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (succ n)

vappend : {A : Set}{n m : Nat} → Vec A n → Vec A m → Vec A (n + m)
vappend [] ys = ys
vappend (x :: xs) ys = x :: vappend xs ys
```

This will generate the following Haskell code:

Haskell

```
data Vec n x xs = VNil
                | VCons n x xs

vappend _a _n m v1 v2 = vappend' m v1 v2
vappend' m v1 v2 = case v1 of
  VNil → VNil
  VCons n x xs → VCons (plus n m) x (vappend' m xs v2)
```

Note that both implicit and explicit arguments are parameters to the *Vec* data type, there is no difference in the generated Haskell code. The *vappend'* function is called with only the used arguments. The backend handles type-level computations in Agda by storing them at the value level instead.

To make it pass the GHC typechecker, the next step will be to insert coercions.

2.4. Adding coercions

The generated Haskell code will give the same results as the original Agda program, but is not guaranteed to pass the typechecker. Even simple functions such as *plus* fail to compile. When calling GHC to compile the *plus* function, the following type errors occur:

```
Plus.hs:7:26:
    Occurs check: cannot construct the infinite type: a ~ Nat a
    Relevant bindings include
      m :: Nat a (bound at temp.hs:4:8)
      plus :: Nat t -> Nat a -> Nat a (bound at Plus.hs:4:1)
    In the first argument of ‘Succ’, namely ‘(plus n’ m)’
    In the expression: Succ (plus n’ m)

Plus.hs:7:31:
    Occurs check: cannot construct the infinite type: t ~ Nat t
    Relevant bindings include
      n’ :: t (bound at temp.hs:7:14)
      n :: Nat t (bound at temp.hs:4:6)
      plus :: Nat t -> Nat a -> Nat a (bound at Plus.hs:4:1)
    In the first argument of ‘plus’, namely ‘n’’
    In the first argument of ‘Succ’, namely ‘(plus n’ m)’
```

While this is partly caused by the simplistic data type translation (which will be improved in chapter 4), it is hard to do the translation in such a way so that no type errors will occur. While simpler functions like *plus* and *vappend* might possibly be translated correctly, doing so for more complicated functions with dependent types is a lot harder.

To circumvent the Haskell typechecking requirement, the backend makes use of the *unsafeCoerce* function:

Haskell

```
{−# INLINE [1] coerce #−}
coerce :: a → b
coerce = unsafeCoerce
{−# RULES "coerce−id" forall (x :: a) . coerce x = x #−}
```

This function tricks the typechecker into seeing a term of type a to be whatever type is needed. There are some optimizations for inlining that bypass coercions, but it essentially enables the backend to avoid having incorrect types in the generated code. However, because it is hard to know where the code fails to typecheck, the backend chooses to insert these coercions around every term:

Haskell

```
plus n m
  = case coerce n of
      Zero → coerce m
      Succ n' → coerce Succ (coerce plus n' m)
```

In this example, the coercions ensure that the *Succ* constructor does not trigger any type errors. The *vappend* function is handled similarly:

Haskell

```
vappend _a _n m v1 v2 = vappend' m v1 v2
vappend' m v1 v2 = case coerce v1 of
  VNil → coerce VNil
  VCons n x xs → coerce VCons (coerce plus n m) x (coerce vappend' m xs v2)
```

By adding a coercion at nearly every AST node, GHC struggles to optimize the program. Even though the backend tries to inline coercions by using an optimized version of *unsafeCoerce*, GHC has to be very pessimistic because it has very little type information.

2.5. Implementation

The Agda compiler transforms every Agda definition into a corresponding Haskell definition:

Haskell

```
definition :: Definition → TCM [HS.Decl]
```

The *TCM* (Type Checking Monad) contains information about the typechecking phase of the compiler. There are two important kinds of definitions to be handled: data types, and functions.

2.5.1. Data types

The translation of data types is handled by the following function:

Haskell

```
condecl :: QName → TCM (Nat, HS.ConDecl)
```

Even though Agda data definitions can be very complicated, the translation to Haskell types is quite simple. The *condecl* function adds the number of parameters to the arity of the data type, which results in the number of arguments the Haskell data type takes. As seen in the translation of vectors in the previous section, the translation of constructors does not distinguish between implicit and explicit parameters, and simply makes everything a parameter, instead of specifying the type of its arguments. The data type itself calculates the maximum number of arguments each constructor takes, and uses that as the arity of the data type.

2.5.2. Functions

The translation of functions, or more importantly, terms, can be divided into 3 stages:

- Translating an *internal syntax* term to a *treeless syntax* term;
- creating a Haskell expression,
- and inserting coercions at function applications

These steps are handled respectively by the following three functions:

Haskell

```
toTreeless :: QName → TCM (Maybe TTerm)
term :: T.TTerm → CC HS.Exp
hsCast :: HS.Exp → HS.Exp
```

The *toTreeless* function simplifies the terms somewhat, so that they are easier to handle by the backend. The *term* function generates Haskell expressions based on the *treeless syntax*. Because they are similar, this is fairly straightforward. Lastly, the *hsCast* function is then called to insert coercions around every node in the AST, particularly applications. Some care is taken to avoid applying coercions twice to the same expression.

The coercions are inserted everywhere, with the sole exception of literal integers. However, as there is no type information tracked at this stage, propagated values are still coerced.

3. Error-based coercion insertion

While adding coercions solves the problem of getting the generated Haskell code past the typechecker, several problems emerge from the use of *unsafeCoerce*. First of all, the Haskell code that is created in this way is very 'unnatural'. The way data types are defined, together with the large number of function arguments, makes interoperability with existing Haskell code very difficult. Generating more 'natural' Haskell code would make it far easier to interact with, both from Haskell and using foreign-function interfaces.

Another big problem of coercions is the performance impact. For normal Haskell programs, GHC can do optimizations based on the types of the program, such as inlining. However, because of the large number of coercions, there is less type information to work with to optimize the program. Also, the extra function applications make it harder to inline function calls. An attempt is made to alleviate this by letting the *coerce* function be inlinable and making an optimization rule for it, but this does not remedy the core problem. Coercions are still a severe performance hit for the compiled Agda program, even though most coercions can be omitted. As such, the best way to avoid the performance hit of coercions is to reduce the number of coercions added to the program.

The main problem with reducing the number of coercions is to figure out which coercions are necessary, and which ones are not. While it is possible to produce better code for some programs, for example those using just simple types like natural numbers and lists, it is hard to know where the translation will fail to typecheck. This is because the input language, Agda, is quite big compared to the complexity of the generated Haskell code. Knowing where the translation will fail essentially requires a full-fletched typechecker for Haskell to be implemented.

It is also possible, however, to make use of the existing GHC infrastructure to locate the parts of the program that fail to typecheck. Because GHC exposes its core functionality as a Haskell library, it can be interacted with directly from the Agda compiler, so that the GHC typechecker can do most of the work. Calling GHC as a library has the benefit of avoiding duplicate work, so that the generated Haskell program has to be parsed only once, while enabling access to type error information so that the Haskell AST that GHC exposes through a Haskell library can be modified directly to insert coercions.

3.1. Removing or inserting coercions

The main goal of this thesis is to reduce the number of coercions that are used in the generated Haskell code. There are two possible directions in reducing the number of coercions.

The first possibility is to start with all coercions that might be needed, so that the program is guaranteed to compile, and then remove the ones that are not needed. This keeps the required changes to the Agda compiler simple,

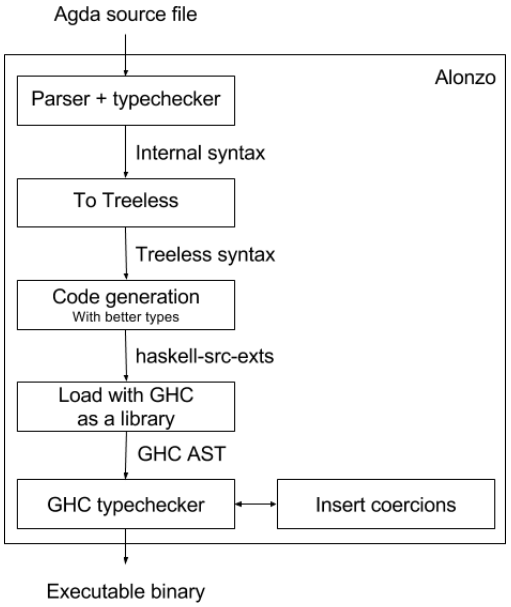

Figure 3.1.: Structure of the Agda compiler using error-based coercion insertion

as those coercions are already inserted in the current compiler. As such, the compiler only needs to be extended with a further pass over the generated Haskell code, without needing to change the existing stages of the compiler. The main downside of this approach, however, is that it is difficult to know which coercions can be eliminated.

In the GHC backend, Agda data types are translated to simplified Haskell data types, that leave out most type information because fully translating dependent types to Haskell is quite difficult. However, by using simpler types, programs that use these data types require more coercions so that there are no type mismatches. To know exactly at which points coercions are needed and at which points they are not, the Haskell types need to be examined, and a typechecker is needed to know which coercions can be omitted. Therefore, while removing coercions might keep the earlier stages of the compilation process the same, it complicates the last stage to the point of requiring another typechecker.

The other direction, in which the number of coercions can be reduced, is to start with no coercions and insert them when needed. This means that we start with a Haskell program that might not compile, because there might be type errors. Such a Haskell program can then be passed to the GHC compiler, which can typecheck the program. Either the program compiles, in which case no coercions are needed at all, or some type errors occur. GHC includes location information when it reports the type errors, which means that coercions are probably needed in those locations. This makes it possible to change the Haskell code to include coercions at those locations, and repeat this process until all type errors are fixed. However, when calling GHC from the command-line like normal, there is one big downside to this approach: GHC needs to go over the whole program each time it is called. When some type errors are fixed by

inserting coercions, GHC needs to start over and parse and typecheck the whole program again. Luckily, these downsides can be fixed by using the GHC Haskell library, instead of calling GHC as a program.

3.2. GHC as a library

While GHC is typically invoked from the command-line, it is also possible to call GHC directly from Haskell, using the exposed GHC library[1]. The Haskell code generated by the backend currently undergoes the following steps:

1. Haskell code is generated
2. Coercions are added
3. Haskell source files are written to disk
4. GHC compiles the program

By delaying the coercions until they are needed to pass the typechecker, the way coercions are added changes:

1. Haskell code is generated
2. Haskell source files are written to disk
3. GHC checks module dependencies and parses each module
4. GHC typechecks each module, and repeatedly inserts coercions at the error locations until it passes the typechecker

Figure 3.2 outlines how the GHC library can be used to implement iterative compilation based on type errors:

- The **iterativeCompilation** function uses the *Ghc* monad, which is essentially a wrapper around *IO* with some added state. The *load* function is used to attempt to compile the whole program. Any type errors are ignored at this moment, as the important part here is the module graph, which will be constructed even if the program contains type errors. After sorting the modules based on their dependencies, and parsing each module a single time, each module can be recompiled in order.
- The **tryCompile** function tries to typecheck the module. If there are any type errors, a *SourceError* will be thrown as an Haskell exception, which will be caught by the *handleError* function. If, instead, there are no more type errors, then the module wil be loaded using the `loadModule` function, which writes the compiled module to disk.
- The **handleError** function takes a parsed module and a thrown *SourceError*. The *SourceError* can contain multiple type errors. Each type error is then inspected for its location, and a coercion will be added in that location using the *insertCoercion* function. When all current type errors are fixed using coercions, the *compileModule* function is called again to check for more type errors.

[1] `https://wiki.haskell.org/GHC/As_a_library`

Haskell

```
iterativeCompilation :: IO ()
iterativeCompilation = runGhc (Just libdir) $ do
  setTargets ["Main.hs"]
  load LoadAllTargets
  modGraph ← getModuleGraph
  parsedMods ← mapM parseModule modGraph
  mapM_ tryCompile parsedMods

tryCompile :: ParsedModule → Ghc ()
tryCompile p = do
  t ← handleSourceError (handleError p) (typecheckModule p)
  loadModule t
  return ()

handleError :: ParsedModule → SourceError → Ghc ()
handleError p sourceError = do
  let errors = srcErrMessages sourceError
  p' ← foldM insertCoercionsInModule p errors
  tryCompile p'

insertCoercionsInModule :: ParsedModule → ErrMsg → ParsedModule
```

Figure 3.2.: Iterative compilation

- The **insertCoercionsInModule** function searches the current Haskell module for expressions that match any of the error locations, and inserts a coercion at that AST location. This is done using the *ParsedModule* type, which holds an AST of the Haskell module, including location information.

3.3. Type errors

When the generated Haskell code is passed to the GHC typechecker, one or more type errors can occur. We will illustrate how such type incorrect code is produced in the following example. Because the GHC backend currently uses a rather simplistic translation for data types, even simple types cause type errors. For example, the Vec data type:

Agda

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (succ n)
```

will generate the following Haskell data type:

Haskell

```
data Vec n x xs = VNil
                | VCons n x xs
```

Note that all arguments to the constructor are taken as parameters to the data type.

The translation of data types can be improved, as will be done in chapter 4, reducing the number of type errors that occur. However, some Agda data

types, such as those containing type-level functions, cannot be straightforwardly translated to Haskell types.

Even though the original Agda program has passed the typechecker, the Agda types are not easily translated to Haskell, resulting in type errors when calling the GHC typechecker on the generated Haskell code. But while the problem lies in the data types, the type errors only occur when constructing values of the data types. In other words, a type error manifests itself when a constructor of a data type is used. One such type error that can occur is the *infinite type* error, which occurs when a recursive data type is constructed. GHC does not allow such infinite types:

```
Plus.hs:7:26:
    Occurs check: cannot construct the infinite type: a ~ Nat a
    Relevant bindings include
      m :: Nat a (bound at temp.hs:4:8)
      plus :: Nat t -> Nat a -> Nat a (bound at Plus.hs:4:1)
    In the first argument of ‘Succ’, namely ‘(plus n’ m)’
    In the expression: Succ (plus n’ m)
```

3.4. Haskell AST traversal

When typechecking a Haskell module, one or more type errors can occur. Each type error holds location information in the form of a *SrcSpan*, which holds the row and column numbers of the AST node at which the error occurs.

While it would be possible to change the Haskell source code directly, doing so would require the source code to be parsed and loaded again, which would slow down the compilation process. Instead, the Haskell AST, which is exposed by the GHC library[2], can be modified directly. Each AST node contains both the Haskell expression itself, and the location of that expression in the Haskell source code.

GHC can report multiple separate type errors in a single module. To achieve good performance, the backend should aim to fix these errors in a single pass, without going through the AST multiple times unless necessary.

Because AST nodes contain location information, it is possible to look only at AST nodes that contain a type error. As such, the best approach is to walk through the AST, and check each node against every type error.

Figure 3.3 shows how a coercion is inserted using location information. The *SrcSpan* type holds information about the starting and ending locations of the expression. When inserting a coercion, the location information of the outer expression is not changed, even though adding a coercion would cause the row number to change, for example. As a coerced expression should not cause any more type errors, it is not a problem if the location information of nested expressions are inconsistent.

Now that we have a list of error locations, the next step is to check each expression in the current Haskell module. If the location information matches of an expression matches the location information of any of the type errors, a coercion is inserted around that expression.

[2] `http://downloads.haskell.org/~ghc/7.10.3/docs/html/libraries/ghc-7.10.3/HsExpr.html`

Haskell

```haskell
-- GHC SrcLoc
data SrcSpan = ...
type Located e = L SrcSpan e

noLoc :: e → Located e

-- HsExpr
type LHsExpr = Located HsExpr

insertCoercion :: [SrcSpan] → LHsExpr → LHsExpr
insertCoercion errorLocs (L l e) = L l  case e of
  HsCase e' matches | any errorInMatch matches → HsCase (coerce e') matches
  _ → if l `elem` errorLocs then coerce e else e
```

Figure 3.3.: Inserting a coercion using location information

The *insertCoercion* function takes a list of error locations, and a Haskell expression that potentially matches any of the error locations. If it does, then a coercion is inserted. A special case is needed for *case* statements: a pattern in a case statement may use a data constructor for its pattern matching, and GHC reports the location of the whole case statement, instead of the part of the pattern where the error occurs.

3.5. Inserting coercions

In the algorithm shown in the previous section, a way to insert a coercion based on a type error is needed. The type error raised by GHC contains location information, in the form of row and column numbers. It does not, sadly, point to the AST node itself. Each node in the AST of the parsed Haskell code also contains location information, so it is possible to traverse the AST until the correct node is found. At that point, the expression can be changed into an application of the coercion function.

haskell

```haskell
coerce :: HsExpr → HsExpr
coerce (HsApp e1 e2) | e1 == HsVar "coerce" → HsApp e1 (coerceApp e2)
coerce e = HsApp (HsVar "coerce") e
```

This only holds when the type error is fixable by surrounding it with a coercion, though. For function application, the error location can be at the head of the application, while the error actually occurs because of one of the applied expressions.

While it is possible to find the argument that causes the expression, to avoid inserting multiple coercions, this was not done for the sake of simplicity. Because GHC does not report the correct source location on which to insert a coercion, figuring this out would require implementing part of a typechecker, which this thesis aimed to avoid. The alternative, to iteratively insert a coercion at each argument until the correct argument is found, is deemed too expensive, as it would require many extra iterations, especially because the code generator often generates functions with many arguments.

4. Haskell Data types

Even if coercions are only applied when necessary, there are still many places in which coercions are needed because of the way the Haskell code is generated. While dependent types are hard to translate without using coercions, simpler datatypes which can be naturally defined in Haskell can be translated without coercions by slightly modifying the Haskell code generation. In particular, the translation of simple recursive data types can be improved to reduce the number of coercions needed while only changing the definition of data types, without having to change the way terms are translated.

Basic recursive data types, such as the natural numbers as defined in section 2.3, can be expressed in Haskell naturally without using any coercions. However, because the backend translates data types in a general manner, the types in the generated Haskell code are not accepted by GHC. Take the example of adding two natural numbers:

Haskell

```
data Nat a = Zero
           | Succ a

plus n m
  = case n of
      Zero → m
      Succ n' → Succ (plus n' m)
```

Running GHC on these programs produces errors about infinite types, as shown in figure 4.1. By adding coercions, replacing `Succ (plus n' m)` by `coerce Succ (plus (coerce n') m)`, it is possible to compile the program, which is what the current GHC backend does. However, these coercions are not needed at all if the *Nat* datatype is translated in the more natural way:

```
Plus.hs:25:22:
    Occurs check: cannot construct the infinite type: a0 = Nat a0
    In the return type of a call of ‘plus’
    In the first argument of ‘Succ’, namely ‘(plus n’ m)’
    In the expression: Succ (plus n’ m)

Plus.hs:25:27:
    Occurs check: cannot construct the infinite type: t0 = Nat t0
    In the first argument of ‘plus’, namely n’
    In the first argument of ‘Succ’, namely ‘(plus n’ m)’
    In the expression: Succ (plus n’ m)
```

Figure 4.1.: GHC does not like infinite types

```
data Nat = Zero
         | Succ Nat
```
Haskell

The types in the program will change slightly, but the *plus* function does not have to change at all. Note, however, that this translation only works on simple recursive data types without arguments. But because the terms do not have to change, this translation can be mixed seamlessly with the current translation, all the while letting GHC figure out which coercions are still needed.

4.1. A better generated data type

The translation of data types will be improved in several stages. Throughout these stages, we will use a running example to show how each stage influences the data type for a slightly more complicated data type:

```
data Vec {a} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (succ n)
```
Agda

This type is similar to the vector type seen in chapter 2: there is a constructor for the empty vector, and a constructor to cons an element to a vector. As before, a natural number is taken as an index, which is included in the type of the vector. This version of the type, however, also takes a *level* as parameter (the `{a}` parameter). This level parameter is then used as an argument of the next parameter (`Set a`).

While this version of the vector type looks slightly different from the one without the `Set a`, it should behave identically. However, for the sake of the data type translation, this requires some extra work in the translation for it to actually translate to the same Haskell data type.

The current translation looks like this:

```
data Vec0 n x xs = VNil | VCons n x xs
```
Haskell

To improve the generated data type, we will make the following three improvements, which will be more extensively discussed in the following sections:

1. Type paramaters in Agda data type declarations are mapped to type parameters in the generated Haskell declarations, but in contrast to the previous translation, these are the only parameters of the generated declaration.

   ```
   data Vec1 a set_a = VNil | forall n xs. VCons n set_a xs
   ```
   Haskell

2. When a constructor argument is a known Haskell type, the type is filled in, instead of passing it as a parameter.

   ```
   data Vec2 a set_a = VNil | VCons Nat set_a (Vec a set_a)
   ```
   Haskell

3. Remove any parameters that correspond to Agda levels, and remove level indirection when using them.

   ```
   data Vec3 a = VNil | VCons Nat a (Vec a)
   ```
   Haskell

While the original type `Vec0` needs many coercions to compile a function that uses vectors, the final type `Vec3` can often be used without needing any coercions at all. The following subsections will explain these steps in more detail.

4.1.1. Parameters

The Agda programming language, like most statically typed functional programming languages, supports polymorphic data types. These data types take one or more parameters, which are then used by their constructors. One simple example of a polymorphic data type is the list type:

Agda

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A
```

Here, the `A` parameter is used to be able to create a list of any type `A`.

When Agda is asked to compile this data type to Haskell, however, it will not use parameters in the same way. In the current translation, instead of using Agda parameters as Haskell parameters, all constructor arguments are handled as if they were parameters. So in this case, the `A` and `List A` arguments of the *cons* constructor are taken as parameters by the list datatype:

Haskell

```
data List a l = Nil | Cons a l
```

The main upside of this approach is that the translation is very simple: constructors do not define what type their arguments should be, and blindly take arguments of any type that are passed as parameters.

Even though the Agda compiler knows that the second argument to *cons* should be of type `List A`, it does not specify this in the constructor. This makes the translation straightforward, as the compiler does not have to worry about which arguments can be specified and which ones need to be passed as a parameter. Also, because the GHC typechecker can infer these types for us, constructors with ample type information are not needed to compile the program, as type inference, combined with added coercions, means that the exact argument types of constructors can be left for GHC to infer.

However, if the goal is to retain more type information at the Haskell level, this causes some problems. Because all constructor arguments are passed as parameters, filling in some arguments with their known type would mean that the parameters would be ignored. If the goal is to fill in as much constructor arguments as possible, then it makes sense to change the mechanism of passing type information through parameters.

Some constructor arguments are difficult to fill in, especially when dependent types are used. With dependent types, functions can be called inside a type, and arbitrary calculations can be required just to know the exact type. While type-level functions can be expressed in Haskell, this is not something that can easily be done in general.

To make it easier for constructor arguments to be filled in, the argument variables should be local to the constructors, instead of being passed as parameters. This can be done by quantifying over those variables, using existential quantification:

Haskell

```
data List a = Nil | forall l. Cons a l
```

This change moves the l parameter to inside the *Cons* constructor. Now, the Haskell definition matches the Agda definition in the number of parameters.

This also makes it easier to change the type of constructor arguments, without needing to change other data types which use lists in their constructors.

For the vector example, the number of parameters changes more drastically. Recall the definition of *Vec*:

Agda
```
data Vec {a} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (succ n)
```

Using the old method, there are 3 parameters: the indexed natural number `n`, the prepended element `x`, and the vector that is being consed to, `xs`:

Haskell
```
data Vec0 n x xs = VNil | VCons n x xs
```

Note that this ignores the {a} parameter. By following the parameters that are used in the Agda definition, the resulting Haskell data type gets two parameters:

Haskell
```
data Vec1 a set_a = VNil | forall n xs. VCons n set_a xs
```

Note that the old translation did not include these level parameters, as it took all constructor arguments as parameters, and ignored the parameters in the Agda definition.

The first parameter is not used in the Haskell definition, and will be erased in the third step. In this example, it is clear that the `n` has type ℕ, and `xs` is a vector. These will be filled in in the second step.

4.1.2. Known types

Now that the data types have the correct number of parameters, it is possible to improve the types of the constructors. Often, the types in a constructor are of a known type, which is always the same. Even vectors, which are indexed by their length, have two fixed parameters, because indices are not translated to parameters. So for the *cons* constructor, the recursive case can be filled in in the constructor argument:

Haskell
```
data Vec1 a set_a = VNil | forall n xs. VCons n set_a xs
data Vec2 a set_a = VNil | VCons Nat set_a (Vec a set_a)
```

Notice that the natural number is also trivial to fill in, as it is fixed and has zero parameters.

For vectors, filling in the constructors means that no quantified variables are left. However, in general, this is not always the case. For dependent types, the exact type that is used in the constructor may depend on a type-level computation, which is not translated to Haskell, and has to be left open with an existentially quantified variable.

4.1.3. Levels

The last step is to remove level indirection. Levels are often used in Agda, but are a feature that is important for the typechecker, but not for executing the program. Because of this, level indirection can be safely removed, reducing the number of parameters that many data types take.

In the case of vectors, this reduces the number of parameters to just one:

Agda

```
data Vec {a} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (succ n)

data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (succ n)
```

Figure 4.2.: Vector definitions with and without levels

Haskell

```
data Vec2 a set_a = VNil | VCons Nat set_a (Vec a set_a)
data Vec3 a = VNil | VCons Nat a (Vec a)
```

This last data definition does not mention levels. In particular, this means that the inclusion of levels make no difference for the runtime representation. As such, both data types in figure 4.2 are equivalent.

5. Benchmarks

To benchmark the changes made to the Agda compiler backend, it is useful to distinguish several versions of the compiler:

- **The old backend** The old Agda backend, which inserts a coercion at every location where it might be needed. This is the base case, upon which we hope to improve.
- **Stage 1: GHC-lib** The first stage uses GHC as a library to generate Haskell code, while mimicking the insertion of coercions at the same locations that the old backend did. While it might seem that this stage works the same as the old backend, it is important to measure any changes in performance that are caused by calling GHC as a library.
- **Stage 2: Error-based coercion insertion** The second stage uses the error-based insertion of coercions to reduce the number of coercions inserted to only those that are needed, while disabling any changes to how data types are translated to Haskell.
- **Stage 3: Improved data types** The final stage combines the error-based coercion insertion with the improvements in data type translation, and is meant to give the best results. However, as not all data types can be fully translated, especially those containing dependent types, this may not give the best results in all instances. As the changes in data types add more type information, this can cause some extra coercions, as an increase in type information also increases the number of locations where a coercion might be needed.

Figure 5.1 shows a comparison of these different versions.

5.1. Metrics

The changes to the Agda compiler aim to improve the code generation, which allows GHC to better optimize the generated code and produce a faster program. However, there is a potential trade-off between compile-time and run-time: to reduce the number of needed coercions, GHC needs to be run multiple times.

Compiler version	How GHC is called	Coercions inserted	Better datatypes
Old backend	Normal compilation	All	No
Stage 1	As a library	All	No
Stage 2	As a library	Where needed	No
Stage 3	As a library	Where needed	Yes

Figure 5.1.: Comparison of compiler versions

Each eliminated coercion hopefully speeds up the generated program, but might take extra time to compile because of repeated work in the GHC compiler.

To show the impact of the changes made to the backend, the following statictics are gathered:

- **Number of coercions** The central metric to measure is the number of coercions. While other metrics are important in their direct impacts on the compilation of Agda programs, it is important to know how many coercions are needed, as it shows how close the Haskell types are to the Agda types. The larger the number of coercions that can be eliminated, the better the type information is so that GHC can compile and optimize the program.
- **Number of iterations** Another important metric for error-based coercion insertion is the number of iterations. Generally, Haskell expressions that are independent of each other can be coerced in the same iteration. This includes different functions that do not call each other, and let bindings that can be calculated independently. However, when a Haskell module has deeply nested code, each nested expression may cause another type error, which is only detected when the expressions that are nested within are correctly coerced and typed. Each nested layer thus requires the backend to start a new iteration, which calls on GHC to typecheck the module again.
- **Runtime** Error-based insertion of coercions is a trade-off in multiple ways. By reducing the number of coercions used, the runtime of the compiled program is hoped to be lower, as the improved type information allows for more optimization opportunities.
- **Compile time** The reduced runtime comes at the cost of a potentially higher compile time. While the final GHC compilation may be faster, as fewer coercions means more complete type information to work with, this comes at the cost of extra time taken to insert coercions at error locations, and potentially multiple iterations for which typechecking must be done again in GHC.

To measure these trade-offs, the new backend will be compared using the compiler test suite and the Agda standard library.

5.2. Compiler test suite

To test whether the reduction in the number of coercions improves the performance of the Agda compiler, a collection of executable Agda programs is needed. By testing programs that can be executed, one can not only compare the number of coercions and compile-time, but also the runtime of the programs in relation to the number of coercions.

As Agda is not often used to create executable programs, but instead typically used as a theorem prover, it is difficult to find such a collection of Agda programs. Luckily, to test the Agda compiler itself, there is a test suite of executable Agda programs that can be used for benchmarking purposes[1]. The test

[1] `https://github.com/agda/agda/tree/master/test/Compiler`

suite consists of 60 testcases, and each testcase is a simple Agda program that can be compiled and executed.

For example, the *Sort* testcase generates the numbers 1200 down to 1, and then sorts it using a simple insertion sort:

Agda

```
insert : Nat → List Nat → List Nat
insert x [] = x :: []
insert x (y :: xs) = if x < y then x :: y :: xs else (y :: insert x xs)

sort : List Nat → List Nat
sort [] = []
sort (x :: xs) = insert x (sort xs)

...

main : IO Unit
main = mapM! printNat (sort (downFrom 1200))
```

Notice that even though sorting numbers could be done by just using datatypes for lists and numbers, the testcase uses the IO monad to map a print function over the list of sorted numbers, which results in a slightly more complicated generated program than might seem at first glance.

While this particular testcase specifically tests the runtime performance of the generated Haskell code, some of the testcases test Agda language features instead, and are more geared towards the earlier compiler phases such as type-checking. However, such tests are still useful when comparing the number of coercions, even when the runtime of the program might be very low.

5.3. Standard library

One downside of the Agda compiler test suite is that most of the testcases are small programs. They generally test some self-contained piece of code, without making use of the Agda standard library.

The Agda standard library itself[2] is helpful for testing the compiler improvements. While the standard library is not a single executable in itself, and as such cannot be used to compare runtime performance, it is very helpful for comparing the number of coercions and compile time, as the original backend inserts more than 23000 coercions to compile the whole standard library.

Most of the standard library is already tested in one of the testcases, which tries to cover a large part of the standard library by using several parts of it:

Agda

```
main = run (putStrLn "Hello World!") >>
       DivMod.main >>
       HelloWorld.main >>
       HelloWorldPrim.main >>
       ShowNat.main >>
       TrustMe.main >>
       Vec.main >>
       dimensions.main
```

[2] https://github.com/agda/agda-stdlib

However, just knowing that the whole of the standard library needs fewer coercions is not very useful. A more fine-grained approach is important here. Because the Agda standard library is already split into many Agda modules, and each corresponding compiled Haskell module has coercions inserted in turn, test results can be gathered for each individual module.

5.3.1. A more specific compile-time measurement

One additional upside to comparing by module is that some metrics can be more closely compared. Specifically, when the compile time is measured for the other test cases, it is measured for the whole compilation process, which includes the backend steps to typecheck generated Haskell code and insert coercions, but also the earlier compiler stages like the parsing and typechecking of the Agda program code.

Because the standard library is measured for each module individually, it is possible to limit the time measurement to only include the phase where GHC is repeatedly called through the GHC library API, and coercions are inserted at the reported error locations.

Note that this not just excludes the parsing and typechecking of the compiled Agda program, but also the parsing of the generated Haskell code. The Haskell code is always parsed only once, into a Haskell AST that is exposed by the GHC library, which is subsequently modified by the error-based coercion insertion until it compiles without type errors.

5.4. Results

Both the test suite and the standard library have been tested according to the metrics defined in section 5.1. As these consist of 60 tests and 99 modules respectively, we will not discuss each testcase or module separately. The full results can be found in the appendix. Some representative testcases are selected to show the general trends for each metric. After that, some problematic test cases will be explained in more detail.

5.4.1. Number of coercions and iterations

On average, the testcases used 77.8 coercions for the first stage. This was reduced to 2.2 coercions per testcase by using coercions only where needed, and was reduced further to just 0.8 coercions by improving the types that were used. Figure 5.2 also shows this reduction.

In fact, because many of the testcases are quite small in terms of code size, it was found that most of the test cases did not need any coercions at all. Indeed, 47 of 60 testcases did not need any coercions at all, even without changing the data types used. The following test cases are examples of these:

Test case	**#coe stage 1**	**#coe stage 2**	**#coe stage 3**
Arith	66	0	0
CompileNumbers	105	0	0
Records	78	0	0
Sort	88	0	0
String	69	0	0

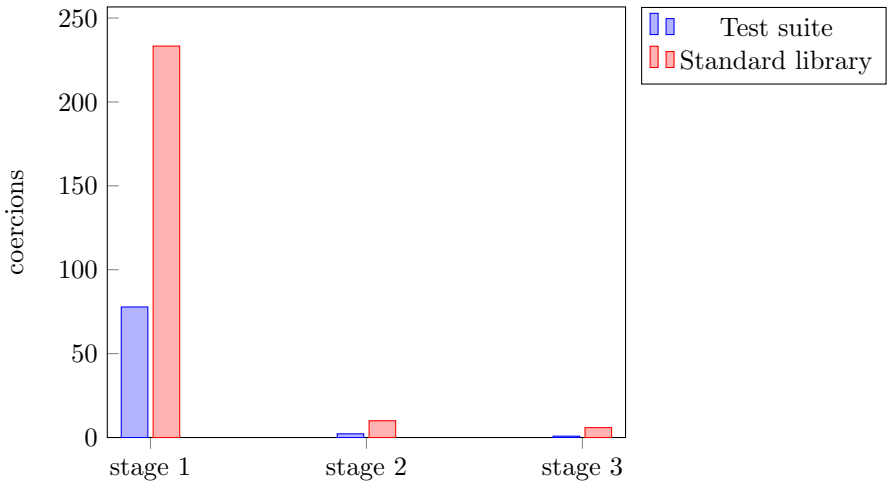

Figure 5.2.: Average number of coercions per testcase/module

These tables show the number of coercions that are inserted in each stage. If multiple iterations are required, the number of extra iterations is also shown between braces. As the generated Haskell code does not need any coercions in these testcases, the code is accepted on the first pass, and no additional iterations are required.

Other testcases have their required number of coercions lowered, but not eliminated entirely. The third stage, which uses improved data types, further reduces the number of coercions in some cases, while in other cases it makes no difference. In some rare cases, changing the data types even increases the number of coercions required, as an increased amount of type information can cause more type errors. However, the data type translation can be further improved, leading to further reductions in the number of coercions left over.

Test case	**#coe stage 1**	**#coe stage 2**	**#coe stage 3**
Coind	81	4 (1)	1 (1)
Issue2123	72	0	3 (1)
FlexibleInterpreter	92	24 (2)	24 (2)
Forcing	182	26 (4)	0
VaryingClauseArity	72	8 (2)	8 (2)
VecReverse	163	29 (4)	0

The number of iterations required for the second and third stages loosely follow the number of coercions; testcases that require many coercions also use more iterations to insert those coercions. Iterations are primarily required in deeply nested code, and as such are not directly a result of the length of the code.

For the standard library, the same pattern arises, as stage 2 eliminates most coercions. The total number of coercions needed was reduced from 23099 to 989. Improved types also help to further reduce the number of coercions in the modules that still require many coercions. This reduced the total number of coercions to just 584.

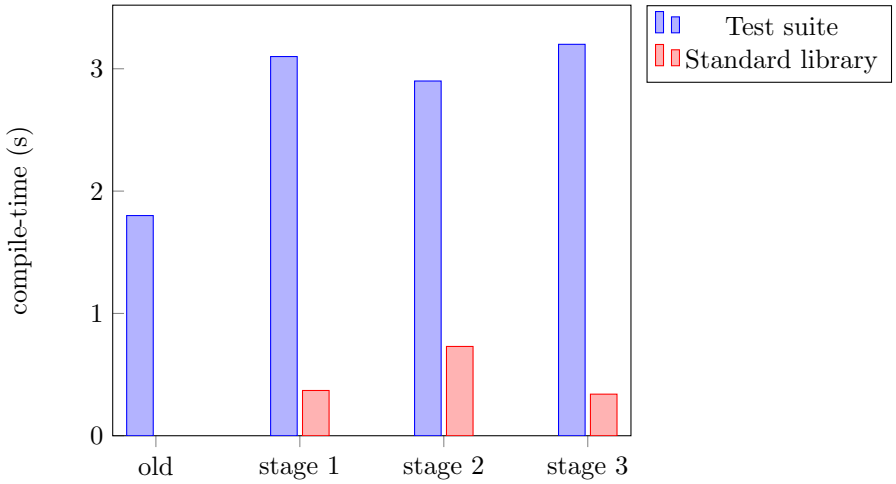

Figure 5.3.: Average compile-time per testcase/module

Test case	**stage 1**	**stage 2**	**stage 3**
Algebra	2747	0	77 (1)
Algebra.RingSolver.Lemmas	2083	228	0
Data.Colist	436	47 (3)	70 (3)
Relation.Binary	965	0	25 (1)
Relation.Binary.List.StrictLex	287	26 (3)	19 (2)

While the third stage actually re-introduces some coercions in some cases, fewer coercions are re-introduced than eliminated.

5.4.2. Compile time

To measure compilation times, it is important to compare only the changes in where coercions are used and how data types are changed, while taking into account the changes that are caused because of the way GHC is called. As such, the three stages are not only compared between themselves, but also to the old GHC backend.

Figure 5.3 shows a comparison of compilation times of 13 of the 14 executable testcases. The *VecReverseIrr* test is excluded, as the runtime of that test case is severely increased from 37 milliseconds to 26 seconds. This problematic test case will be discussed in section 5.5.1.

There is a clear increase in compilation time when GHC is called as a library, as the compilation times of all three stages are higher than those of the old GHC backend. This is possibly due to differences in compiler arguments or the different linking style. The way the current implementation uses the GHC library, all modules are compiled at least twice. This is because all modules have to be loaded before being able to insert coercions by modifying the syntax tree.

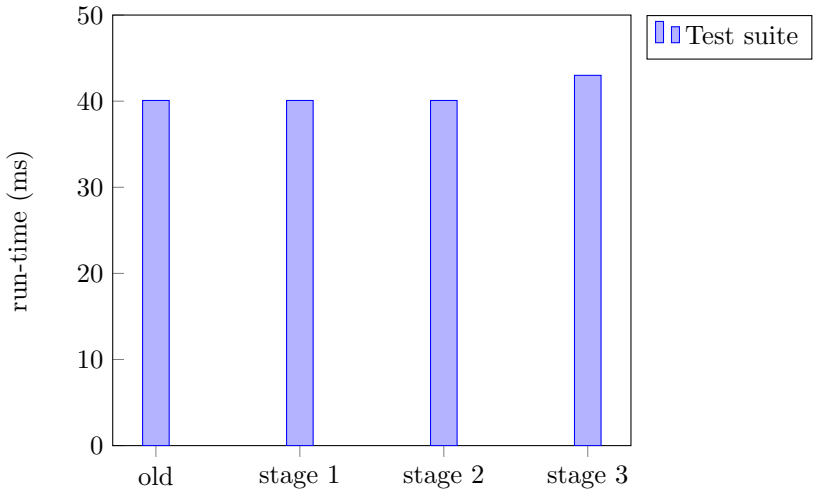

Figure 5.4.: Average run-time per testcase

Test case	**Old**	**stage 1**	**stage 2**	**stage 3**
Arith	1.67s	2.78s	2.61s	2.8s
Coind	1.85s	2.75s	2.22s	2.53s
CompileCatchAll	1.71s	2.71s	2.93s	3.05s
Records	1.87s	2.82s	2.68s	2.92s
Sort	1.98s	3.02s	3.57s	3.25s
VecReverse	2.61s	3.69s	2.72s	3.55s
VecReverseIrr	1.93s	3.16s	2.95s	3.13s

As error-based coercion insertion requires each module to be typechecked for each iteration, the second stage is slower than the first stage, in which all coercions are inserted beforehand and as such no extra iterations are required.

The third stage varies in compile-time compared to the second stage. This is because the difference in types may result in coercions in different locations, and as such influences the time taken to compile the program. In some instances, the third stage is faster than the first stage, because even though no extra iterations are needed, the increased amount of type information can simplify the typechecking process, speeding up the compilation.

5.4.3. Runtime

Because Agda is most often used as a theorem prover, there are relatively few executable Agda programs. This is also the case for the compiler testsuite: most testcases have no notable running time when executed, taking roughly one millisecond to complete. As such, those testcases have been excluded from the results. Still, 14 of the testcases are useful to compare the running time of compiled Agda programs.

Test case	Old	stage 1	stage 2	stage 3
PrimSeq	165ms	164ms	163ms	179ms
Sort	24ms	25ms	25ms	27ms
VecReverse	313ms	316ms	319ms	341ms
VecReverseIrr	38ms	37ms	26189ms	26174ms

As figure 5.4 shows, there is little difference in runtime in most cases. There are small differences between the old backend and the first stage, which are caused by slight differences in GHC compiler arguments. However, neither the old backend nor the first stage is consistently faster than the other.

Removing most coercions, as is done in the second stage, has barely any impact on the runtime. This has not resulted in the performance improvements that were a goal of this thesis. Coercions are designed to have little impact on the runtime, by using rewrite rules that attempt to erase coercions during compilation. The improved types, however, cause most testcases to run slightly slower.

5.5. Problematic test cases

5.5.1. VecReverseIrr

One very problematic test case is `VecReverseIrr`. While it takes only about 37 milliseconds to run in the first stage, with all coercions left intact, reducing the number of coercions used increases the runtime to over 26 seconds, a 700x increase.

This testcase first reverses a long vector and then sums its contents:

Agda

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : ∀ ..{n} → A → Vec A n → Vec A (suc n)

foldl : ∀ {A} {B : Nat → Set}
  → (∀ ..{n} → B n → A → B (suc n))
  → B 0 → ∀ ..{n} → Vec A n → B n
foldl {B = B} f z (x :: xs) = foldl {B = λ n → B (suc n)} f (f z x) xs
foldl f z [] = z

reverse : ∀ {A} ..{n} → Vec A n → Vec A n
reverse = foldl {B = Vec _} (λ xs x → x :: xs) []

downFrom : ∀ n → Vec Nat n
downFrom zero = []
downFrom (suc n) = n :: downFrom n

main : IO Unit
main = printNat (sum (reverse (downFrom 100000)))
```

The important part is the `foldl` function, which generates the following Haskell code if all coercions (stage 1) are inserted:

Haskell

```
foldl _a _b f b0 _n vec = foldl f b0 vec
foldl' f b0 vec = case coe vec of
    VNil → coe b0
    VCons x xs
        → coe foldl' (λ_ → coe f erased) (coe f erased b0 x) xs
```

While the first stage inserts 5 coercions in this example, the second stage eliminates all of those.

The last coercion in this code fragment is the cause of this big increase in runtime. The call $(f\ z\ x)$, which recurses over the vector, is translated to $(coe\ f\ erased\ b0\ x)$. The program still compiles correctly when the coercion is removed, but the resulting program behaves differently, as the recursive call is used directly instead of lazily. Because the laziness is removed in this case, the direction of the fold operation is changed, causing it to use much more memory and runtime.

As Haskell programs are evaluated lazily, whereas Agda programs are evaluated in normal order, it is not straightforward to decide on laziness in the generated Haskell code. There are some solutions to automatically decide where to use laziness, such as the `AUTOBAHN` system [12], which uses genetic algorithms to infer strictness annotations. Other approaches, such as using profiling to decide on the locations in which to be lazy [4], may also be useful to improve runtime performance.

5.5.2. BooleanAlgebra.Expression

The `BooleanAlgebra.Expression`[3] module in the Agda standard library is problematic in stage 2, as it takes 56 extra iterations, just to insert 99 coercions. The culprit is the `lift` function, which creates a very large, nested Agda record, as shown in figure 5.5. Each ... in the figure denotes another solve expression, like the one shown for `∨-comm`.

While such a record can be compiled perfectly fine in Agda, it becomes problematic when compiled to Haskell. The Agda compiler translated these records into one big expression, with large nested function applications. If this expression compiles without coercions, as is the case in the third stage, then it is compiled efficiently.

However, because this expression needs coercions as to type-check correctly in the second stage, GHC only finds the outermost type error in any one iteration. As a result, after 4 iterations of inserting multiple coercions at once, the backend then spends 52 more iterations, inserting only a single coercion each time.

To avoid this problem, there are two solutions. Either the code generation has to be changed to avoid translating records to nested expressions, or the programmer should avoid nesting large expressions in records, opting to move those expressions to their own top-level functions.

[3] `http://www.cse.chalmers.se/~nad/listings/lib/Algebra.Props.BooleanAlgebra.Expression.html`

Agda

```
isBooleanAlgebra = record
  { isDistributiveLattice = record
    { isLattice = record
      { isEquivalence = PW.isEquivalence isEquivalence
      ; ∨-comm        = λ _ _ → ext λ i →
                          solve i 2 (λ x y → x or y , y or x)
                                (∨-comm _ _) _ _
      ; ∨-assoc       = ...
      ; ∨-cong        = ...
      ; ∧-comm        = ...
      ; ∧-assoc       = ...
      ; ∧-cong        = ...
      ; absorptive    = ...
      }
    ; ∨-∧-distrib^r = ...
    }
  ; ∨-complement^r = ...
  ; ∧-complement^r = ...
  ; ¬-cong        = ...
  }
```

Figure 5.5.: Part of the `lift` function

6. Related work

6.1. Searching for Type-Error Messages

The error-based coercion insertion described in chapter 3 is not a novel approach. In the SEMINAL system [7], error messages in the Caml language are improved by searching for similar expressions that are type-correct. This is done by treating the compiler as a black box, and searching for some interesting errors. The system then tries to replace some expressions, in the hope of producing well-typed modifications.

The approach taken by the SEMINAL system is very similar to our approach, in that it takes an ill-typed input AST, modifies some locations and then uses a typechecker to see if the changed program is well-typed. Some more sophisticated search methods are also tried [8], in particular some methods to handle multiple errors in a better way by selecting the most important ones. These extensions could also be applied to our error-based coercion insertion.

6.2. Partial types

The idea of assigning types to a partially typed language is not new. Soft typing [3] is a generalization of *static* and *dynamic* type systems, using static type information to check well-typed parts of the program, which also helps with generating more efficient code. In dynamic parts of the code, runtime checks are injected to check for type errors at runtime. Coercions resemble these checks, and while they are inserted for a different reason, both coercions and run-time checks are similar in that it is necessary to limit their number as much as possible.

While soft typing is a general framework, this idea has also been applied to functional languages in the form of Gradual typing [11]. The λ-calculus can be extended to allow optional type annotations, creating type-safe programs in the case of fully-annotated terms. Similar to the types shown in section 2.4, each term is assigned a type which may be partially unknown. However, similar to the soft types mentioned earlier, the source language may contain errors which have to be caught at run-time, while the GHC backend has to add coercions in those places.

6.3. Dependent Types in Haskell

For simple, Haskell-like types, the coercion can just be left out. But for more complicated Agda constructs, for example dependent types and type-level functions, Haskell has no direct alternative. As such, a translation is needed to describe or embed these constructs. However, as opposed to the recursive data

types shown before, translating dependent types to Haskell requires a different translation of both types and terms.

Section 6.3.1 will introduce type-level functions, which allow functions to be lifted to the type level. To allow terms to access type-level information in Haskell, *singletons* are needed, which are described in section 6.3.2.

6.3.1. Type-level computations

The translation of Agda terms to Haskell terms loses all type-level computations, because they are not needed during runtime execution, only for compile-time type safety. However, in the effort of restoring types in the generated Haskell code, it might be beneficial to encode Agda's type-level computations in Haskell types. An example of such a computation occurs when defining the vector append function *vappend*, as defined before in figure 2.2. The result type of this function sums the lengths of the two input vectors. However, the current translation omits the calculation altogether.

To do this calculation at the type level, the *plus* function has to be lifted to the type level. This can be done in Haskell using type families. Given a normal function, a corresponding type family can be defined which is usable at the type level:

Haskell

```
plus :: Nat → Nat → Nat
plus Zero m = m
plus (Succ n) m = Succ (plus n m)

type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus 'Zero m = m
type instance Plus ('Succ n) m = 'Succ (Plus n m)
```

When used together with singletons, many Agda terms can be translated to Haskell terms. One such example is the *vappend* function, which adds the length of two lists at the type level:

Haskell

```
vappend :: Vec a n → Vec a m → Vec a (Plus n m)
vappend VNil v2 = v2
vappend (VCons h t) v2 = VCons h (vappend t v2)
```

6.3.2. Singletons

In Agda, dependent types can use values at the type level. This feature is missing from Haskell, as it does not support dependent types. They can be simulated, though, with *data type promotion*. Take for example the natural numbers datatype:

Haskell

```
data Nat = Zero | Succ Nat
```

The *(DataKinds)* extension allows for the promotion of constructors to types, and promotes types to kinds. Which means that for this datatype, *Nat* can be used as a kind, and *'Zero* denotes the type corresponding to the constructor *Zero*.

However, by promoting values to types, we lose the value-level information. To use values at both the value and type level, *singleton types* are used, types with only one non-$\bot$ value [5]. Because there is only one way to construct a value of a singleton type, both the term and the type hold the same information. It is similar to a phantom type, but with the possibility of pattern matching on it at the value level.

An example is the *singleton* type for the *Nat* datatype defined earlier:

Haskell

```
data SNat :: Nat → ∗ where
  SZero :: SNat 'Zero
  SSucc :: ∀(n :: Nat). SNat n → SNat ('Succ n)
```

This allows for type-safe functions on dependent types, for example *vreplicate*:

Haskell

```
data Vec :: ∗ → Nat → ∗ where
  VNil :: Vec a 'Zero
  VCons :: a → Vec a n → Vec a ('Succ n)

vreplicate :: SNat n → a → Vec a n
vreplicate SZero _ = VNil
vreplicate (SSucc m) a = VCons a (vreplicate m a)
```

The singleton `SNat n` is used to construct the type `Vec a n`, but also used to pattern-match on.

6.4. Other compilers and backends

6.4.1. The Epic backend

The Agda compiler also has a backend which translates Agda to Epic. The Epic programming language [6] is similar to Haskell, but differs in that it is strict, and that it is especially created to be used as a backend for dependently typed languages. Type annotations can be used in Epic programs, but they are not checked or used while compiling, easing the translation of dependent types.

Epic code is compiled down to the C programming language, and has multiple optimisations in the translation process. Because the target language is not a functional language, several optimisations are necessary to ensure good performance. These optimisations include erasure of unneeded arguments, simplification of constructors when there is only a single way to return a value, and an efficient implementation of primitive data like natural numbers. The backend does not use such optimisations internally, but relies on GHC to perform them.

6.4.2. Idris

Idris [2] is a different dependently typed programming language, which is influenced by Haskell. It is different from Agda, however, in that it is primarily a general-purpose compiled language, whereas Agda is more geared towards theorem-proving than general-purpose features like I/O.

An important feature of Idris is its *phase distinction*. While some programming languages separate types from terms, and erase those types before execution, Idris also separates *compile time* and *run time* terms. Conventionally, types are automatically *compile time* terms. With dependent types in the

mix, however, this separation is harder, because terms might be used in types. Therefore, the *Idris* compiler tries to prove that some values are unused during run-time, so that it can erase those terms.

6.4.3. Coq

The Coq proof assistant [9] features an extraction mechanism which is able to turn Coq proofs and functions into runnable programs. There is a distinction between Coq programs and proofs. Coq programs are declared in *Set*, and are mostly simply typed. Proofs, on the other hand, are declared in the logical *Prop*, and can be extracted so that the program can be run.

Coq can generate ML programs during extraction, but also allows for other languages like Scheme and Haskell. Because the generated code might not type-check, a type-checker checks the generated code, and adds coercions at the points where the code does not type-check.

7. Conclusion

7.1. Summary

By changing the GHC backend of the Agda compiler, the goal of this thesis is to reduce the number of coercions used in the generated Haskell program. Because most coercions that were initially placed are not needed to compile the generated programs, GHC can be used to find locations where coercions are really needed. This results in a large reduction in the number of coercions used.

Even though the lower number of coercions does not lead to improvements in compile- or run-time, improvements in the generated Haskell code by the Agda compiler make it easier to interface with compiled Agda programs from Haskell.

7.2. Future work

7.2.1. Better Haskell code generation

By limiting the number of locations in Haskell expressions that coercions are inserted, the generated Haskell code is improved, in the way that it more closely follows the Agda code of the to be compiled Agda program. This thesis has made a deliberate attempt to avoid modifying the way that the code generation works, and instead opted to work with what is currently in place. The changes to data types are similar, in that they only change data type definitions, while keeping the structure of Haskell expressions the same.

While this approach helps make these changes easier to implement without adding more complexity, it also has a major shortcoming. As the code generation of Agda's GHC compiler backend has been originally implemented with the idea that coercions are used everywhere, the generated code works well under the assumption that every expression is coerced. However, when removing some of those coercions, that assumption falls away, and some problems arise.

As the approach of error-based coercion insertion uses type error locations to insert coercions, deeply nested expressions are troublesome as each type error only shows itself when the expressions nested below are well-typed. This problem can be alleviated by changing the Haskell code generation, so that nested expressions are instead translated using let-bindings, or creating separate functions for deeply nested expressions, or some other mechanism to make the generated Haskell program easier to compile.

Another aspect that can be improved by changing Haskell code generation is the translation of dependent types. While most coercions are eliminated using error-based coercion insertion, there are still instances where coercions are needed. Many of these are needed when dependent types are used, as they are not directly translatable to Haskell. By extending the translation to handle dependent types better, many if not all coercions that are as of yet needed can be eliminated.

7.2.2. Haskell interoperability

As the Agda compiler can compile to a Haskell program, it is possible to use functions defined in Agda from within a Haskell program. The Haskell code that the compiler outputs using the GHC backend code generation, however, is difficult to work with. The code examples so far in this thesis have had their variable names changed, and several noisy parts left out, as to improve readability. To demonstrate this, see for example the foldr_1 function from the standard library:

Haskell

```
name286 = "Data.Vec.foldr\8321"
d286 v0 v1 v2 v3 v4 = du286 v3 v4
du286 v0 v1
  = case v1 of
      C22 v2 v3 v4
        → case v4 of
            C14 → v3
            C22 v5 v6 v7 → v0 v3 (du286 v0 v4)
            _ → MAlonzo.RTE.mazUnreachableError
      _ → MAlonzo.RTE.mazUnreachableError
```

There are several problems with this generated Haskell code:

- The function (d286) takes 5 arguments (v0 to v4), but immediately drops 3 of them, and calls another function which just takes the required arguments (du286). This clutters the types of the functions, and makes it harder to call them from a non-generated Haskell program. These unused arguments could be removed during the code generation, instead opting to just use the arguments that are really used.

- The function name (d286) is cryptic, and does not resemble the original Agda function name (foldr_1). The name is given as a separate string variable, and uses the unicode escape sequence for any non-ascii symbols (which are used very often in most Agda code). While Haskell does not support arbitrary unicode symbols in function names, having the generated names be closer to the original Agda function names would ease interoperability from Haskell programs. While the programmer can use Pragma annotations to change the names of compiled functions, generating better default names would ease the use of Agda functions from Haskell.

- Similarly, the constructor names for vectors in this example are C14 and C22. It would be better to name them something closer to their original Agda definition, like *VNil* and *VCons*.

- Lastly, the code generation inserts statements to denote parts of the program that should be unreachable. While this could be useful to debug a compiler error, having these statements inserted during code generation just increases code length, without doing anything useful. In this example, the checks are entirely superfluous, as vectors have only two constructors, meaning that these statements are unreachable.

7.2.3. GHC as a library

By using GHC as a library instead of a separate program, we can gain access to the internal GHC representation of a parsed Haskell program. This allows coercions to be inserted without requiring a restart of the GHC compilation process. However, in the current implementation of the error-based coercion insertion, each Haskell file is still compiled at least twice, which leads to an increase in compilation time.

It is unclear whether this is a defect of the usage of the GHC library, or that it is a shortcoming of the GHC API. As the GHC library exposes GHC internals, and is poorly documented, it is difficult to work with. Stil, improving the interface with GHC could remove the increase in compilation time, bringing the compile-times of the three stages in line with the original backend.

Bibliography

[1] Marcin Benke. Alonzo—a compiler for agda. In *Talk at Agda Implementors Meeting*, volume 6, 2007.

[2] Edwin Brady. Idris: general purpose programming with dependent types. In *PLPV*, pages 1–2, 2013.

[3] Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN Notices*, volume 26, pages 278–292. ACM, 1991.

[4] Stephen Chang and Matthias Felleisen. Profiling for laziness. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM, 2014.

[5] Richard A Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *ACM SIGPLAN Notices*, 47(12):117–130, 2013.

[6] Olle Fredriksson and Daniel Gustafsson. A totaly epic backend for agda. 2011.

[7] Benjamin Lerner, Dan Grossman, and Craig Chambers. Seminal: searching for ml type-error messages. In *Proceedings of the 2006 workshop on ML*, pages 63–73. ACM, 2006.

[8] Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM SIGPLAN Notices*, volume 42, pages 425–434. ACM, 2007.

[9] Pierre Letouzey. Extraction in coq: An overview. In *Logic and Theory of Algorithms*, pages 359–369. Springer, 2008.

[10] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[11] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[12] Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. Autobahn: using genetic algorithms to infer strictness annotations. In *Proceedings of the 9th International Symposium on Haskell*, pages 114–126. ACM, 2016.

A. Full results

A.1. Number of coercions

Test case	stage 1	stage 2	stage 3
Arith	66	0	0
BuiltinInt	89	0	0
CaseOnCase	90	0	0
CatchAllVarArity	94	0	0
Coind	81	4 (1)	1 (1)
CompareNat	90	0	0
CompileAsPattern	82	0	0
CompileCatchAll	75	0	0
CompileNumbers	105	0	0
CompiledRecord	71	0	0
CompilingCoinduction	40	0	0
CompilingQNamePats	86	0	0
CopatternRecord	80	0	0
CopatternStreamSized	91	6 (1)	2 (2)
EraseRefl	72	0	0
FlexibleInterpreter	92	24 (2)	24 (2)
Floats	101	0	0
FloatsUHCFails	79	0	0
Forcing	182	26 (4)	0
Forcing2	79	0	0
Forcing3	84	0	0
Forcing4	120	8 (2)	0
HelloWorld	59	0	0
IdentitySmashing	67	2 (1)	0
InlineRecursive	64	0	0
Irrelevant	62	0	0
Issue1441	62	0	0
Issue1486	60	0	0
Issue1496	58	0	0
Issue1624	72	0	0
Issue1632	70	0	1 (1)
Issue1664	86	0	0
Issue1855	61	0	0
Issue2123	72	0	3 (1)
Issue2218	86	0	0
Issue2222	61	0	0
Issue326	60	0	0
Issue561	61	0	0
Issue727	65	4 (2)	4 (2)
Issue728	68	0	0
Literals	64	0	0
ModuleArgs	60	0	0
ModuleReexport	71	0	0
Mutual	91	5 (2)	0
NoRecordConstructor	77	0	0
PrimSeq	75	0	0
PrintBool	82	5 (1)	5 (1)
QNameOrder	65	0	0
Records	78	0	0
Sort	88	0	0
StaticPatternLambda	73	0	0
String	69	0	0
StringPattern	64	0	0
TrailingImplicits	64	0	0
UniversePolymorphicIO	10	0	0
UnusedArguments	78	0	0
VaryingClauseArity	72	8 (2)	8 (2)
VecReverse	163	29 (4)	0
VecReverseIrr	80	4 (1)	0
WfRec	103	5 (1)	0

A.2. Coercions in standard library

Test case	stage 1	stage 2	stage 3
Agda.Builtin.Bool	1	0	0
Agda.Builtin.Char	1	0	0
Agda.Builtin.Coinduction	1	0	0
Agda.Builtin.IO	1	0	0
Agda.Builtin.List	1	0	0
Agda.Builtin.Nat	1	0	0
Agda.Builtin.String	1	0	0
Agda.Primitive	1	0	0
Algebra	2747	0	77 (1)
Algebra.Operations	827	0	0
Algebra.Properties.AbelianGroup	161	0	0
Algebra.Properties.BooleanAlgebra	1578	0	0
Algebra.Properties.BooleanAlgebra.Expression	805	99 (56)	22 (1)
Algebra.Properties.DistributiveLattice	182	0	0
Algebra.Properties.Group	185	0	0
Algebra.Properties.Lattice	334	0	0
Algebra.Properties.Ring	450	0	0
Algebra.RingSolver	3058	0 (4)	8 (2)
Algebra.RingSolver.AlmostCommutativeRing	764	0	15 (1)
Algebra.RingSolver.Lemmas	2083	228	0
Algebra.Structures	1337	0	31 (1)
AllStdLib	12	1 (1)	0
Category.Applicative.Indexed	54	0	3 (1)
Category.Functor	5	0	1 (1)
Category.Monad.Identity	3	0	0
Category.Monad.Indexed	162	1 (1)	4 (1)
Data.Bool	2	0	0
Data.Bool.Base	31	0	0
Data.Bool.Properties	639	144 (1)	0
Data.BoundedVec.Inefficient	20	4 (1)	0
Data.Char	10	0	0
Data.Char.Base	1	0	0
Data.Colist	436	47 (3)	70 (3)
Data.Conat	47	11 (2)	23 (2)
Data.Digit	98	5 (1)	10 (1)
Data.Empty	1	0	0
Data.Fin	170	40 (3)	2 (1)
Data.Fin.Dec	154	23 (2)	0
Data.Fin.Subset	30	4 (2)	0
Data.Fin.Subset.Properties	165	21 (3)	1 (1)
Data.Integer.Base	67	0	0
Data.List.Any	122	11 (1)	14 (2)
Data.List.Base	207	2 (1)	3 (1)
Data.List.NonEmpty	164	16 (3)	17 (3)
Data.Maybe.Base	30	0	0
Data.Nat	72	6 (1)	0
Data.Nat.Base	84	1 (1)	0
Data.Nat.DivMod	52	0	0
Data.Nat.Properties	676	28 (2)	0
Data.Nat.Show	9	2 (1)	2 (1)
Data.Plus	34	11 (1)	6 (2)
Data.Product	35	0	1 (1)
Data.Sign	19	0	0
Data.String	15	0	0
Data.String.Base	8	0	0
Data.Sum	16	0	0
Data.Vec	214	38 (3)	10 (1)
Data.Vec.Equality	58	12 (2)	0
Data.Vec.NZ45Zary	92	25 (2)	14 (2)
Data.Vec.Properties	194	45 (3)	11 (2)
DivMod	20	2 (1)	0
Foreign.Haskell	1	0	0
Function	20	0	0
Function.Bijection	110	0	3 (1)
Function.Equality	76	0	2 (1)
Function.Equivalence	44	0	2 (1)
Function.Injection	19	0	2 (1)
Function.Inverse	186	0	5 (1)
Function.LeftInverse	172	0	3 (1)
Function.Related	317	62 (5)	53 (1)
Function.Surjection	82	0	4 (1)
HelloWorld	4	1 (1)	0
HelloWorldPrim	4	0	0
Induction	5	0	0
Induction.WellFounded	8	0	0
IO	99	8 (2)	8 (2)
IO.Primitive	1	0	0
Level	4	0	0
Qdimensions	246	5 (1)	5 (1)
Relation.Binary	965	0	25 (1)
Relation.Binary.Consequences	59	0	1 (1)
Relation.Binary.Consequences.Core	6	0	4
Relation.Binary.Core	20	0	4 (1)
Relation.Binary.Indexed.Core	22	0	4 (1)
Relation.Binary.Lattice	933	0	26 (1)
Relation.Binary.List.Pointwise	143	21 (2)	48 (2)
Relation.Binary.List.StrictLex	287	26 (3)	19 (2)
Relation.Binary.On	121	0	0
Relation.Binary.PreorderReasoning	23	0	1 (1)
Relation.Binary.PropositionalEquality	31	0	0
Relation.Binary.PropositionalEquality.Core	4	0	0
Relation.Binary.Reflection	56	4 (1)	4 (1)
Relation.Binary.Vec.Pointwise	147	23 (2)	4 (1)
Relation.Nullary	1	0	0
Relation.Nullary.Decidable	34	0	0
Relation.Nullary.Negation	33	0	0
ShowNat	4	1 (1)	0
TrustMe	11	1 (1)	0
Vec	54	10 (1)	12 (2)

A.3. Compile times

Test case	Old	stage 1	stage 2	stage 3
Arith	1.67s	2.78s	2.61s	2.8s
BuiltinInt	1.83s	2.81s	2.9s	2.76s
CaseOnCase	1.77s	2.91s	2.83s	2.74s
CatchAllVarArity	1.68s	2.88s	2.71s	2.73s
Coind	1.85s	2.75s	2.22s	2.53s
CompareNat	1.96s	3s	2.93s	3.08s
CompileAsPattern	1.74s	2.99s	2.91s	3.14s
CompileCatchAll	1.71s	2.71s	2.93s	3.05s
CompileNumbers	1.88s	3.08s	3.01s	3.71s
CompiledRecord	1.69s	2.92s	2.7s	4.43s
CompilingCoinduction	1.29s	1.96s	1.94s	2.1s
CompilingQNamePats	2.21s	3.43s	3.46s	4.96s
CopatternRecord	1.67s	2.76s	2.95s	3.88s
CopatternStreamSized	1.95s	3.24s	2.71s	2.97s
EraseRefl	1.81s	2.89s	3.04s	4.17s
FlexibleInterpreter	1.87s	3.23s	2.48s	2.99s
Floats	2.2s	3.76s	3.42s	4.29s
FloatsUHCFails	1.9s	3.16s	2.85s	4.08s
Forcing	2.07s	3.5s	2.24s	3.29s
Forcing2s	1.77s	2.84s	2.91s	2.8s
Forcing3s	1.76s	3.02s	2.93s	2.95s
Forcing4s	1.95s	3.45s	2.31s	3.91s
HelloWorld	1.67s	2.83s	2.8s	2.87s
IdentitySmashing	1.72s	3.29s	2.29s	3.15s
InlineRecursive	1.78s	2.76s	2.8s	2.93s
Irrelevant	1.65s	2.74s	2.75s	2.96s
Issue1441s	1.82s	2.8s	2.67s	2.81s
Issue1486s	1.77s	3.16s	2.96s	3.08s
Issue1496s	1.65s	2.79s	2.89s	3.08s
Issue1624s	1.69s	2.99s	2.9s	3.16s
Issue1632s	1.66s	3.02s	2.81s	2.98s
Issue1664s	2.26s	3.88s	3.93s	4.32s
Issue1855s	1.69s	2.91s	2.8s	3.06s
Issue2123s	1.75s	2.94s	2.74s	2.53s
Issue2218s	2.19s	4.24s	3.64s	3.88s
Issue2222s	1.69s	3.2s	2.61s	2.69s
Issue326s	1.7s	3.49s	2.78s	2.92s
Issue561s	1.82s	3.15s	2.8s	2.87s
Issue727s	1.83s	2.95s	2.51s	2.5s
Issue728s	1.63s	2.86s	2.45s	2.67s
Literals	1.64s	3.39s	2.63s	2.85s
ModuleArgs	1.72s	3.35s	2.8s	2.96s
ModuleReexport	1.74s	3s	2.78s	3.35s
Mutual	1.68s	3.16s	2.82s	4.32s
NoRecordConstructor	1.71s	3.11s	2.74s	3.48s
PrimSeq	2.04s	3.16s	2.97s	3.18s
PrintBool	1.78s	2.95s	2.3s	2.35s
QNameOrder	2.23s	3.6s	3.75s	3.82s
Records	1.87s	2.82s	2.68s	2.92s
Sort	1.98s	3.02s	3.57s	3.25s
StaticPatternLambda	1.77s	3.18s	4.15s	3.38s
String	1.93s	2.96s	3.3s	3.25s
StringPattern	1.86s	2.83s	3.5s	3.22s
TrailingImplicits	1.7s	2.72s	3.15s	3s
UniversePolymorphicIO	1.01s	1.58s	1.86s	1.82s
UnusedArguments	1.77s	3.24s	3.06s	3.32s
VaryingClauseArity	1.68s	2.95s	3.3s	2.4s
VecReverse	2.61s	3.69s	2.72s	3.55s
VecReverseIrr	1.93s	3.16s	2.95s	3.13s
WfRec	1.78s	3.23s	2.39s	2.98s

A.4. Standard library compile times

Test case	stage 1	stage 2	stage 3
Agda.Builtin.Bool	10.8ms	10.4ms	10.8ms
Agda.Builtin.Char	26ms	26.2ms	26.6ms
Agda.Builtin.Coinduction	10.3ms	9.8ms	10.4ms
Agda.Builtin.IO	8.8ms	8.5ms	8.5ms
Agda.Builtin.List	13ms	12.4ms	12.9ms
Agda.Builtin.Nat	23ms	23.2ms	24.2ms
Agda.Builtin.String	52.7ms	51.6ms	51.7ms
Agda.Primitive	11.4ms	11.3ms	11.7ms
Algebra	3155.8ms	4647.3ms	3726.5ms
Algebra.Operations	803.7ms	917.9ms	620.5ms
Algebra.Properties.AbelianGroup	199.3ms	255.3ms	201.3ms
Algebra.Properties.BooleanAlgebra	3846.7ms	5269.4ms	3451.4ms
Algebra.Properties.BooleanAlgebra.Expression	756.8ms	11015.1ms	1489.6ms
Algebra.Properties.DistributiveLattice	299.1ms	327.4ms	257.4ms
Algebra.Properties.Group	220.9ms	266.2ms	218.9ms
Algebra.Properties.Lattice	406.8ms	452.3ms	418.7ms
Algebra.Properties.Ring	1484.2ms	680.1ms	363.7ms
Algebra.RingSolver	6149.3ms	23603.3ms	5560.5ms
Algebra.RingSolver.AlmostCommutativeRing	859.7ms	1244.9ms	1640.6ms
Algebra.RingSolver.Lemmas	1999.6ms	3888.1ms	1039.7ms
Algebra.Structures	2772.3ms	2618.4ms	1840.5ms
AllStdLib	26.6ms	32.5ms	28ms
Category.Applicative.Indexed	309ms	182.4ms	300.4ms
Category.Functor	15.7ms	15.4ms	20.4ms
Category.Monad.Identity	12.1ms	12.7ms	11.1ms
Category.Monad.Indexed	678.7ms	451.2ms	423.9ms
Data.Bool	8.3ms	8.5ms	7.8ms
Data.Bool.Base	27.9ms	26.1ms	25.1ms
Data.Bool.Properties	595.7ms	1128.8ms	558.3ms
Data.BoundedVec.Inefficient	26ms	29.5ms	24.6ms
Data.Char	22.1ms	34.6ms	31.7ms
Data.Char.Base	8.9ms	8.6ms	8.5ms
Data.Colist	773.1ms	1028ms	904.4ms
Data.Conat	62.8ms	65.8ms	65.1ms
Data.Digit	105.8ms	133.3ms	105.7ms
Data.Empty	8.5ms	8.6ms	8.4ms
Data.Fin	163ms	223.6ms	152.4ms
Data.Fin.Dec	173.9ms	287.9ms	136.8ms
Data.Fin.Subset	70.5ms	459.7ms	51.5ms
Data.Fin.Subset.Properties	1242.8ms	743.5ms	206.1ms
Data.Integer.Base	66ms	58.7ms	58.2ms
Data.List.Any	295.1ms	317.2ms	261.3ms
Data.List.Base	227.7ms	219.7ms	220ms
Data.List.NonEmpty	219ms	257.3ms	243.7ms
Data.Maybe.Base	58.9ms	53.3ms	53ms
Data.Nat	62.4ms	77.4ms	54.5ms
Data.Nat.Base	115ms	115.5ms	98.6ms
Data.Nat.DivMod	81.8ms	66.8ms	65.7ms
Data.Nat.Properties	641ms	995.3ms	541.7ms
Data.Nat.Show	50.9ms	50ms	49.6ms
Data.Plus	63.5ms	70.1ms	60.8ms
Data.Product	82ms	63.2ms	71.3ms
Data.Sign	20.1ms	19ms	18.7ms
Data.String	43.8ms	57.4ms	115.7ms
Data.String.Base	17.8ms	167.1ms	164.6ms
Data.Sum	39.5ms	34.6ms	34.5ms
Data.Vec	253.1ms	362.1ms	248.7ms
Data.Vec.Equality	98ms	125.6ms	79.9ms
Data.Vec.NZ45Zary	157.4ms	178.2ms	154.7ms
Data.Vec.Properties	289.8ms	410.2ms	290.7ms
DivMod	28.4ms	187ms	181.7ms
Foreign.Haskell	9.1ms	9.3ms	9.2ms
Function	53.9ms	44.7ms	44ms
Function.Bijection	134.4ms	133.3ms	141.8ms
Function.Equality	108.8ms	121.1ms	101.2ms
Function.Equivalence	99.5ms	84ms	66.4ms
Function.Injection	45.4ms	37.2ms	37.9ms
Function.Inverse	241ms	208ms	284.2ms
Function.LeftInverse	173.5ms	162.4ms	161.8ms
Function.Related	323.1ms	577.3ms	343.7ms
Function.Surjection	112.3ms	101.9ms	166.2ms
HelloWorld	12.2ms	14.3ms	11.9ms
HelloWorldPrim	10.9ms	12.1ms	12.1ms
Induction	22.1ms	20.3ms	21ms
Induction.WellFounded	53.1ms	47.5ms	45.9ms
IO	161.6ms	147.7ms	146.4ms
IO.Primitive	60.9ms	80.3ms	80.5ms
Level	9.9ms	10ms	10.1ms
Qdimensions	267.2ms	466ms	425.3ms
Relation.Binary	1132.6ms	1237.2ms	1963.2ms
Relation.Binary.Consequences	114.6ms	800.3ms	100.6ms
Relation.Binary.Consequences.Core	14.9ms	11ms	10.6ms
Relation.Binary.Core	55.9ms	56.5ms	65.4ms
Relation.Binary.Indexed.Core	42.1ms	42ms	53.5ms
Relation.Binary.Lattice	1144.6ms	1017.6ms	1027.8ms
Relation.Binary.List.Pointwise	191.8ms	365.1ms	230ms
Relation.Binary.List.StrictLex	354.9ms	579.2ms	385.6ms
Relation.Binary.On	242.3ms	222.7ms	147.1ms
Relation.Binary.PreorderReasoning	28.1ms	31.3ms	41.1ms
Relation.Binary.PropositionalEquality	92.7ms	88ms	76.9ms
Relation.Binary.PropositionalEquality.Core	21.8ms	18.3ms	18ms
Relation.Binary.Reflection	90.5ms	87.7ms	80.7ms
Relation.Binary.Vec.Pointwise	232.1ms	311.3ms	221ms
Relation.Nullary	10ms	12ms	9.7ms
Relation.Nullary.Decidable	54ms	49.3ms	49.8ms
Relation.Nullary.Negation	73.9ms	946.5ms	66.1ms
ShowNat	12.3ms	14.7ms	11.9ms
TrustMe	19.4ms	22.9ms	18.1ms
Vec	62ms	76.3ms	67ms

A.5. Runtime

Test case	Old	stage 1	stage 2	stage 3
CatchAllVarArity	1ms	1ms	1ms	2ms
Coind	2ms	2ms	2ms	1ms
CompareNat	3ms	1ms	1ms	1ms
CompileNumbers	2ms	1ms	1ms	1ms
CopatternStreamSized	2ms	2ms	2ms	2ms
IdentitySmashing	4ms	1ms	1ms	1ms
Issue1496	1ms	4ms	1ms	1ms
Issue561	2ms	1ms	1ms	1ms
ModuleReexport	1ms	1ms	3ms	1ms
PrimSeq	165ms	164ms	163ms	179ms
Sort	24ms	25ms	25ms	27ms
StaticPatternLambda	1ms	2ms	1ms	1ms
VecReverse	313ms	316ms	319ms	341ms
VecReverseIrr	38ms	37ms	26189ms	26174ms