Improving Agda's module system

Master's Thesis

Ivar Cornelis de Bruin

Improving Agda's module system

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ivar Cornelis de Bruin
born in Dordrecht, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Improving Agda's module system

Author:	Ivar Cornelis de Bruin
Student id:	4944135
Email:	ivardb@gmail.com

Abstract

Agda is a language used to write computer-verified proofs. It has a module system that provides namespacing, module parameters and module aliases. These parameters and aliases can be used to write shorter and cleaner proofs. However, the current implementation of the module system has several problems, such as an exponential desugaring of module aliases. This thesis shows how the module system can be changed to address these problems. We have found that we do not need any desugarings during type-checking, but can instead handle module parameters and aliases during signature lookup by making a small change to the scope-checker, completely eliminating any exponential growth problems and unnecessary complexity. This will allow users to make more effective use of the module system, simplifying their proofs. Furthermore, the improvements to the module system will allow future research to fix the problems with Agda's implementation of pretty-printing, records and open public statements.

Preface

This thesis has been written to fulfil the graduation requirements of the Master Computer Science at the Technical University in Delft. It has been written between November 2022 and June 2023.

During my master, I focused on programming languages and algorithms. At some point, I realised that I was starting to enjoy the programming language courses much more and I have always liked projects like this where you can try out different approaches and try to find their advantages and disadvantages. Combined with the fact that I had worked with both Jesper and Bohdan before, made this a perfect thesis to work on.

This report has been written in a way that should allow almost all computer scientists to follow what is happening and why, as special care is taken to introduce the reasons for each step. Sections 5.3 and 8.4 are the exception and they will be harder to follow for anyone not involved in Agda as they discuss topics that should be covered in the future without introducing all of the relevant background information separately. Readers that are not working with Agda, nor planning to, should probably skip these sections as they are difficult to follow and heavily focused on Agda's implementation.

I would like to thank my supervisors Jesper and Bohdan for their help during the thesis. Our weekly meetings and their feedback were extremely useful.

Ivar Cornelis de Bruin
Delft, the Netherlands
June 23, 2023

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background 3
- 2.1 Dependently-typed languages 3
- 2.2 Agda's module system 4
- 2.3 Type-checking Agda 5

3 The Problem 9
- 3.1 Identifying the problems 9
- 3.2 Addressing the problems 10

4 Simple Agda 11
- 4.1 Syntax 11
- 4.2 Scope-checking 12
- 4.3 Evaluation 12
- 4.4 Type-checking 12
- 4.5 Removed features 14

5 Iteratively improving the implementation of the module system 17
- 5.1 Problem 1 - Lack of structure 17
- 5.2 Problem 2 - Module alias performance 22
- 5.3 Problem 3 - Pretty-Printing 23

6 Generating Agda files 27
- 6.1 Why use a generator? 27
- 6.2 Limitations of randomly generated files 27
- 6.3 The Simple Agda Generator 28
- 6.4 Setting parameters to realistic values 29

7 Evaluation 33
- 7.1 Simple Agda results 33

7.2 Agda results 41

8 Discussion **47**
8.1 The Simple Agda experiments 47
8.2 Impact of the Agda experiments 48
8.3 Remaining limitations of the experiments 48
8.4 Advice for improving Agda's module system 49

9 Related work **53**

10 Conclusion **55**

Bibliography **57**

Acronyms **63**

A Full typing judgements for version 1 **65**

B Full typing judgements for version 2 **67**

C Paper for IFL **69**

List of Figures

4.1 WHNF evaluation of Simple Agda 12
4.2 getCommonTelescope definition 13
4.3 Term typing and elaboration judgements for Simple Agda 13
4.4 Declaration typing and elaboration judgements for Simple Agda 14
4.5 Telescope typing rules for Simple Agda 14

5.1 Signature lookup for version 1 19
5.2 Updated term typing judgements for version 1 20
5.3 Declaration typing judgements for version 1 21
5.4 `newDecls` definition for version 1 21
5.5 Signature lookup for version 2 22
5.6 Declaration typing judgements for version 2 23

6.1 The number of decls in files in both agda-stdlib and agda-categories 29
6.2 The number of imports in files in both agda-stdlib and agda-categories 30
6.3 The size of module parameters in files in both agda-stdlib and agda-categories . 31
6.4 The size of module alias arguments in files in both agda-stdlib and agda-categories 31

7.1 Experiment 1: Only declarations 34
7.2 Experiment 2: Impact of both size and number of module parameters 35
7.3 Experiment 3: Impact of type size 35
7.4 Experiment 4: Influence of the size of alias arguments 36
7.5 Experiment 5: Aliasing larger modules 36
7.6 Experiment 6: Nested aliases 37
7.7 Experiment 7: Impact of using aliased declarations 38
7.8 Multi-file experiment 1: Only declarations 39
7.9 Multi-file experiment 2: Import chains 39
7.12 Agda experiment 1: Only declarations 42
7.13 Agda experiment 2: Impact of module parameters 42
7.14 Agda experiment 4: Influence of the size of alias arguments 43
7.15 Agda experiment 5: Aliasing larger modules 43
7.16 Agda experiment 6: Nested aliases 44
7.17 Agda multi-file experiment 1: Only declarations 44

A.1 Signature lookup for version 1 65
A.2 WHNF evaluation of version 1 65
A.3 Telescope typing rules for version 1 65
A.4 Term typing judgements for version 1 66
A.5 Declaration typing judgements for version 1 66

A.6 `newDecls` definition for version 1 66

B.1 Signature lookup for version 2 67
B.2 WHNF evaluation of version 2 67
B.3 Telescope typing rules for version 2 68
B.4 Term typing judgements for version 2 68
B.5 Declaration typing judgements for version 2 68

List of Tables

7.10 Average size of the core files outputted during type-checking in the various experiments 40
7.11 Core file size of imported files in various multi-file experiments 40
7.18 File sizes for Agda experiments 45

Chapter 1

Introduction

Throughout programming history, programmers have struggled with bugs. This is especially a problem in security-related programs where bugs can have big consequences. One possible solution to this problem has been to create code that is proven to be correct. This has already seen some use in, for example, the creation of a C compiler [1].

To create such proofs, programmers have to work in proof-assistants [2]. Many of these proof-assistants are programming languages with strong type systems and additional properties such as guaranteed termination to allow for the creation of proofs. This thesis focuses on Agda [3], a proof assistant that uses a syntax similar to Haskell. This allows users to write programs in Agda, prove them correct, then transform them to Haskell code using a Haskell backend, thereby maintaining the guarantee that their code has the proven properties, while still using a programming language optimised for performance.

Agda is also used to prove various mathematical properties or even entire mathematical fields [4, 5]. To help programmers structure larger proofs, Agda provides a module system. Modules allow for easy namespacing and grouping of proofs. In addition, there are two interesting features that can be of great help when writing proofs: module parameters and module aliases.

Module parameters are in scope for all declarations in that module while module aliases allow programmers to create aliases that instantiate these parameters with specific arguments. This allows programmers to make proofs generic over some module parameters. Then, using a module alias, these proofs can be instantiated for a specific value as can be seen here, where `CategoryProofs` contains a variety of proofs generalised over some `Category c`:

```
postulate d e : Category
module CategoryProofs (c : Category) where
...
module DProofs = CategoryProofs d
module EProofs = CategoryProofs e
```

Structuring proofs this way greatly increases both the usability as well as the readability and ease of writing of the proofs. Unfortunately, the current version of Agda does not perform well when many aliases are used due to exponential growth problems. This occurs because Agda replaces the alias with specialised functions for each declaration being aliased, similar to what many compilers do when dealing with parametrised functions [6]. While this technique makes sense for a compiler, it is not necessarily a good idea for a proof-assistant, as we do not care for the speed of the type-checked code, but about the speed of the type-checker.

Furthermore, removing aliases makes the implementation of the type-checker more complex and bug-prone due to the increase in transformations needed and finally, any backends that are implemented for Agda will not have access to the aliases as the type-checker already removed them, limiting the potential compilations.

This thesis will analyse different approaches to Agda's module system to remove this performance bottleneck while preserving the module features during type-checking. Concretely, we make the following contributions:[*]

- We analyse the issues on the Agda GitHub issue tracker to identify the main problems with Agda's module system: The lack of structure, the performance problems with nested module aliases and a variety of issues related to pretty-printing (Chapter 3).
- We create a specification for a simplified version of Agda called Simple Agda which allows us to analyse Agda's module system in isolation of its other features (Chapter 4). This type-checker is implemented in Haskell and this original version will be referred to as version 0.
- We iteratively improve the type-checking process and implementation to address the previously mentioned problems by first adding a sense of structure by keeping modules and module aliases intact. Next, we address the performance problems by keeping aliases intact and finally we address some of the pretty-printing problems. These improvements will be referred to as versions 1 through 3 and all versions preserve the original semantics of the module system (Chapter 5).
- We create a generator for generating random Simple Agda programs and analyse two existing Agda libraries to find out what settings to use for the generator (Chapter 6).
- We analyse the different type-checkers in a variety of scenarios which shows that keeping aliases intact without expanding them has far superior performance, with barely any downsides. We also verify the accuracy of our experiments by executing them on Agda to make sure that our baseline version 0 matches with Agda's performance (Chapter 7).
- We analyse the results, the issues on the Agda GitHub repository and Agda's implementation to show that Agda should implement at least version 2 as soon as possible. We also use this information to outline what we believe to be the logical next steps in improving Agda (Chapter 8).

[*]All code and results can be found at `https://github.com/ivardb/AgdaModuleImprovement`

Chapter 2

Background

This chapter will provide the required background knowledge for this thesis. Section 2.2 will explain the most important features of Agda's module system while section 2.3 will explain how these features are type-checked. However, as Agda is dependently-typed, section 2.1 will first explain what dependent types are, as well as an explanation of weak head normal form evaluation which is needed to properly type-check dependently-typed programs. Readers already familiar with these concepts can skip this section.

2.1 Dependently-typed languages

Dependent types are types that depend on terms [7]. For example:

```
f : (b : Bool) -> (if b then Unit else Bool) -> (if b then Unit else Bool)
f = \b x . x
```

Here f is either the identity function for `Bool` or for `Unit`, depending on the value `b`. This dependency makes this a dependent type. To make this possible, parameters can have names in type signatures and these parameters are in scope for all later parameters. Such a list of named type parameters is called a telescope [8] and denoted with a Greek letter such as Δ.

Dependent typing also introduces some interesting changes to case-matching judgements such as if-statements. When the condition is a variable, we should introduce definitions for that variable to allow more programs to type-check, as types in the branches could depend on the variable in the condition. A typing judgement for such a case would look something like this:

$$\frac{\begin{array}{cc} x : \textbf{Bool} \in \Gamma & \Sigma;\Gamma \vdash A : \textbf{Type} \\ \Sigma;\Gamma \vdash v[x := \textbf{True}] : A[x := \textbf{True}] & \Sigma;\Gamma \vdash w[x := \textbf{False}] : A[x := \textbf{False}] \end{array}}{\Sigma;\Gamma \vdash \textbf{if } x \textbf{ then } v \textbf{ else } w \ : A}$$

2.1.1 Weak head normal form

Dependently-typed languages sometimes need to evaluate a term during type-checking.

```
x : if True then Bool else Unit
x = True
```

The above example should type-check. The type evaluates to `Bool` and `True` is a valid value of type `Bool`. To allow this, dependently-typed languages will evaluate types to weak head normal form (WHNF) [9]. To convert a term to WHNF, the outermost part of the term is evaluated until evaluation is no longer possible.

```
Bool -> if True then Bool else Unit
```

The above term is in WHNF as a function type cannot be evaluated further. The if expression is also not simplified, as knowing that we have a function type is enough to continue. Only once we need a more specific return type will we evaluate the if expression. Doing this evaluation lazily like this can lead to large performance increases as we do not perform unnecessary work.

To be able to reduce terms to WHNF we not only need to track the types of definitions, but also their actual definitions when type-checking.

```
b : Bool
b = True

f : if b then Bool else Unit
f = True
```

2.2 Agda's module system

This section will explain the module system of Agda in so far as it is relevant for this thesis. A complete overview of the module system can be found in chapter 4 of Norell's thesis [10].

Agda [10] is a dependently-typed language, mainly used as a proof assistant [2]. To accommodate this, the language is pure, total and terminating [11, 12]. These properties are needed in combination with the Curry-Howard correspondence to ensure that a proof that type-checks, is a valid mathematical proof [13].

To help structure proofs, Agda uses modules which can also be nested. We could for example have a module with a bunch of definitions and proofs related to data structures, which contains a module for Lists and one for Sets.

```
module DataStructures where
  module Lists where
    append : (A : Type) -> List A -> A -> List A
    contains : (A : Type) -> A -> List A -> Bool
  module Sets where
    contains : (A : Type) -> A -> Set A -> Bool
```

Many of these definitions will need to know the type of the elements in the data structure. Dependently-typed languages can simply take that type as a parameter to make the functions polymorphic. As this value is the same for each function, Agda allows it to be moved to the module level to create a module with data structures over elements of type `A`.

```
module DataStructures (A : Type) where
  module Lists where
    append : List A -> A -> List A
    contains : A -> List A -> Bool
  module Sets where
    contains : A -> Set A -> Bool
```

This feature is not limited to type polymorphism, but any parameter of any type can be moved to the module level so that all declarations can access it. As parameters are in scope for all further parameters, the following is also perfectly valid:

```
module M (A : Type) (x : A) where ...
```

When calling a function in a module with module parameters, we need to provide these parameters as if they were at the function level:

```
module Example where
  module M (A : Type) (x : A) where
```

```
    f : A
    f = x

    g : A
    g = f
  h : Bool
  h = M.f Bool True
```

In this example, `g` can call `f` without providing any arguments as `A` and `x` are in scope for both. As `h` is outside of `M`, it does need to provide some arguments to be able to call `f`.

If you often need to call functions inside of a module with the same parameters, you can make use of a module alias:

```
module DataStructuresBool = DataStructures Bool
```

This will create a module with all of the data structures for elements of type `Bool`. Module aliases can also introduce new parameters and do not have to provide an argument for all parameters of the original module:

```
module M' (b : Bool) = M (if b then Bool else Unit)
```

Both of these features can be incredibly useful to keep proofs readable and nicely structured. There is also a variety of purely scoping-related features in Agda such as a variety of import and open statements:

```
import M1 as M2
open import M1 hiding (_+_) renaming (N to N')
open M using (Z)
...
```

These features are all dealt with during scope-checking and as this thesis focuses on the type-checking part we will ignore these features as they can easily be added to any type-checker by simply changing the scope-checker that runs before it.

2.3 Type-checking Agda

This section will cover the type-checking process Agda uses for its modules. During type-checking, Agda will remove modules and instead output a completely flattened program consisting only of top-level definitions.

2.3.1 Scope-checking Agda

Agda starts by scope-checking a file. During this process, it verifies that all names that are referenced are declared beforehand. It then replaces all names with fully qualified names so that each name is unique and stores where it is defined.

It can then remove any visibility modifications as these are not necessary for type-checking. This means we end up with only two kinds of module declarations after scope-checking, module definitions and module aliases:

```
module M (A : Type) (x : A) where ...
module M1 (x : Bool) = M Bool x
```

These will be the constructs used in the remainder of this thesis.

2.3.2 Agda's core language

Once Agda finishes type-checking, it outputs a signature in a core language. This core language can be used to cache the result of type-checking in an interface file. This means that when the file is imported we do not have to type-check anything again. For this to be possible, the core signature has to contain type signatures for every declaration that is publicly accessible. So, if the type-checking implementation supports any form of inference, then all inferred information should be stored explicitly so that it does not have to be inferred again. Furthermore, the core language should be easy to load into the typing environment. This means that the typing environment and the core language are very closely related.

An Agda core interface has two essential parts. First, there is a list of section declarations. These are the module signatures.

```
section Example
section M (A : Type) (x : A)
```

These signatures are required when type-checking module aliases to see if the provided arguments are valid. Next, we get a list of definitions. Each definition is accompanied by a type signature that we know is correct. The core language does not support modules as these are flattened to the top level. How this flattening is performed is explained in section 2.3.3.

As an example: the Example module shown in section 2.2 will turn into the following core code:

```
section Example
section Example.M (A: Type)(x: A)

Example.M.f : (A : Type) -> A -> A
Example.M.f = \A x. x
Example.M.g : (A : Type) -> A -> A
Example.M.g = \A x. Example.M.f A x
Example.h : Bool
Example.h = Example.M.f Bool True
```

2.3.3 Type-checking and elaboration of module statements

The current implementation of Agda removes modules during type-checking. Part of this process is moving module parameters to the definition level. For example:

```
module M (X : Type) where
  M.id : X → X
  M.id x = x
```

will be transformed to:

```
M.id : (X : Type) → X → X
M.id X x = x
```

If there are multiple declarations in a module, this will create additional copies of the module parameter type. Because functions now have more arguments, each call will need to be modified during type-checking to introduce these new arguments.

```
module M (b : Bool) where
  f : Bool
  f = b
  module M2 (a : Unit) where
    g : Bool
    g = f
```

This is a perfectly valid program. Both `f` and `g` have `b` automatically in scope. However, when the module parameters are transferred to the definition level we end up with:

```
M.f = \b . b
M.g = \b a . M.f b
```

`M.f` now needs to be explicitly passed the `b` argument.

To type-check a module alias, new definitions are introduced. For each definition in the original, we need a definition in the new module that redirects to the old definition, passing the appropriate arguments.

```
module M (X : Type) where
  M.id : X → X
  M.id x = x
module MBool = M Bool
```

will transform to:

```
M.id : (X : Type) → X → X
M.id X x = x

MBool.id : Bool -> Bool
MBool.id = M.id Bool
```

This means that any arguments passed to an aliased module will be copied once for each declaration.

There are several problems with this way of type-checking and chapter 3 will explain what these are and how this thesis will address them.

Chapter 3

The Problem

This chapter will explain the various problems with the current Agda module system in section 3.1. Section 3.2 will then explain how this thesis will address these issues.

3.1 Identifying the problems

Agda's module system has several problems, many of which are reported as issues on GitHub. Before we go through the problems found in these issues there is a more general problem. The type-checker acts as a compiler by not only type-checking the code but immediately transforming it to a simplified language.

Agda allows for the implementation of custom backends which can be used to compile Agda to different languages. However, by then Agda will already have removed its modules and moved the module parameters to the declaration level. This means that the backend is not able to decide to do so for itself. If you want to make a backend for Agda that applies a transformation and then returns valid Agda code you will not be able to do so while preserving the original module structure.

Besides this conceptual problem, there are also many issues that have been reported on the issue tracker related to the module system. We will group these issues according to the type of problem they are about and see if they can be fixed through a redesign of the module system. Issues related to rewrite rules and other new Agda features will be ignored as the reason for these issues is not that the module system does not work properly but that whatever new feature is being implemented does not work well with modules and as such is out of scope for this thesis.

1. **Lack of proper structure during type-checking:** Agda has to make changes to the declarations to lift them to the top level and to move the module parameters to the declaration level. These changes can easily introduce bugs due to renaming to preserve unique names [14] or because it becomes harder to know if a parameter is part of the original declaration or if it came from a module [15–17]. While these bugs could all be easily resolved, some issues are still not fixed. One such issue is issue #6359 which is a problem that is caused by declarations being added to the global scope while they should only exist locally, thereby breaking the tracking of specific data resulting in buggy behaviour [18]. These issues show that the current approach of Agda is both complex and restrictive.

2. **Pretty-Printing problems:** Agda has a variety of problems with pretty-printing values from different modules. Either because it cannot remember where a definition originated [19, 20] or because it loses track of module parameters making infix operators especially confusing to read [21, 22].

3. **Performance problems:** While the previous problems were more convenience problems, Agda also has severe performance problems [23–25]. These problems are either caused by very large numbers of module parameters or more often by nested module aliases as this produces an exponential growth in the number of new declarations being created. This problem is so bad that the agda-categories library has a special wiki dedicated to avoiding Agda's performance problems [26].

4. **Open public statements:** While this thesis will mostly ignore the scoping part of Agda's module system there is one scoping feature we do want to discuss. Take the following example:

   ```
   module A where
     f = True
   module M (x : Bool) where
     open public A
     g = x
   y : Bool
   y = M.g True
   z : Bool
   z = M.f
   ```

   Here open public makes the contents of module `A` visible inside module `M`. However, the parameter `x` will not be added to the telescope of module `A`'s declarations. This is rather inconsistent behaviour that causes a lot of confusion [27, 28]. A more consistent behaviour would be to make open public desugar to using a module alias. This is not done yet because of the performance issues with module aliases [29].

5. **A variety of weird module features:** Open public statements are not the only module feature that could use a redesign. Anonymous modules [30] and named where blocks [31, 32] both have some weird behaviour as well. However, as none of these are affected by the handling of modules they will not be covered in this thesis.

6. **Records:** Records interact somewhat poorly with module parameters [33]. However, records also have a large variety of other problems [34, 35] as well as open proposals for changes [36, 37] and several calls for a proper overview of records to be created [35, 38]. This means that it would be better to redesign these separately after implementing the changes proposed in this thesis and as such records will not be covered by this thesis.

3.2 Addressing the problems

This thesis will answer the following question: "What changes need to be made to Agda's module system to increase the usefulness of its core features, improve its time complexity and make its core files more useful for potential compilers?" We also answer the following related questions:

- What are the individual effects of each of the changes made?
- Are there any disadvantages or side-effects that are introduced by these changes?
- What are the realistic use cases of Agda's module system and how do these cases affect performance?
- What are the follow-up steps to the proposed changes?

Chapter 4

Simple Agda

For most of this thesis, we will use a much simpler language than Agda. This chapter will describe the starting point of this language which will accurately represent Agda's module system. Later chapters will make modifications to the language to improve upon Agda's module system. This version is implemented as version 0[*]. The code is a modified version of pi-forall [39].

Section 4.1 will first cover the syntax of Simple Agda. Sections 4.2 to 4.4 will then cover the type-checking process for this language. Finally, section 4.5 will explain which features have been removed from Simple Agda and why this is acceptable.

4.1 Syntax

A, B, u, v	$::=$	x	*Variable*
	$\mid$	$\alpha.f$	*Qualified name*
	$\mid$	**Type**	*Universe*
	$\mid$	**Unit** $\mid$ **Bool**	*Built-in types*
	$\mid$	`1` $\mid$ **True** $\mid$ **False**	*Basic values*
	$\mid$	**if** u **then** v **else** w	*if expressions*
	$\mid$	$(x : A) \to B$	*Dependent function types*
	$\mid$	$\backslash x.u$	*Lambda*
	$\mid$	$u\ v$	*Application*
α, β	$::=$	ι	*No qualifier*
	$\mid$	$M.\alpha$	*Module qualifier*
Δ, Γ	$::=$	$\epsilon \mid (x : A)\Delta$	*Contexts*
Σ	$::=$	$\epsilon \mid decl, \Sigma$	*Declaration context*
decl	$::=$	$f : A$	*Type signature*
	$\mid$	$f = u$	*Definition*
	$\mid$	**module** $M\ \Delta$ **where** $decls$	*Module definition*
	$\mid$	**module** $M'\ \Delta = \alpha.M\ \bar{u}$	*Module alias*

Grammar 4.1: Simple Agda grammar

Simple Agda's grammar can be found in grammar 4.1. It is dependently typed with Pi-types, primitive Unit and Bool values as well as a dependent case expression to eliminate Bool values.

The core language, which type-checking outputs, can be found in grammar 4.2. It starts with a list of sections describing the modules and then contains a list of declarations. These declarations are either type signatures or definitions, as modules are removed.

[*] https://github.com/ivardb/AgdaModuleImprovement/tree/master/version0

$Core$	$::=$	$[section]\ [coreDecl]$	*Core*
$section$	$::=$	**section** $\alpha.M\ \Delta$	*Section with telescope*
$coreDecl$	$::=$	$\alpha.f : A$	*Type signature*
	$\mid$	$\alpha.f = u$	*Definition*

Grammar 4.2: Core language grammar

4.2 Scope-checking

Before a file is type-checked, we scope-check it. During scope-checking, we verify that all referenced names exist. We also fully qualify each name to simplify module operations during type-checking.

4.3 Evaluation

As Agda is dependently typed, we need to be able to evaluate terms to WHNF. The evaluation rules for this can be found in figure 4.1.

$$\frac{\alpha.f = u \in \Sigma}{\Sigma \vdash \alpha.f \xrightarrow{WHNF} u} \qquad \frac{}{\Sigma \vdash (\lambda x\ .\ u)\ v \xrightarrow{WHNF} u[x := v]} \qquad \frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash u\ v \xrightarrow{WHNF} u'\ v}$$

$$\frac{}{\Sigma \vdash \textbf{if True then } v \textbf{ else } w \xrightarrow{WHNF} v} \qquad \frac{}{\Sigma \vdash \textbf{if False then } v \textbf{ else } w \xrightarrow{WHNF} w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash \textbf{if } u \textbf{ then } v \textbf{ else } w \xrightarrow{WHNF} \textbf{if } u' \textbf{ then } v \textbf{ else } w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} v \qquad \Sigma \vdash v \xrightarrow{WHNF} w}{\Sigma \vdash u \xrightarrow{WHNF} w} \qquad \frac{}{\Sigma \vdash u \xrightarrow{WHNF} u}$$

Figure 4.1: WHNF evaluation of Simple Agda

4.4 Type-checking

The typing judgements for Simple Agda are split into term level judgements and declaration level judgements. Both are of the form: $\Sigma; \Gamma \vdash u : A \rightsquigarrow u'$. This states that given a signature environment Σ and a variable context Γ, the term u has the type A and type-checking it results in u', where u' is a modified version of u.

Term typing and elaboration judgements The judgements can be found in figure 4.3. These judgements make no distinction between type-inference and type-checking although this could be implemented [40].

The reason that term typing outputs modified terms is that we sometimes need to insert additional terms when type-checking a qualified name. Take our earlier example:

```
module M (b : Bool) where
  f : Bool
  f = b
  module M2 (a : Unit) where
```

```
    g : Bool
    g = f
```

This is a perfectly valid program. Both `f` and `g` have `b` automatically in scope. However, when the module parameters are transferred to the definition level we end up with:

```
M.f = \b . b
M.g = \b a . M.f
```

This will not type-check. The shared module parameter `b` needs to be threaded through the function call. These variables can be found using the `getCommonTelescope` method, defined in figure 4.2, which will find the shared module telescope with a qualified name. The found telescope can then be turned into a set of arguments by the `asVars` method which will convert a telescope into terms by taking the names in the telescope. Adding these arguments to the function results in $M.f\ b$, which is correct.

$$
\begin{aligned}
&\texttt{getCommonTelescope}(M(\Delta)\Gamma, M.\alpha) = \Delta \ \texttt{++}\ \texttt{getCommonTelescope}(\Gamma', \alpha)\\
&\texttt{getCommonTelescope}(N(\Delta)\Gamma, M.\alpha) = \epsilon\\
&\texttt{getCommonTelescope}(\epsilon, M.\alpha) = \epsilon\\
&\texttt{getCommonTelescope}(\Gamma, \iota) = \epsilon
\end{aligned}
$$

Figure 4.2: getCommonTelescope definition

$$\frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x : A \rightsquigarrow x} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{Type} : \textbf{Type} \rightsquigarrow \textbf{Type}} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{Bool} : \textbf{Type} \rightsquigarrow \textbf{Bool}}$$

$$\frac{}{\Sigma; \Gamma \vdash \textbf{Unit} : \textbf{Type} \rightsquigarrow \textbf{Unit}} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{True} : \textbf{Bool} \rightsquigarrow \textbf{True}}$$

$$\frac{}{\Sigma; \Gamma \vdash \textbf{False} : \textbf{Bool} \rightsquigarrow \textbf{False}} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{1} : \textbf{Unit} \rightsquigarrow \texttt{1}}$$

$$\frac{\Sigma; \Gamma \vdash u : \textbf{Bool} \rightsquigarrow u' \qquad \Sigma; \Gamma \vdash v : A \rightsquigarrow v' \qquad \Sigma; \Gamma \vdash w : A \rightsquigarrow w'}{\Sigma; \Gamma \vdash \textbf{if}\ u\ \textbf{then}\ v\ \textbf{else}\ w\ : A \rightsquigarrow \textbf{if}\ u'\ \textbf{then}\ v'\ \textbf{else}\ w'}$$

$$\frac{\Sigma; \Gamma \vdash A : \textbf{Type} \rightsquigarrow A' \qquad \Sigma; \Gamma, x : A' \vdash B : \textbf{Type} \rightsquigarrow B'}{\Sigma; \Gamma \vdash (x : A) \to B : \textbf{Type} \rightsquigarrow (x : A') \to B'} \qquad \frac{\Sigma; \Gamma, x : A \vdash u : B \rightsquigarrow u'}{\Sigma; \Gamma \vdash \lambda x\ .\ u : A \to B \rightsquigarrow \lambda x\ .\ u'}$$

$$\frac{\Sigma; \Gamma \vdash u : A \rightsquigarrow u' \qquad \Sigma; \Gamma \vdash A \xrightarrow{WHNF} (b : B) \to C \qquad \Sigma; \Gamma \vdash v : B \rightsquigarrow v'}{\Sigma; \Gamma \vdash u\ v : C[b := v'] \rightsquigarrow u'\ v'}$$

$$\frac{\texttt{getCommonTelescope}(\Gamma, \alpha) = \Delta \qquad \alpha.f : \Delta \to A \in \Sigma}{\Sigma; \Gamma \vdash \alpha.f : A \rightsquigarrow \alpha.f\ \texttt{asVars}(\Delta)}$$

Figure 4.3: Term typing and elaboration judgements for Simple Agda

Declaration typing and elaboration judgements The judgements for declarations are found in figure 4.4. While the language uses separate type signatures and definitions, we will treat them as one unit in the judgements.

When type-checking modules, we need to move the module parameters to the declaration level and lift the declarations inside to the top level. The first rule shows how this works for definitions. Once the term itself has been type-checked, we can insert the entire variable context around both the type and the definition to lift it to the top level. This means that the signature will only contain top level declarations. This is why we need to remove some of the parameters of a definition when type-checking a qualified term.

The second rule shows the typing of a module. Here we simply add the module parameters to the context together with the module name, before continuing with typing the declarations. The module name is needed for our `getCommonTelescope` function to identify which parameters to remove from a lifted declaration.

The third rule shows the typing of a module alias. For this, we create new declarations for each of the original declarations. These declarations call the original with the arguments provided to the alias. Note that just like with normal qualified names, we also insert the parameters acquired from `getCommonTelescope` to the function call and just like with normal modules, we put the module parameters around the new definitions.

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \rightsquigarrow A' \qquad \Sigma;\Gamma \vdash u : A' \rightsquigarrow u'}{\Sigma;\Gamma \vdash \alpha.f : A = u \rightsquigarrow [\alpha.f : \Gamma \to A' = \lambda\Gamma.\, u']}$$

$$\frac{\Sigma;\Gamma \vdash \Delta \rightsquigarrow \Delta' \qquad \Sigma;\Gamma M(\Delta') \vdash \mathit{decls} \rightsquigarrow \mathit{decls}', M(\Gamma\Delta')}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls} \rightsquigarrow \mathit{decls}'}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash \Delta \rightsquigarrow \Delta' \qquad \texttt{getCommonTelescope}(\Gamma, M') = \Gamma' \qquad M'(\Gamma'\Theta) \in \Sigma \\ \Sigma;\Gamma M(\Delta') \vdash \bar{u} : \Theta \rightsquigarrow \bar{u}' \qquad \text{for each } M'.f_i : \Gamma'\Theta \to A \in \Sigma \\ \text{let } \delta_i = M.f_i : \Gamma\Delta' \to A[\Theta := \bar{u}'] = M'.f_i\ \texttt{toVars}(\Gamma')\ \bar{u}'\end{array}}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta = M'\ \bar{u} \rightsquigarrow \bar{\delta}, M'(\Gamma\Delta)} \qquad \frac{}{\Sigma;\Gamma \vdash [] \rightsquigarrow []}$$

$$\frac{\Sigma;\Gamma \vdash \mathit{decl} \rightsquigarrow \Sigma' \qquad \Sigma\Sigma';\Gamma \vdash \mathit{decls} \rightsquigarrow \Sigma''}{\Sigma;\Gamma \vdash \mathit{decl} :: \mathit{decls} \rightsquigarrow \Sigma'\Sigma''}$$

Figure 4.4: Declaration typing and elaboration judgements for Simple Agda

$$\frac{}{\Sigma;\Gamma \vdash \epsilon \rightsquigarrow \epsilon} \qquad \frac{\Sigma;\Gamma \vdash A : \textbf{Type} \rightsquigarrow A' \qquad \Sigma;\Gamma, x : A \vdash \Delta \rightsquigarrow \Delta'}{\Sigma;\Gamma \vdash (x : A), \Delta \rightsquigarrow (x : A', \Delta')}$$

Figure 4.5: Telescope typing rules for Simple Agda

4.5 Removed features

Simple Agda is missing quite a few features compared to Agda, which include:

- Inductive data types
- Pattern matching
- Mutual definitions
- Universe levels
- Meta variables
- Record types

For each of these, we will explain why it is acceptable to remove them for Simple Agda.

Inductive data types can be seen as simply another sort of declaration when it comes to how it interacts with modules. Adding them to the language would make the type-checker much more complicated but would not change much about the module system. Not including it does make our experiments in chapter 7 a bit less accurate as we can only create complicated terms from if expressions and lambdas while in reality, the most complex terms will be proofs and data types.

Pattern matching and mutual definitions mostly influence the totality and terminality checker which are completely separate from the module system. In a similar vein, metavariables and universe levels only interact with the term level typing. All of these features can thus be left out.

Record types do interact quite a bit with modules. However, the interaction between record types and modules would probably make an interesting thesis subject on its own, so it's considered out of scope for this thesis.

Chapter 5

Iteratively improving the implementation of the module system

This chapter will propose iterative improvements to the module system to fix its problems. Section 5.1 will address the lack of structure in the type-checker by keeping modules and module parameters intact. Section 5.2 will address the performance problems by keeping module aliases intact as well. These two improvements will be evaluated on their performance.

The final problem that this chapter will address is pretty-printing in section 5.3. Not all of Agda's pretty-printing problems can be solved by changing the module system nor can they be modelled perfectly in Simple Agda. This means that this section will split the problem into two. The first set of problems relates to mix-fix notation inside parametrised modules and can be fixed with a simple change to the module system. This change will also be programmed into Simple Agda as version 3. However, not all of its effects can be explored as Simple Agda does not actually implement mix-fix notation.

The second set of problems with pretty-printing does not have an obvious solution. There are multiple ways of solving them, either through different evaluation strategies, changes in the module system or improvements to Agda's display form system. Section 5.3 will explain these various approaches so that future work can compare them in more detail.

5.1 Problem 1 - Lack of structure

Chapter 3 described a number of issues that occurred due to the lack of structure in Agda's type-checking and the moving of module parameters to the declaration level. This lack of structure is completely unnecessary as we are already tracking the sections in the core language, which are simply the module headers detached from their statements. We can combine them and allow for nested modules which would allow us to maintain the structure with few changes to the output, but with much more information.

This section will explain what needs to change to accomplish this and why. We will also go over the complexity of implementing this version and see how it compares to version 0.

5.1.1 Changes made

The main change in this version is the introduction of modules in the core language. This change means that sections are no longer necessary, as those were simply the module headers detached from their definitions and we can now simply get these headers from the modules themselves. The updated grammar for the core language can be found in grammar 5.1.
Now that we keep track of modules, our typing environment Σ also becomes more structured. A module qualifier now becomes a path used to find the declaration in the signature.

$$
\begin{array}{lcll}
\mathit{Core} & ::= & \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{coreDecls} & \textit{Core} \\
\mathit{coreDecl} & ::= & \alpha.f : A & \textit{Type signature} \\
 & \mid & \alpha.f = u & \textit{Definition} \\
 & \mid & \textbf{module}\ \alpha.M\ \Delta\ \textbf{where}\ \mathit{coreDecls} & \textit{Module}
\end{array}
$$

Grammar 5.1: Core language grammar for version 1

Each module has its own signature that can contain other modules with their own signatures, creating a tree. This structure also allows for easier module parameter handling and easier creation of new modules when dealing with module aliases as we can easily find all declarations belonging to a specific module. Before we can define a notion of module lookup in such a structure we need to look at how we use qualified names in version 1.

Qualified names In version 0 we had to insert module parameters as arguments to any function call to ensure they would still type-check after being lifted to the top level. If module parameters are kept intact we no longer need to do this to ensure everything type-checks. By keeping track of the module parameters for each module, we could have the typing environment take care of these module parameters. This will however lead to problems when evaluating terms. Take the following code:

```
module M (b : Bool) where
  f : Bool
  f = b

  g : Bool
  g = M.f

module N = M False
main : (if N.g then Unit else Bool)
main = N.g
```

Here `g = M.f` is a perfectly valid definition for type-checking as the `b` parameter is in scope for both `g` and `f`. We have already qualified the name `f` so that when we evaluate `N.g` in the if statement, we will still reference the same `f`. However, this is not yet enough as evaluating the type of `main` in this example will lead to:

```
module M (b : Bool) where
  f : Bool
  f = b

  g : Bool
  g = M.f

module N = M False
main : (if M.f then Unit else Bool)
main = N.g
```

This does not type-check since `M.f` outside of module `M` is a function `Bool -> Bool`. To address this issue we will have the scope checker insert names together with the module parameters like so:

```
module M (b : Bool) where
  f : Bool
  f = b
  g : Bool
  g = (M b).f

module N = M False
main : (if N.g then Unit else Bool)
main = N.g
```

We make explicit which parameters are used for each module. If we now evaluate the call to `N.g` we get:

```
module M (b : Bool) where
  f : Bool
  f = b

  g : Bool
  g = (M b).f

module N = M False
main : (if (M True).f then Unit else Bool)
main = N.g
```

This program still type-checks.

This means that terms in our core language now use term-qualified names as can be seen in grammar 5.2 for function calls and module aliases. This does not require changes to the surface language of Simple Agda as we only insert these during scope-checking.

$$\begin{array}{rcll} \alpha, \beta & ::= & \iota & \textit{No qualifier} \\ & | & M.\alpha & \textit{Module qualifier} \\ & | & (M\ \bar{u}).\alpha & \textit{Term qualifier} \end{array}$$

Grammar 5.2: Term-qualified names

Signature lookup Now that we have updated our typing environment and redesigned our fully qualified names we can define signature lookup. We will use the syntax $\Sigma!\alpha \rightsquigarrow (\Delta, \Sigma')$. As users are not required (nor even able) to pass module parameters in the qualified name we also return a telescope Δ which contains any remaining module parameters. The split operator will split the telescope into the parameters for which arguments were supplied and those that have not been applied yet. The definition of signature lookup can be found in figure 5.1.

$$\frac{\textbf{module}\ M\ \Delta = \Sigma' \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{u}) \qquad \Sigma'[\Delta_1 := \bar{u}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma!(M\bar{u}).\alpha \rightsquigarrow (\Delta_2\Delta', \Sigma'')}$$

$$\frac{}{\Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure 5.1: Signature lookup for version 1

Updated term typing Another advantage of having the scope-checker take care of the module parameter insertion is that the term type-checker no longer needs to make any changes and we can thus remove its output. Note that this is only true for Simply Agda and not Agda as Agda still requires changes for inference and other removed features.

Figure 5.2 shows how we previously type-checked function calls and if-expressions as well as their new versions. All other typing judgements are similarly simplified to the if-expressions but are left out for brevity. A complete overview of the updated judgements can be found in figure A.4 in appendix A. For the function calls we can see that we find the correct signature, look up the type in that signature and then add any required module parameters to it.

Old judgements

$$\frac{\Sigma;\Gamma \vdash u : \mathbf{Bool} \rightsquigarrow u' \qquad \Sigma;\Gamma \vdash v : A \rightsquigarrow v' \qquad \Sigma;\Gamma \vdash w : A \rightsquigarrow w'}{\Sigma;\Gamma \vdash \mathbf{if}\ u\ \mathbf{then}\ v\ \mathbf{else}\ w\ : A \rightsquigarrow \mathbf{if}\ u'\ \mathbf{then}\ v'\ \mathbf{else}\ w'}$$

$$\frac{\texttt{getCommonTelescope}(\Gamma, \alpha) = \Delta \qquad \alpha.f : \Delta \to A \in \Sigma}{\Sigma;\Gamma \vdash \alpha.f : A \rightsquigarrow \alpha.f\ \texttt{asVars}(\Delta)}$$

Updated judgements

$$\frac{\Sigma;\Gamma \vdash u : \mathbf{Bool} \qquad \Sigma;\Gamma \vdash v : A \qquad \Sigma;\Gamma \vdash w : A}{\Sigma;\Gamma \vdash \mathbf{if}\ u\ \mathbf{then}\ v\ \mathbf{else}\ w\ : A} \qquad \frac{\Sigma!\alpha \rightsquigarrow (\Delta, \Sigma') \qquad f : A \in \Sigma'}{\Sigma;\Gamma \vdash \alpha.f : \Delta \to A}$$

Figure 5.2: Updated term typing judgements for version 1

Declaration typing judgements The typing judgements for declarations can be found in figure 5.3. The `extend` function places the declaration in the appropriate signature in the environment.

Type-checking definitions and modules is very simple in this version. Module aliases are a bit more complicated as we need to create a new module for the alias. The `newDecls` function will recursively generate a new module that calls the declarations from the aliased modules with the appropriate arguments and any module parameters can be passed along using term-qualified names.

```
module M (b : Bool) where
  module M2 (b2 : Bool) where
    and : Bool
    and = if b then b2 else False
module MTrue = M True
```

which becomes:

```
module M (b : Bool) where
  module M1.M2 (b2 : Bool) where
    M1.M2.and : Bool
    M1.M2.and = if b then b2 else False
module MTrue where
  module MTrue.M2 (b2 : Bool) where
    MTrue.M2.and : Bool
    MTrue.M2.and = M.(M2 b2).and True
```

the definition for `newDecls` can be found in figure 5.4.

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma \vdash u : A}{\Sigma;\Gamma \vdash \alpha.f : A = u \rightsquigarrow \alpha.f : A = u}$$

$$\frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma;\Gamma M(\Delta) \vdash \mathit{decls} \rightsquigarrow \mathit{decls}'}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls} \rightsquigarrow \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls}'}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash \Delta \qquad \Sigma!\alpha \rightsquigarrow (\Theta, \Sigma') \qquad \textbf{module}\ M'\ \Theta'\ \textbf{where}\ \mathit{decls} \in \Sigma' \\ \Sigma;\Gamma M(\Delta) \vdash \bar{u} : \Theta\Theta' \qquad \text{let}\ \mathit{decls}' = \texttt{newDecls}(M, \alpha.M', \bar{u}, \mathit{decls}[\Theta\Theta' := \bar{u}])\end{array}}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta = \alpha.M'\ \bar{u} \rightsquigarrow \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls}'}$$

$$\frac{}{\Sigma;\Gamma \vdash [] \rightsquigarrow \epsilon} \qquad \frac{\begin{array}{c}\Sigma;\Gamma \vdash \mathit{decl} \rightsquigarrow \mathit{decl}' \qquad \text{let}\ \Sigma' = \texttt{extend}(\Sigma, \mathit{decl}') \\ \Sigma';\Gamma \vdash \mathit{decls} \rightsquigarrow \Sigma''\end{array}}{\Sigma;\Gamma \vdash \mathit{decl} :: \mathit{decls} \rightsquigarrow \Sigma''}$$

Figure 5.3: Declaration typing judgements for version 1

$$\begin{aligned}
&\texttt{newDecls}(M, M', \bar{u}, D) = \texttt{newDecl}(M, M', \bar{u}, d) \text{ for each } d \in D \\
&\texttt{newDecl}(M, M', \bar{u}, M'.f : A = u) = M.f = M'.f\ \bar{u} \\
&\texttt{newDecl}(M, M', \bar{u}, \textbf{module}\ M'.N\ \Delta\ \textbf{where}\ \mathit{decls}) = \\
&\qquad\qquad \textbf{module}\ M.N\ \Delta'\ \textbf{where}\ \texttt{newDecls}(M.N, M'.(N\ \Delta), \bar{u}, \mathit{decls})
\end{aligned}$$

Figure 5.4: `newDecls` definition for version 1

5.1.2 Implementation analysis

Implementing a structured module system is much easier than an unstructured module system. Having to figure out which parameters go where in version 0 is not easy and is extremely bug-prone. In this version, we only need to insert module parameters into declarations when adding qualified names. This is really easy as the scope-checker already knows which qualifiers to insert and we now simply have to also add the parameters when adding the qualifier. This does not lead to additional complexity compared to only inserting the qualifier.

Dealing with the module aliases does become more complex however as the generated declarations all have different types depending on how far nested they were in the original module. In version 0 this was simple, as all declarations being aliased were already lifted to the top level. In version 1 they all exist at different levels, requiring you to track all this extra data to generate the appropriate new declarations.

Generating these modules becomes even more complicated with partial aliases. These are not represented in the typing judgements, but in an actual implementation, they can present a challenge as the following code is perfectly acceptable:

```
module M (A : Type) (x : A) where
...
module MBool = M Bool
```

In Agda, the module parameters are moved to the declaration level and a partial application of a module is the same as partly applying a function, which is perfectly fine. However, if we change aliases to modules, this is no longer possible. This means we need to insert the remaining parameters:

```
module M (A : Type) (x : A) where
```

```
...
module MBool (x : Bool) = M Bool x
```

note that the added parameter `x` now has type `Bool` instead of type `A`. When applying the arguments to the aliased module, we need to perform the created substitutions everywhere in the new module, including in the inserted parameters.

This problem with partial aliases is only present if aliases go to modules. If we transform them into declarations or keep aliases intact, it will automatically work, as the alias arguments become function arguments and functions can be partially applied.

5.2 Problem 2 - Module alias performance

The biggest problem with Agda's module system is the performance problems caused by the expansion of module aliases. As we now have a structured core language, it is relatively easy to also preserve aliases, thereby making the source and core languages of Simple Agda equal except for the term-qualified names introduced by the scope-checker.

We will again cover both the required changes and the ease of implementation.

5.2.1 Changes made

If we no longer expand module aliases then we do not need to make any changes to Simple Agda while type-checking as the module system is kept intact during type-checking. Instead, we now need to deal with aliases in the signature lookup.

The updated signature lookup can be found in figure 5.5. Dealing with module aliases is very similar to the changes made in version 1 to support term-qualified names. In addition, we need to keep track of both the current signature and the root signature as aliases are relative to the root, not to the current signature.

When we encounter a module alias we have to look up the module it is aliasing. We then have to substitute the provided arguments in the found signature and then we can continue our lookup in this signature. These substitutions can of course be applied lazily in an actual implementation as we only need a single declaration from the final signature.

$$\frac{\textbf{module}\ M\ \Delta = \Sigma' \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma^R \vdash \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta', \Sigma'')}$$

$$\frac{\begin{array}{c}\textbf{module}\ M\ \Delta = M'\ \bar{u} \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma^R \vdash \Sigma^R!M' \rightsquigarrow (\Delta', \Sigma') \\ \Delta_1', \Delta_2' = \texttt{split}(\Delta', \bar{u}) \qquad \Sigma^R \vdash \Sigma'[\Delta_1' := \bar{u}[\Delta_1 := \bar{a}]]!\alpha \rightsquigarrow (\Delta'', \Sigma'')\end{array}}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta_2'\Delta'', \Sigma'')}$$

$$\frac{}{\Sigma^R \vdash \Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure 5.5: Signature lookup for version 2

Our typing judgements will remain mostly intact. The typing judgement for module aliases changes of course and none of our typing judgements need an output as we no longer change any terms. These new judgements can be found in figure 5.6. The judgement for aliases is very similar to that of normal modules. We simply have to do an additional lookup and substitution to find the module we are aliasing.

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma \vdash u : A}{\Sigma;\Gamma \vdash \alpha.f : A = u} \qquad \frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma;\Gamma M(\Delta) \vdash decls}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta\ \textbf{where}\ decls}$$

$$\frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma \vdash \Sigma!\alpha \rightsquigarrow (\Theta, \Sigma') \qquad \textbf{module}\ M'\ \Theta' \in \Sigma' \qquad \Sigma;\Gamma M(\Delta) \vdash \bar{u} : \Theta\Theta'}{\Sigma;\Gamma \vdash \textbf{module}\ M\ \Delta = \alpha.M'\ \bar{u}}$$

$$\frac{}{\Sigma;\Gamma \vdash []} \qquad \frac{\Sigma;\Gamma \vdash decl \qquad \Sigma;\Gamma \vdash decls}{\Sigma;\Gamma \vdash decl :: decls}$$

Figure 5.6: Declaration typing judgements for version 2

5.2.2 Implementation analysis

Implementing this version is much easier than the implementation of the other versions. Version 1 already simplified the term-typing and the handling of module parameters but the typing of module aliases was rather complicated. Version 2 completely eliminates this complexity as the declaration typing is now also trivial, while the signature lookup did not get much more complicated as aliases require roughly the same handling as term-qualified names which were already supported.

Overall, this version eliminates a lot of complexity, it eliminates the exponential declaration generation and the type-checker no longer hides features from the backend allowing it to decide for itself how it wants to handle them.

5.3 Problem 3 - Pretty-Printing

While this thesis mostly focuses on the performance problems with Agda's module system, the system also has a variety of pretty-printing problems that we would like to go over. These problems can be divided into two types. The first type of problem is related to infix operators in parametrised modules which will be discussed in section 5.3.1. These are problems that can largely be fixed by a few modifications to our earlier versions. Section 5.3.2 will then discuss the problems Agda has with deciding which qualified name to use when pretty-printing.

5.3.1 Infix operators and module parameters

Agda has great support for infix and mix-fix operators. These can be created by using _ in names such as `if_then_else_` which says that an if expression takes three terms in the specified places. This works great until module parameters get involved. Take the following module which defines String addition with a separator passed as module argument.

```
module M (sep : String) where
  _<+>_ : String -> String -> String
  s <+> s2 = s + sep + s2
x = M.<+> ";" "ab" "de"
```

The definition of the operator inside module `m` can nicely use the infix notation. Outside of the module, this is not possible as we need to pass the module parameter as well. This could be fixed by creating a module alias that supplies an argument to the module but that does not work for automatically generated types when Agda is used interactively. Furthermore, it can lead to pretty-printing problems when we use partly applied operators. `M.<+> ";" "ab"` is a function that adds `"ab;"` in front of its argument. However, this will get pretty-printed as `";" M.<+> "ab"` which is not a function at all and equal to `";ab"` [21]. While this pretty-printing bug might be fixed by keeping the module parameters intact, as it should allow the pretty-printer to detect that it cannot use infix notation here, it would still be nice to have a good way to pretty-print this.

In the previous chapters, we have already introduced the syntax of `(M Bool).f` for use by the scope-checker. Enabling this syntax for use by the programmer does not require any further changes to the type-checker and will prevent this problem for at the very least infix operators where we can now type `"ab" (M ";").<+> "de"`. For mixfix operators, users can qualify the first term, for example: `(M Bool).if b then False else True`.

In the code repository, a version 3 has been defined[*] that enables this syntax in the source language to show that it does not require any further changes from version 2. This version will not be used in experiments as this change is not about performance and does not use the same source language as the other versions.

5.3.2 Choosing the right qualified names

In this section, we would like to mention a variety of problems encountered with pretty-printing qualified names. There is generally not an obvious solution to these problems. This section will propose a variety of possible solutions for further analysis in future work.

5.3.2.1 Pretty-printing generated code

Agda has interactive features that require it to generate code. Agda can split a variable in a pattern into further cases by creating a case for each possible constructor or it can interactively generate the type of a hole in the code. During both of these operations, it needs to figure out how to qualify displayed names. At the moment this is done through an inverse scope lookup and display forms. This system has a variety of bugs related to imports [41] and open statements [22] but there is a good chance that these are simply bugs or could be fixed easily once a structured module signature is used.

The system also has flaws that cannot easily be fixed. Take the following example taken from issue 1643 [42].

```
module M where
  postulate A : Set

module N = M

open M

postulate a : A

x : A
x = {!a!}
```

Here we have three `A`'s in scope. `M.A`, `N.A` and `A` which comes from `M.A` through the open statement. When Agda is asked to display the type of the hole it gives `N.A` instead of `A`. It does this because it has a renaming rule for `M.A` -> `N.A` introduced by the alias and `A` has already been disambiguated to `M.A`. If we remove the renaming rule, then the example would not work correctly in the case where we `open N` as it would display `M.A` but no longer rename it to `N.A`. This example shows that the current system needs a redesign once the module system has been updated as keeping more information in the signature would allow for a more powerful system to be built.

[*] https://github.com/ivardb/AgdaModuleImprovement/tree/master/version3

5.3.2.2 WHNF evaluation

There are also problems when evaluating existing terms. Take the following code where we want to reduce `main` to normal form.

```
module M (b : Bool) where
  postulate f : Bool

  g : Bool
  g = f
module N = M false
main = N.g
```

Reducing `main` to normal form will result in `M.f false`. While correct, a perhaps nicer form would be `N.f`. This could be achieved through a modification of the type-checker to modify the names in a term when looking up a definition from an alias. This could be implemented as a lazy substitution similar to that performed for the arguments of the alias. If we use the new term-qualified syntax we would perform the lazy substitution from `(M b).f` -> `(M false).f` this could be changed to instead do `(M b).f` -> `N.f`.

Mini-modules [43] instead proposes to use relative names for qualified terms. By manipulating the qualifiers of terms we could both improve the normal forms as well as avoid many substitutions of module arguments.

Another approach would be to keep track of the original code similar to how we track where a term was located in the original file through source position annotations. Similar annotations could perhaps be used for function calls to track where a reduced term originated. This information could then be used to clarify how a term came to be as it could be nicer to have the reduced name, to better see why something goes wrong. Take the following example:

```
module M where
  g = True
  f = g
module N where
  open public M

module O = N
module P = O
x : if P.f then Set else Bool
x = True
```

This program will not type-check as `P.f` reduces to `True` instead of `False`. Here it is however not very useful to tell the user that this happens because `P.g` equals True. We would then still have to find out where `P.g` is defined and have to go through multiple steps of aliases and open public statements. In this scenario, it would be nice if we could request some form of stack trace where the type-checker is able to tell us that the following reduction occurred: `P.f` -> `O.f` -> `N.f` -> `M.f`. Such a history would also work in our first example to let us know that `(M false).f` came from `N.g` or could even be used to realise that `(M false).f = N.f` as aliases are kept intact and this information is thus available.

The comparison between these different approaches is non-trivial and should be made in more detail in future work as whatever option is chosen would have a significant impact on how users interact with Agda and can thus not be changed easily to another option afterwards.

Chapter 6

Generating Agda files

One of the biggest issues with the current implementation of Agda is its performance. This means that we need to evaluate our improved versions on performance as well to make sure that they are actually an improvement. These experiments need to analyse the impact of a variety of module features such as the size of aliases, the number of aliases, the number of module parameters and others to make sure that we do not create any performance regressions.

This chapter will describe how we create the files used for this performance comparison and why we need a generator to do so.

6.1 Why use a generator?

Using randomly generated tests is common in Haskell as a form of property-based testing [44]. QuickCheck [45] is a common testing framework that does exactly this. For normal testing, random tests are useful because they are often able to quickly find edge cases that you would not find with manual testing. While this was beneficial for finding bugs during development, it is not the main reason for using a generator for this thesis.

Instead, the reason we rely on random tests is that they allow us to isolate a specific property, such as the number of declarations in a module being aliased. If we tried to do this using existing files this would be near impossible as it is very difficult to find files that have a similar structure except for a single difference such as the size of the module being aliased. We could of course manually create this dataset, but that would take a lot of work and would not allow for the easy creation of new experiments or larger data sets for existing experiments.

To use a generator for this, we need a generator that generates programs of roughly equal difficulty. This is only really possible once programs are large enough that all of the random choices converge to their average. This is the reason that all of our generated files will have at least 50 declarations even though many real files are smaller than this. Furthermore, if we generate enough files for each setting we can average them to get a stable average time for a specific setting. We can then alter one of the settings to analyse its impact.

6.2 Limitations of randomly generated files

One of the biggest limitations of generating random code is ensuring that the result still type-checks. If you generate functions of different types then these become difficult to use as you will only be able to use them in places that need something of that type.

With dependent types, we also need to ensure that all types can be reduced to the intended type. If we generate the following type: `if p1 then Bool else Bool` we need to be certain that p1 will be known when evaluating this type as otherwise the type-checker might not realise that this is equal to `Bool`. This severely limits the possibilities for types.

The second big limitation of random generators is that the generated programs are nonsense. While it is easy to generate an expression of a certain size, this expression will not make sense. No real code will nest 10 if expressions inside a lambda that is immediately called with an argument consisting of a further 5 if expressions. This means that generated code files can never be compared to real code files.

While these are two big disadvantages, generated files still allow us to isolate specific properties to see how they affect performance which is exactly what we need.

6.3 The Simple Agda Generator

The generator developed for this thesis generates declarations of type `Bool` as well as module parameters of type `Bool`. This allows us to easily use other declarations as everything is either of type `Bool` or can become of type `Bool` by supplying a number of boolean arguments. We still cannot use declarations inside types as they could depend on unknown module arguments.

To generate files with the generator, two pieces of configuration are needed. The first is a structure which tells the generator the module hierarchy, how many declarations to generate as well as which modules to alias where. The declarations produced by the aliases will automatically be in scope for later declarations just like normal declarations.

The second set of configurations is a series of settings that tell the generator how large the generated values should be. The following settings are supported:

- Term size: How large should declarations be? This indirectly affects how many function calls can be produced.
- Type size: How large should the types of declarations be? Types need to be reduced to WHNF so making larger types will increase how often this needs to be done. However, we cannot use declarations in types as these are of type Bool, not Type, and we also cannot use declarations in if statements as we cannot be sure that they can be fully evaluated.
- Size and number of module parameters: As the different versions treat module parameters differently these are important properties to analyse.
- Module alias argument size: As different versions treat module aliases differently, this is an important property to analyse.
- Number of module alias arguments: Configuring this separately from the number of module parameters allows for the creation of partial aliases.
- Alias and import modifiers: Normally it is equally likely for the generator to select a normal declaration or a declaration in an alias and the same goes for selecting an identifier from the current file or from an imported file. These modifiers can change this behaviour.

With these two configurations combined, a wide variety of files can be generated. The generator makes use of the syntax of version 0 as well as its pretty printer to generate files that can be parsed by all of the Simple Agda versions. By plugging in a different pretty printer they can also be pretty-printed as normal Agda files so that they can analyse the current version of Agda. Any results gathered are not comparable to those of the Simple Agda versions but they could still be used to verify that Agda is similarly affected by increasing a specific parameter as version 0.

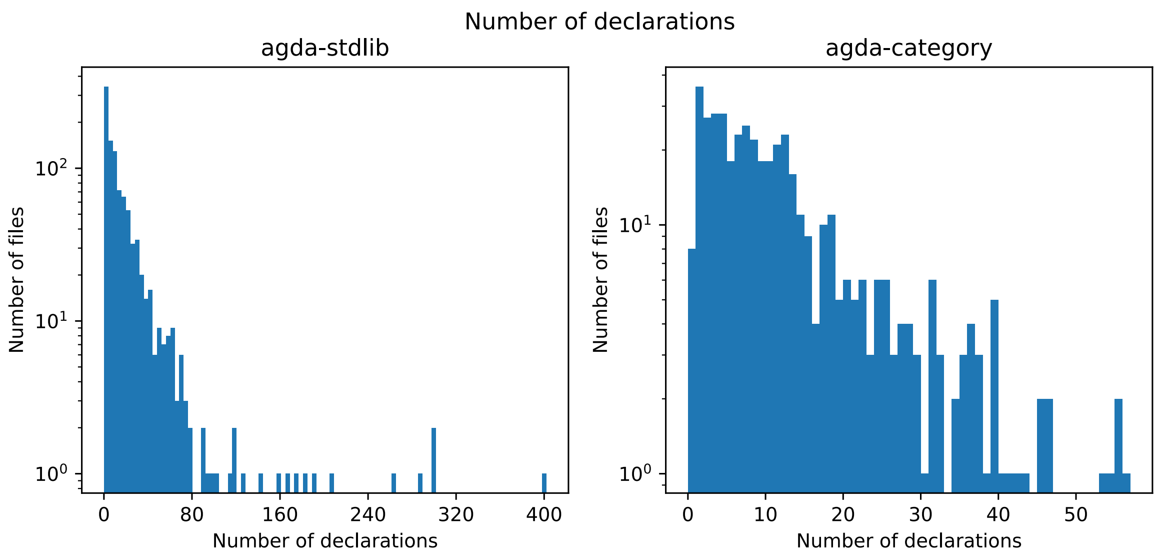

Figure 6.1: The number of decls in files in both agda-stdlib and agda-categories

6.4 Setting parameters to realistic values

Before we can generate files to use in our experiments we need to know what realistic numbers are to use for these experiments. This section will analyse existing libraries to find out things such as what a common number of declarations is, how many module parameters are used etc.

6.4.1 Analysis setup

For this thesis, two libraries were analysed: agda-stdlib [46] and agda-categories [4]. The standard library of Agda is a large library, but it doesn't make extensive use of module aliases. Agda categories is a lot smaller but makes much more use of Agda's module system and some of the original reports of Agda being slow in the presence of multiple aliases came from the development of this library as category theory is well-suited to using module aliases to for example create different categories.

To analyse these libraries we created a tool that makes use of Agda's parser and scope-checker to parse the files. Then instead of type-checking, we instead analyse the generated Abstract syntax tree (AST) to determine how often module aliases are used, how large module parameters get etc. This analysis is not perfect as Agda uses a rather complicated AST and this means that some sizes were estimated. This is no problem as these experiments will only be a guideline for the generator.

6.4.2 Analysis results

In this section, we will cover all of the collected data from both libraries. Note that all graphs use a logarithmic scale.

Number of declarations The first interesting data point is the number of declarations in a file. This is a somewhat difficult number to measure due to pattern matching, where clauses, as well as constructs such as data types and records which are not quite declarations. We have chosen to measure this number as the number of type signatures in a file. The results can be found in figure 6.1. Here we see that most modules are less than a hundred declarations with no modules exceeding 500 declarations. This indicates that even if people are creating

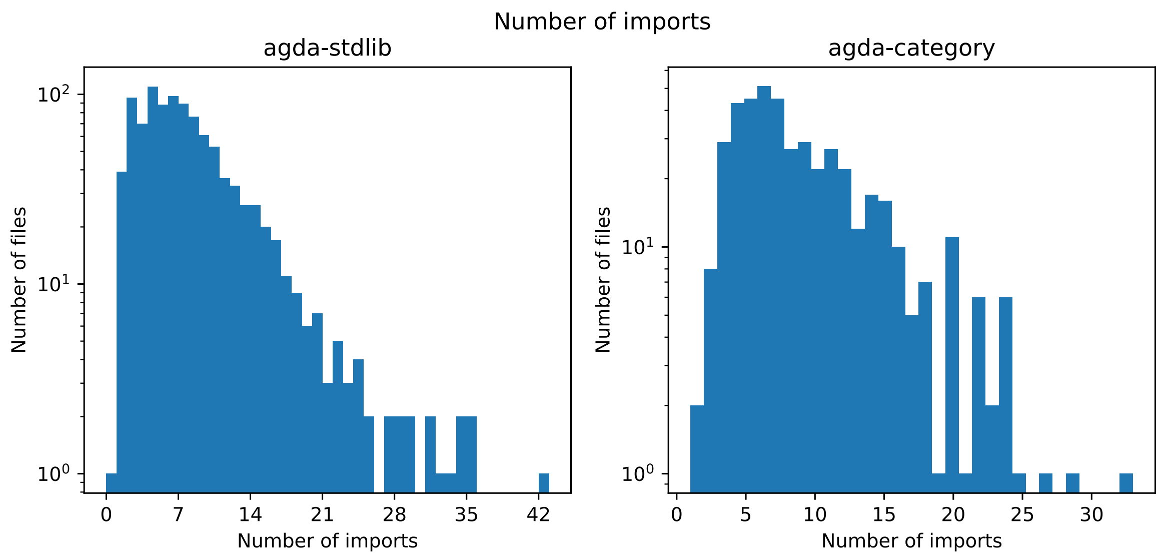

Figure 6.2: The number of imports in files in both agda-stdlib and agda-categories

smaller modules because of performance reasons, they would most likely never be more than a couple hundred declarations. So this is a good size for our experiments.

Number of imports Another important aspect to analyse is how often Agda files import other modules. One of the major changes between versions is what they output after type-checking. This only affects performance when files are being imported and we thus need to know how often this happens. From figure 6.2 we can see that imports are very common and we should thus take into account how importing affects the performance of different versions. We will not be generating 30 files to import however as accurately representing the dependencies between real files in generated files is not easy. Instead, we will only import a single file, but this still allows us to inspect the impact of imports on performance.

Size of module parameters The major difference between version 0 and version 1 is how they deal with module parameters. We thus need to know how large the types of these parameters get. This can be found in figure 6.3. The size of an expression is even harder to determine than the number of declarations as the Agda AST is quite complicated. These numbers should thus be seen as a rough estimate useful for generating terms of roughly similar size and complexity.

Size of module arguments The size of module arguments is also important. We know that the types of module parameters are quite large. However as can be seen in figure 6.4, module arguments are much smaller. This makes sense as these will often be either a literal or an identifier referring to a variable or earlier defined declaration with the few larger arguments being record instantiations.

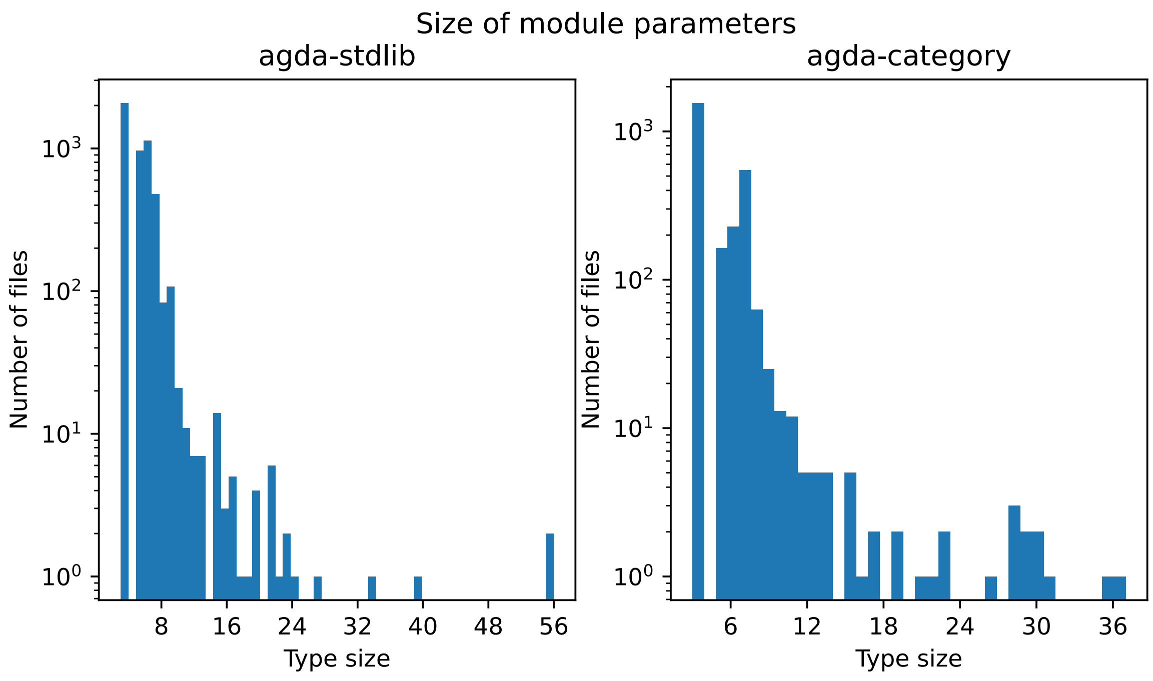

Figure 6.3: The size of module parameters in files in both agda-stdlib and agda-categories

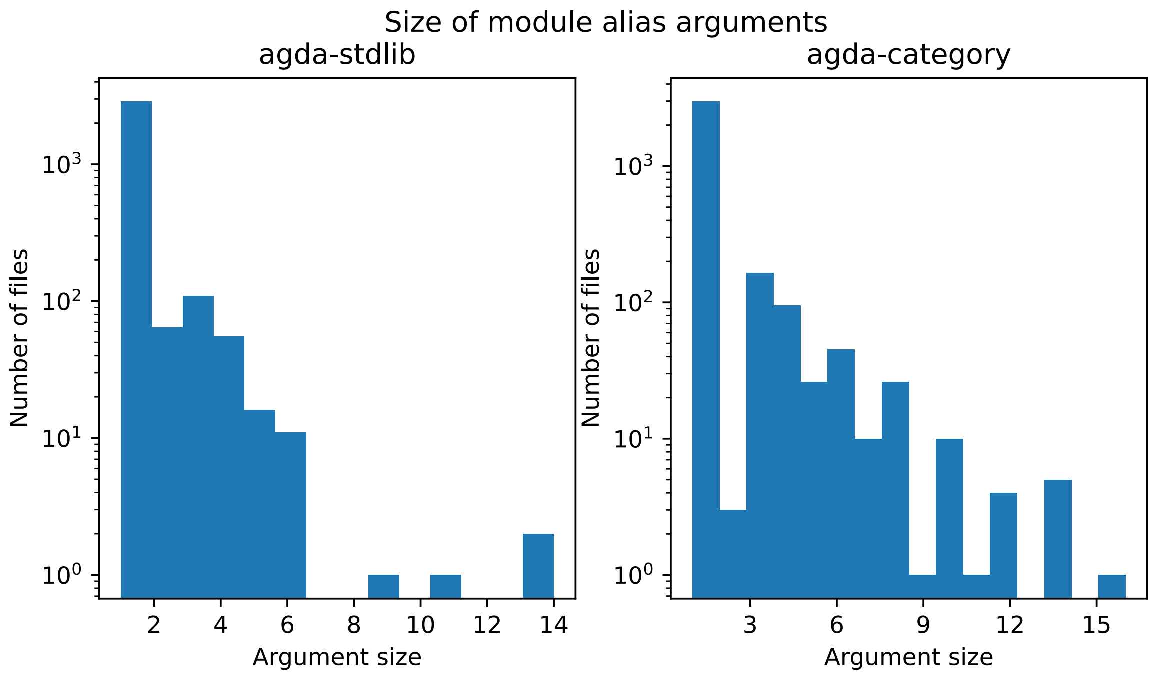

Figure 6.4: The size of module alias arguments in files in both agda-stdlib and agda-categories

Chapter 7

Evaluation

This chapter consists of two sections. Section 7.1 will cover the experiments for Simple Agda while section 7.2 will cover the experiments run for Agda. Both sections will discuss their own experiment setups as they differ quite a bit.

7.1 Simple Agda results

This section will analyse the performance of the different Simple Agda type-checkers using sets of randomly generated files. The results of these experiments are covered in section 7.1.2, each of the experiments will also be accompanied by a short explanation of why the results could be the way they are. A more thorough analysis of all of the results can be found in chapter 8. Before moving on to the experiments, section 7.1.1 will explain how the experiments were executed.

7.1.1 Experiment setup

This section will first explain how we time the code as well as the inherent problems with timing Haskell code. After that, we will explain how the experiments were executed.

7.1.1.1 Timing the code

To run any experiment at all we need to be able to time the execution of the code. Doing this in Haskell is more complicated than it is in other languages as we need to deal with laziness. While benchmarks in almost all languages have to prevent their experiment from being deleted through dead code elimination, Haskell also has the problem that only parts of the final term will be evaluated. Because all type-checking results are written to file this will not happen in our experiments. However, for our experiments, we are tracking more than just the total time. To get an accurate picture of the length of specific parts of the code we use deepseq [47] to force Haskell to evaluate terms fully after type-checking and scope-checking.

The code is timed using the timestats library [48]. With this library, it is easy to time specific parts of the type-checking process such as scope-checking, type-checking and serialization separately. This allows us to get a better idea of why one version is slower than another.

7.1.1.2 Running the experiments

The experiments were all executed on a 6 core Intel i7-8750H running at 2.20 GHz with 16 GB of RAM. To get accurate results we generated 15 random files for each experiment configuration, each file is then executed 50 times. To prevent some effects of caching we run the entire batch of files before we repeat any file.

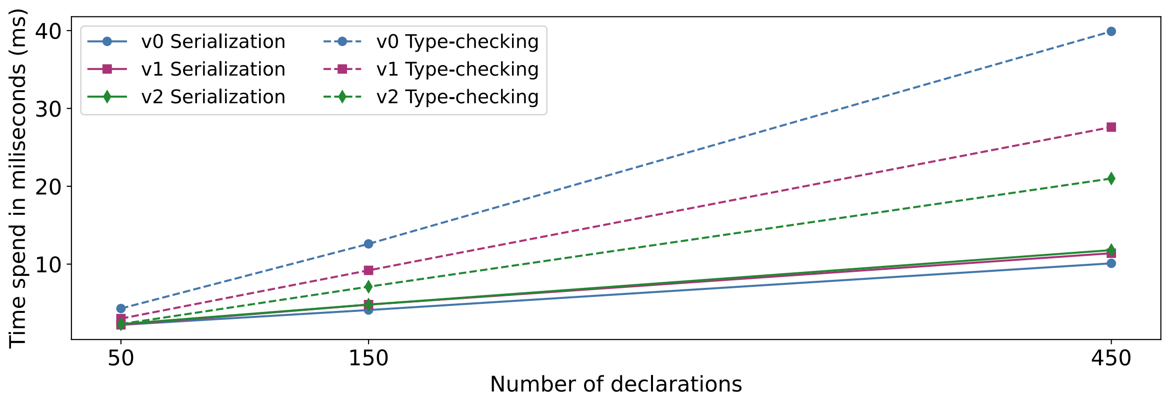

Figure 7.1: Experiment 1: Only declarations

The results of the different runs are then averaged. We use the mean as we have a low standard deviation and this ensures that the total time remains equal to the sum of all the other times. The times from the different files are then also averaged as the standard deviation between them is also very low. This will provide us with a single data point for each test configuration. We also collect the size of the output core files for each generated file and again average all the files generated with a single configuration. These filesizes can be found in table 7.10.

7.1.2 Results

This section will cover a variety of experiments used to compare the different versions. The first set of experiments are all single-file experiments. These will analyse the impact of specific features on the performance. After that, the experiments will look at the performance when importing files with and without using the core output in a variety of circumstances to analyse the differences between the core files of the different versions.

The graphs for the experiments will only contain the data needed to show the difference between the versions. The full experiment data as well as the generated files and their configurations can be found on GitHub[*].

7.1.2.1 Single-file experiments

Experiment 1: Only declarations The first experiment functions as a baseline. What if we generate files that have no module parameters nor any module aliases? Figure 7.1 shows that in that case, the serialization times are all equal. This makes sense as the output formats are roughly equal in this case. From table 7.10 we can even see that version 0 has the smallest core output. This is a side effect of the core output being stored in a human-readable form including indentation. As version 0 moves everything to the top level it has less indentation compared to the other versions and thus a smaller file.

There are however some differences in the speed of the actual type-checking. These differences are most likely caused by the complexity difference in implementation. Because version 0 needs to be able to make the most modifications to terms, it will always take slightly more time than a version that never makes such changes. Even in scenarios such as these were none of the changes are needed. This shows that a simpler type-checker is not only easier to work with but in our case also slightly faster.

Experiment 2: Impact of module parameters As versions 0 and 1 treat module parameters differently but aliases the same, it is important to look at module parameters separately from

[*] `https://github.com/ivardb/AgdaModuleImprovement/tree/master/benching`

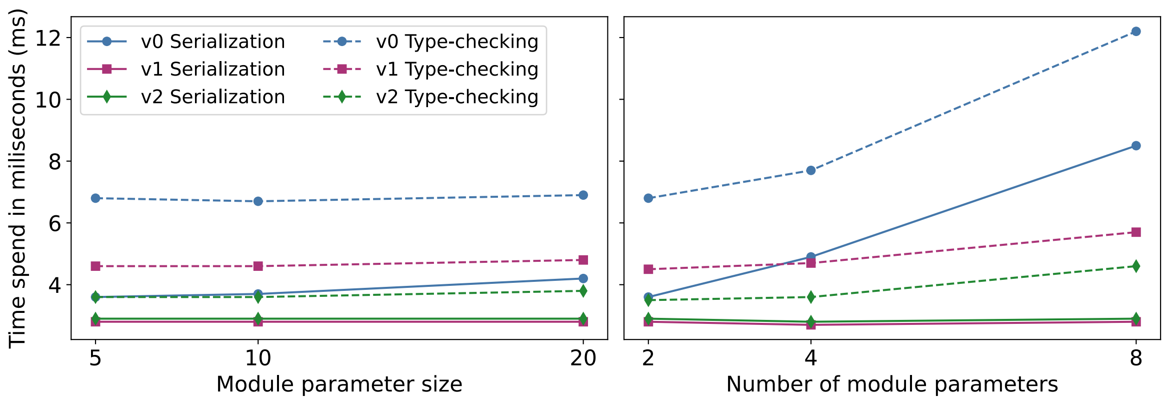

Figure 7.2: Experiment 2: Impact of both size and number of module parameters

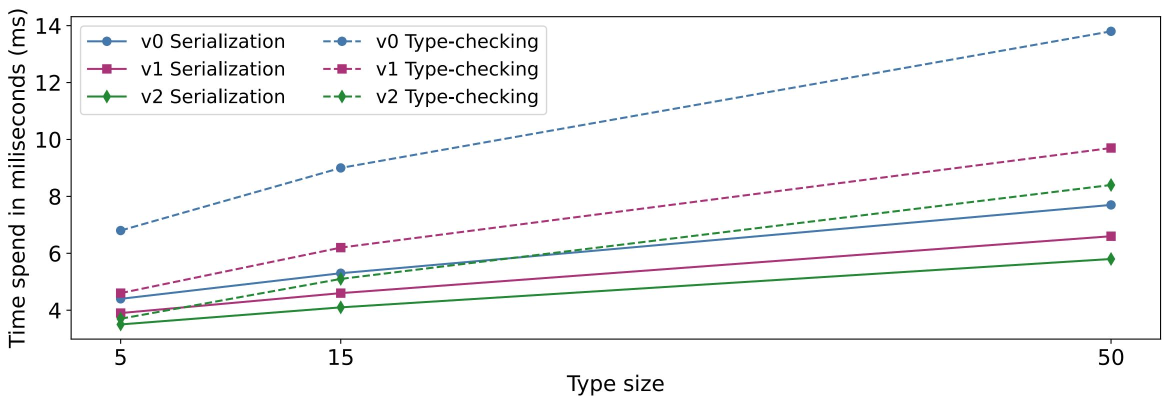

Figure 7.3: Experiment 3: Impact of type size

aliases. This experiment will investigate the effects of increasing both the number and the size of the parameters. From figure 7.2 we can see that module parameter size is not very impactful. Version 0 is a bit slower with module parameters present, but making them larger does not matter very much.

In contrast, increasing the number of parameters does significantly slow down version 0. This is most likely because we now need to wrap more parameters around the declaration creating additional AST nodes in the form of variable bindings and lambdas. From table 7.10, we can indeed see that the file size of the core output grows much faster when increasing the number of parameters, compared to increasing the size of the parameters even though the total size of all of the parameters combined is equal.

Experiment 3: Influence of increased type sizes with module aliases Due to the expansion of aliases, version 1 and version 2 will create copies of the types of declarations being aliased. This experiment will investigate how problematic this is. From figure 7.3 we can see that increasing the type size does not significantly increase the difference between versions. Clearly, this type of copying is not very important. This means that the fix for no longer copying types in issue #5499 [49] is not useful as this does not contribute meaningfully to the performance problems.

Experiment 4: Influence of the size of alias arguments This experiment will investigate the performance impact of using larger arguments when aliasing. As can be seen in figure 7.4, increasing the size has a significant effect on serialization time as well as a smaller effect on the type-checking time for version 0. So in contrast to type size, the argument size will

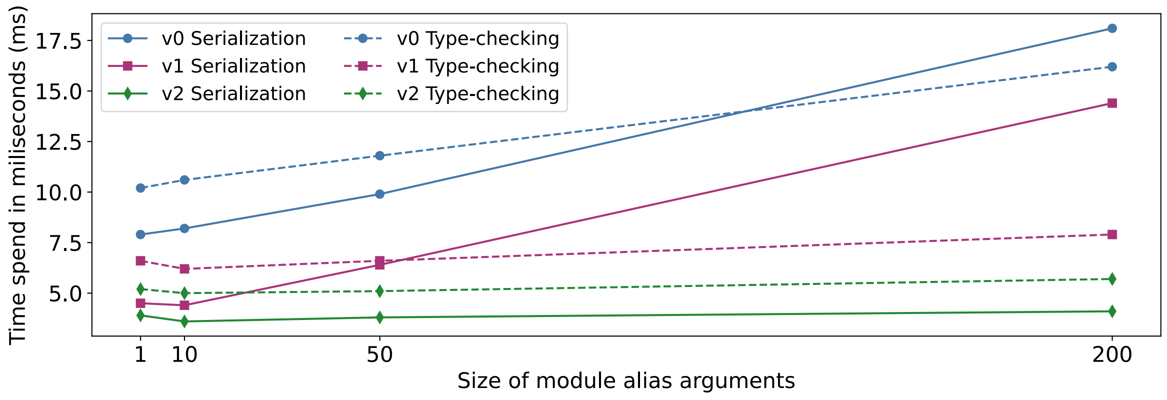

Figure 7.4: Experiment 4: Influence of the size of alias arguments

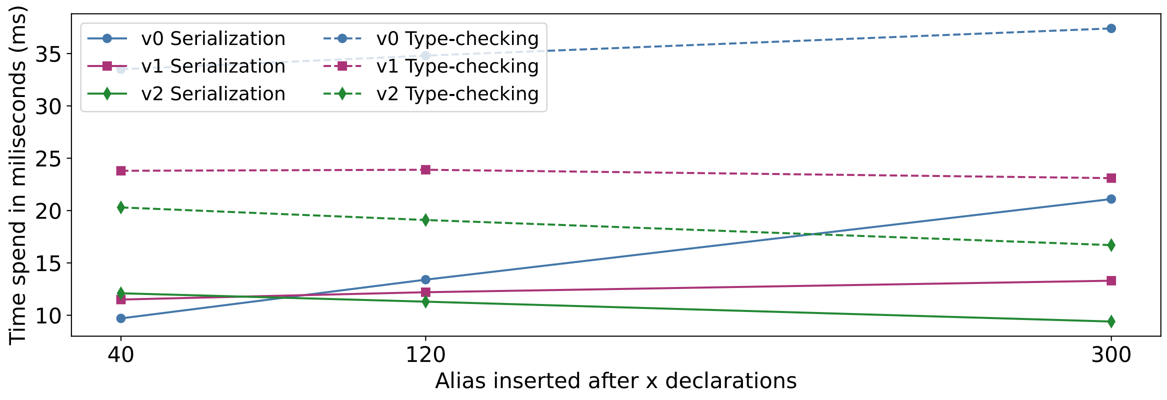

Figure 7.5: Experiment 5: Aliasing larger modules

significantly impact the performance of versions 0 and 1 while not affecting version 2. This matches with the file size changes observed in table 7.10. This makes sense as version 2 does not have to copy the arguments for each declaration it aliases and module arguments are only present for aliases, while the size of the types will also affect the original declarations.

It is however important to realise that from the analysis done in section 6.4, we know that these arguments are often very small in real applications and will thus stay below the sizes where this difference starts to become a problem, but there is no need for performance to be affected by argument size and it could be that the performance problems are the reason that the arguments remain small.

Experiment 5: Aliasing larger modules This experiment analyses the impact of aliasing larger modules. As module aliases will almost always be of modules with parameters, this experiment will also use them. In this experiment, we have 350 declarations in total. Somewhere in there, we will put an alias that covers all previous declarations. This will increase the number of declarations being aliased without increasing the total number of declarations.

From figure 7.5 we can see that versions 1 and 2 need less time to type-check files where the alias is inserted later. This is because there will be fewer function calls generated to the aliased declarations. (The impact of increasing the number of function calls is analysed further in experiment 7.)

We also see that the serialization times of version 0 increase as it needs to create more copies of the module parameters. The effect on version 1 is more minor as version 1 does not move the module parameters and only has to generate a few more declarations. Finally, we see that version 2 does not have an increase in serialization time as its core output is the simplest and does not grow based on alias size as can be seen in table 7.10.

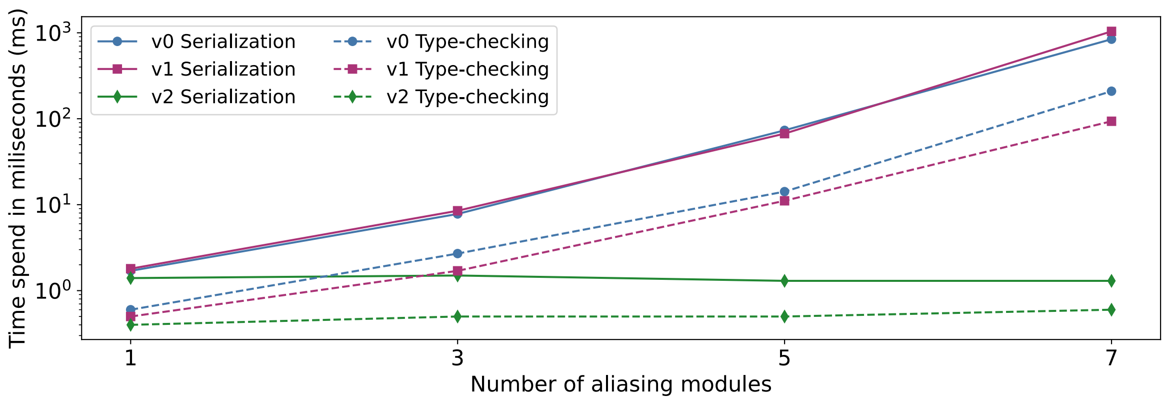

Figure 7.6: Experiment 6: Nested aliases

Experiment 6: Nested aliases While the previous experiment showed that version 0 is affected by the size of a module alias it was only a linear effect. This experiment will analyse nested aliases which can trigger exponential slowdowns. Many of these kinds of slowdowns have been reported by the developers of the Agda-categories library which has a complicated nested structure. However, due to its complexity it is hard to create an experiment from it. Instead, we will make use of a reduced example mentioned in one of the issues reporting the slowdowns [23].

For these experiments, we have a module `M0` with 5 declarations and a module parameter of type `Bool`. We also define a function `f` and `g` which are `const True` and `const False`. We can then create modules that look like this:

```
module M1(x: Bool) where
  module M = M0
  module N = M (f x)
  module O = M (g x)
```

As the previous module becomes part of the current module through the first module alias, this will trigger an exponential growth when aliases are expanded. From figure 7.6 we can see just how bad this exponential growth gets. (Note the log scale used.) In contrast to experiment 5, we see that in this experiment both version 0 and version 1 perform terribly. This shows that no longer moving around the module parameters can lead to improvements but does not fix the more serious problems. The filesizes shown in table 7.10 are a bit misleading when it comes to the difference between version 0 and 1 as core files are kept human-readable in Simple Agda, meaning version 1 has much more indentation making the file larger. If binary files were used this would not be the case.

Experiment 7: Impact of using aliased declarations The final property to investigate related to module aliases is how increased usage affects performance. Version 2 no longer creates the new declarations and should thus become slower when they are used more often as it has to generate the new types each time. This is indeed the effect we see in figure 7.7. Version 2 slowly loses its time advantage compared to the other versions. At 10x increased use its performance becomes roughly equal to version 1. Such an increased usage means that almost every term is a function call to an aliased declaration. This means that even in the worst case for version 2 it remains the fastest version.

This makes sense as creating new declarations for an alias has never been a problem. The real slowdowns are often caused by an increase in serialization time, lookup time or having to modify terms due to the moving around of parameters. So while version 2 now needs to create the declaration far more often than versions 0 and 1, this does not cost much time.

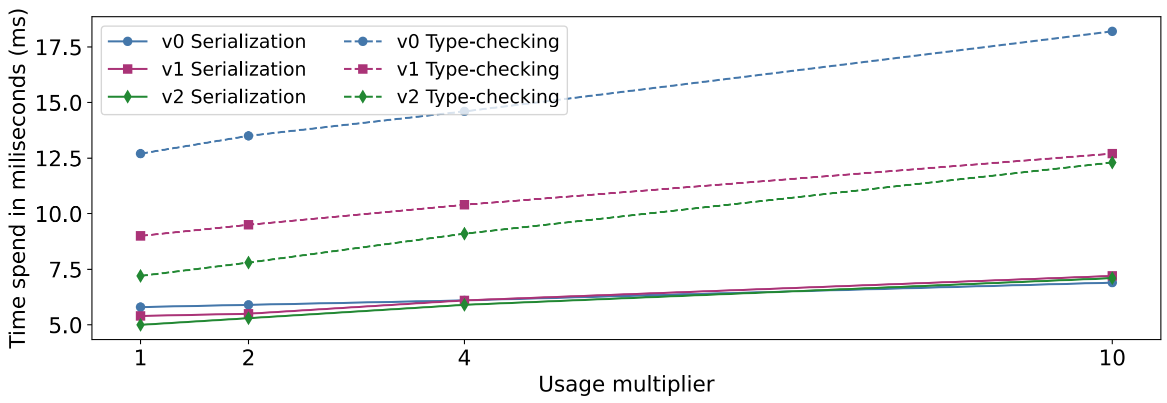

Figure 7.7: Experiment 7: Impact of using aliased declarations

The reason that the serialization times also converge is because the most time-consuming part of pretty-printing becomes the printing of the nested function calls instead of the increased amount of declarations due to the expanded aliases. From table 7.10 we can see that version 2 still has a smaller core file.

7.1.2.2 Multi-file experiments

In this section, we will evaluate the performance difference between having to type-check an imported file versus using the core output of type-checking when importing. These experiments will not be very realistic compared to the real Agda as Simple Agda does not support any real inference nor any complex features, nor does it use a core file format that is easy to de-serialize. This will make using the core output less beneficial than it would be for Agda. Instead, these experiments are meant to see how the changes to the core files impact the performance of versions 1 and 2 compared to version 0.

Multi-file experiment 1: Only declarations The first experiment again only uses simple declarations. In that case we see in figure 7.8 that version 0 is slightly faster compared to the other versions. This is because the qualified terms used in versions 1 and 2 take longer to parse and de-serialization takes up a large percentage of the total time. With a custom efficient serialization algorithm, this would not be the case.

Multi-file experiment 2: Import chains The previous experiment used files with barely any difference between versions. This experiment will instead use files with aliases which are treated very differently by the three versions. We will do this using import chains. We will create a base file with some declarations and an alias. Then we create a test file that makes use of the declarations in this alias. To extend the chain we can add files in between them that alias the alias from the previous file in the chain, allowing the test file to use this alias instead.

In many languages expanding the in-between aliases would lead to a clear benefit as it would eliminate the need to load the transitive imports. However, as Agda is dependently typed we also need access to the definitions and thus versions 0 and 1 still need to load the transitive imports.

Figure 7.9 shows that version 2 is by far the fastest in the presence of such transitive imports as all versions need to load all of the transitive imports, but version 2 can do this much faster as its core files are smaller, as can be seen in table 7.11. As version 2 does not increase in size we can see from figure 7.9 that version 2 is now also the fastest at de-serialization. Version 1 is the slowest as it has more complicated module paths and also expands module aliases.

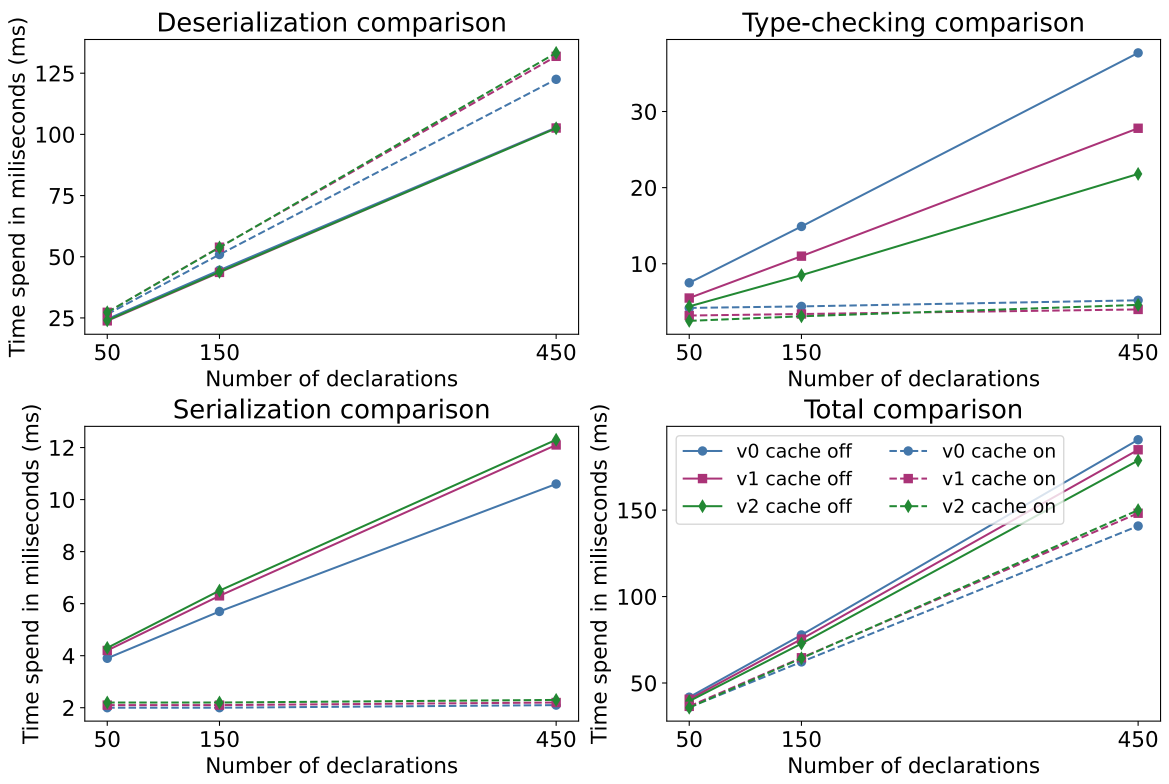

Figure 7.8: Multi-file experiment 1: Only declarations

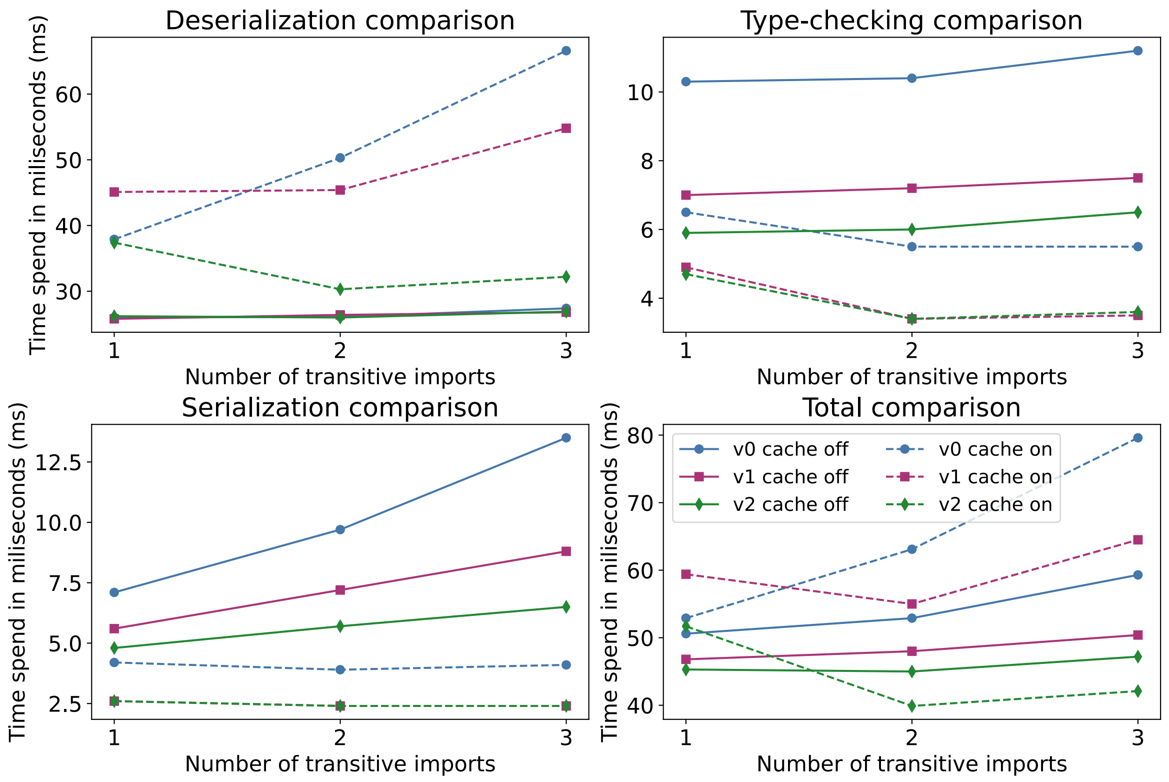

Figure 7.9: Multi-file experiment 2: Import chains

Experiment:	Version 0	Version 1	Version 2
Experiment 1: 50 declarations	15.0 KB	15.7 KB	15.7 KB
Experiment 1: 150 declarations	46.0 KB	48.4 KB	48.4 KB
Experiment 1: 450 declarations	141.4 KB	148.6 KB	148.6 KB
Experiment 2: 2 parameters	25.3 KB	23.0 KB	23.0
Experiment 2: 4 parameters	29.9 KB	23.0 KB	23.0 KB
Experiment 2: 8 parameters	41.9 KB	26.3 KB	26.3
Experiment 2: Size 5 parameters	24.9 KB	22.5 KB	22.5 KB
Experiment 2: Size 10 parameters	25.9 KB	22.7 KB	22.7 KB
Experiment 2: Size 20 parameters	29.8 KB	22.9 KB	22.9 KB
Experiment 3: Types of size 5	35.3 KB	31.1 KB	25.3 KB
Experiment 3: Types of size 15	44.0 KB	39.3 KB	31.1 KB
Experiment 3: Types of size 50	62.7 KB	58.0 KB	43.9 KB
Experiment 4: Arguments of size 1	52.1 KB	37.6 KB	29.3 KB
Experiment 4: Arguments of size 10	55.0 KB	40.4 KB	29.7 KB
Experiment 4: Arguments of size 50	67.7 KB	55.8 KB	30.3 KB
Experiment 4: Arguments of size 200	107.9 KB	101.3 KB	31.1 KB
Experiment 5: Alias after 40 declarations	117.1 KB	114.1 KB	107.2 KB
Experiment 5: Alias after 120 declarations	144.7 KB	131.9 KB	110.3 KB
Experiment 5: Alias after 300 declarations	202.1 KB	165.4 KB	110.8 KB
Experiment 6: 1 module with aliases	6.0 KB	5.0 KB	2.0 KB
Experiment 6: 3 modules with aliases	52.0 KB	54.0 KB	3.0 KB
Experiment 6: 5 modules with aliases	506.0 KB	552.0 KB	3.0 KB
Experiment 6: 7 modules with aliases	4892.0 KB	5585.0 KB	4.0 KB
Experiment 7: 1 time multiplier	54.9 KB	50.7 KB	43.5 KB
Experiment 7: 2 times multiplier	56.9 KB	52.6 KB	45.1 KB
Experiment 7: 4 times multiplier	60.5 KB	55.4 KB	48.3 KB
Experiment 7: 10 times multiplier	70.1 KB	66.1 KB	58.8 KB

Table 7.10: Average size of the core files outputted during type-checking in the various experiments

Experiment:	Version 0	Version 1	Version 2
Experiment 1: 50 declarations	6.6 KB	7.0 KB	7.0 KB
Experiment 1: 150 declarations	20.8 KB	21.9 KB	21.9 KB
Experiment 1: 450 declarations	62.9 KB	66.4 KB	66.4 KB
Experiment 2: Length 1 chain base	11.6 KB	11.4 KB	7.2 KB
Experiment 2: Length 2 chain base	10.0 KB	8.0 KB	5.1 KB
Experiment 2: Length 2 chain in-between	2.7 KB	3.0 KB	0.0 KB
Experiment 2: Length 3 chain base	9.1 KB	6.2 KB	4.0 KB
Experiment 2: Length 3 chain in-between 1	2.6 KB	2.2 KB	0.0 KB
Experiment 2: Length 3 chain in-between 2	2.0 KB	2.2 KB	0.0 KB

Table 7.11: Core file size of imported files in various multi-file experiments

7.2 Agda results

To verify the accuracy of version 0, we will execute some of the experiments on Agda itself to see if version 0 behaves the same as Agda.

7.2.1 Converting the experiments to Agda

The experiments in the previous section were all generated for Simple Agda using the generator from chapter 6. To create these same experiments in Agda we parsed the files to Simple Agda and then pretty-printed them to Agda syntax. However, Simple Agda supports a number of primitives that Agda does not support, so we need to make some changes to the files.

First, we need to add support for `Unit` and `Bool`. Simple Agda supports this natively while Agda does not. It does come with built-in definitions for them, however, we cannot use these directly as they function as actual imports in Agda's benchmarking which we do not want. Instead, we have copied their definitions including the `BUILT-IN` pragma's as these will affect performance.

We also need to support if expressions. For this, we used the following definition:

```
if_then_else_ : ∀{i}{A : Set i} -> Bool → A → A → A
if true  then t else f = t
if false then t else f = f
```

Note that we use a definition that works for all levels of `Set`. We need this as Agda does not have `Set` in `Set` and as such we will have if expressions that need to work in `Set` and some that work in `Set1`.

The final primitive that we need are type annotations. Agda does not support type definitions in arbitrary locations. Luckily type annotations can be created quite easily in Agda:

```
_∋_ : ∀{i}(A : Set i) →A → A
A ∋ x = x
```

This definition allows us to give the type of an expression whenever we want just like in Simple Agda.

These changes will work for single-file experiments but will cause problems with imports as we will have multiple data structures that use the `BUILT-IN` pragmas for `Bool` and `Unit` which is not allowed. We fix this by adding an additional open import for `Bool` and `Unit` whenever we import a file and in that case only adding the definitions to the base file.

A final change that we need to make is to use an operator similar to the $ in Haskell whenever we generate an application of a lambda. Without the $, Agda will automatically beta reduce this application before type-checking and we want to avoid such optimizations as they will be more efficient on some files, thereby messing up the comparisons.

7.2.2 Experiment setup

The experiments are executed using the Agda-2.6.2.2 executable [3]. While Agda can be used as a Haskell library this reduces its performance significantly. So instead we execute Agda with the "-vprofile:7" flag which will activate benchmarking of different phases such as type-checking and serialization. To bring the timings more in line with those of Simple Agda, we combine the parsing and de-serialization times as Simple Agda does not differentiate between these.

The Simple Agda files used the mean for combining the repeated experiment runs. For the Agda experiments we instead made use of the median as the Agda experiments had much more extreme outliers.

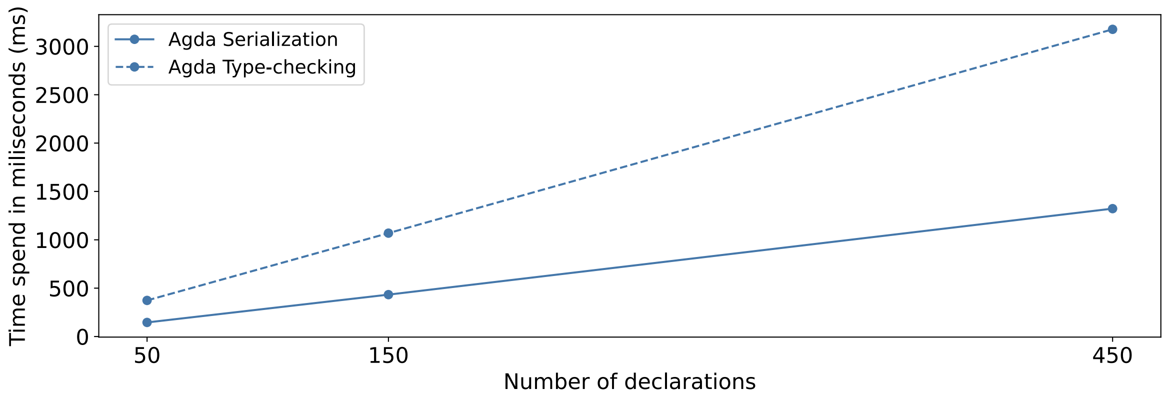

Figure 7.12: Agda experiment 1: Only declarations

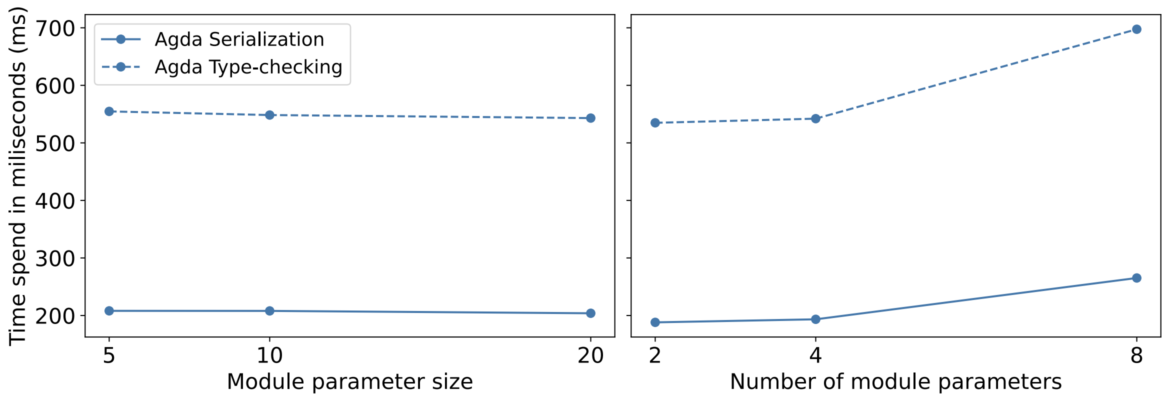

Figure 7.13: Agda experiment 2: Impact of module parameters

7.2.3 Results

This section will go over some of the experiments to see how Agda behaves. This will allow us to evaluate how accurate of a representation version 0 is. Some experiments will be left out such as experiment 7 which was designed to evaluate the weaknesses of version 2 and as such it is not useful for comparing version 0 to Agda.

Agda experiment 1: Only declarations From figure 7.12 we can see that with normal declarations Agda behaves the exact same as our versions. We can also see that Agda is much slower than our versions. This is partly because Agda is much more complicated. Just like version 0 is slower than version 2 in experiment 1 because it needs to support more transformations, so is Agda much slower as it needs to support transformations for all of Agda's features.

Agda experiment 2: Impact of module parameters This experiment shows that version 0 is not a perfect representation of Agda. Experiment 2 showed that version 0 is very sensitive to the number of parameters when serializing. Figure 7.13 shows that for Agda there is only a minor difference during serialization compared to simply increasing the size of parameters. We do see that for larger numbers of parameters, the type-checking time starts to increase. This is most likely because of the increased amount of manipulations needed when dealing with declarations with more parameters.

Agda experiment 4: Influence of the size of alias arguments The results in figure 7.14 match the results from version 0. Increasing the size of the module arguments will signifi-

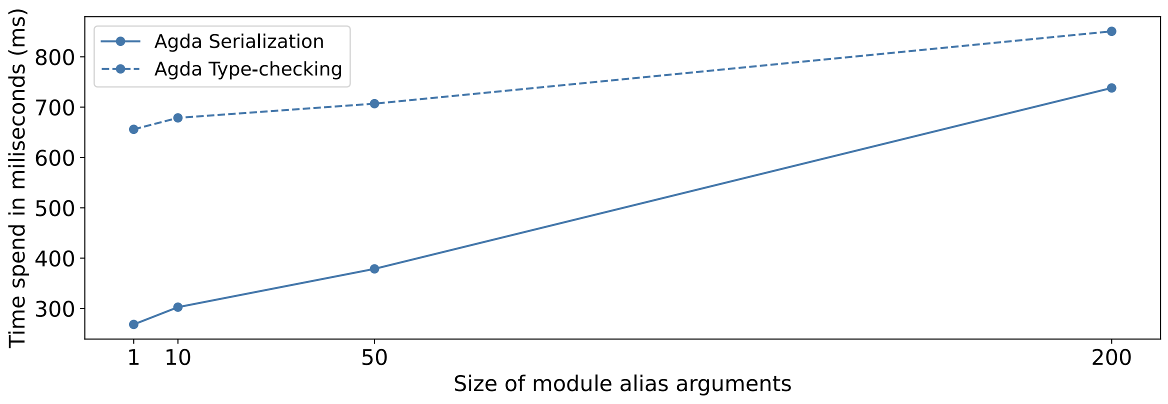

Figure 7.14: Agda experiment 4: Influence of the size of alias arguments

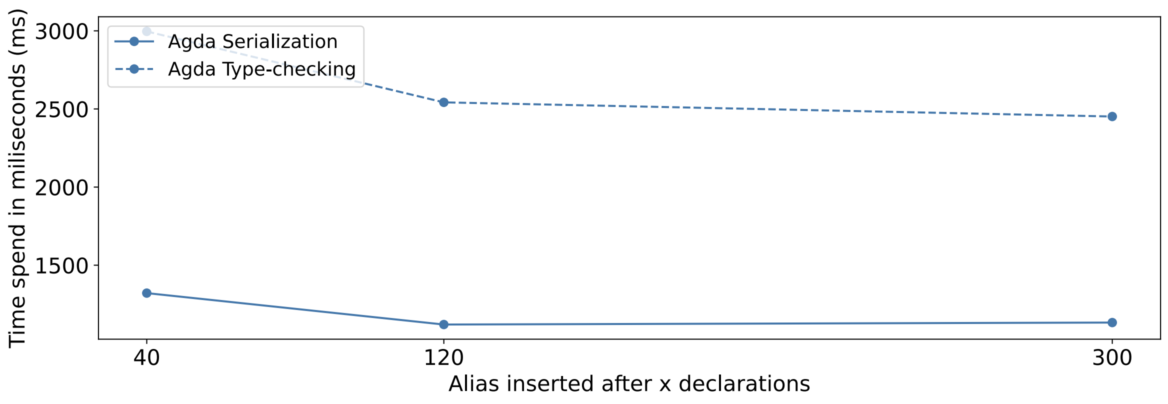

Figure 7.15: Agda experiment 5: Aliasing larger modules

cantly increase the time needed for serialization. What is surprising however is how much the type-checking is affected. This is not something we saw happen with version 0. This seems to indicate that Agda does more unnecessary work with module arguments than just copying them to the new declarations.

Agda experiment 5: Aliasing larger modules While version 0's serialization was affected significantly by increasing the module alias size we do not see the same effect for Agda in figure 7.15. It behaves similar to the way version 2 performed in this experiment.

There are several possible explanations for this. Either serializing function calls is more time-consuming than serializing a couple of short declarations or Agda has another optimization in its file format that deals with this. Agda uses a custom binary file format with a lot of compression that could also reduce the serialization time in this case. The type-checking difference can be explained by the fact that the generator produced simpler declarations when it cannot call functions from an alias. Thus type-checking becomes easier, the fewer declarations there are after the alias.

Agda experiment 6: Nested aliases While the previous experiment showed that Agda is not sensitive to the number of declarations in a module alias, we can see from figure 7.16 that it still suffers extremely badly when we start nesting aliases. Adding two more sets of aliases adds two orders of magnitude to the runtime. This is clearly a huge problem that needs to be addressed.

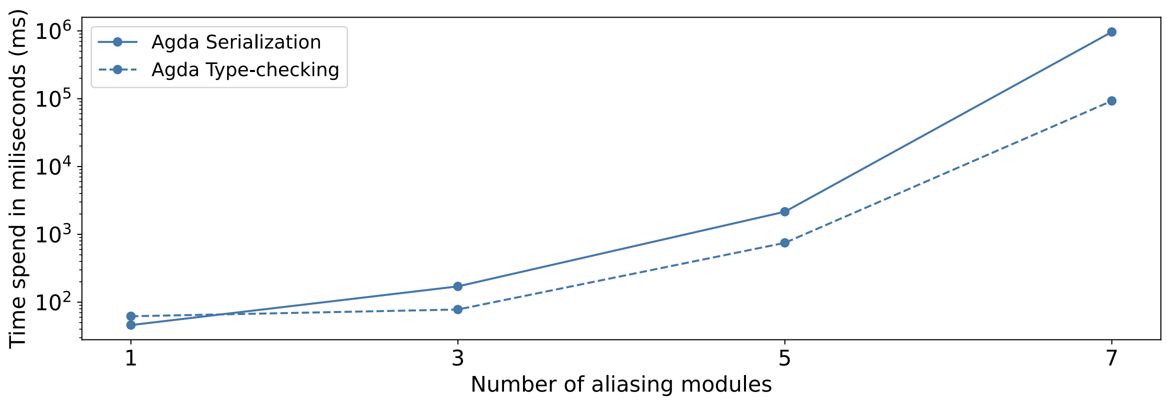

Figure 7.16: Agda experiment 6: Nested aliases

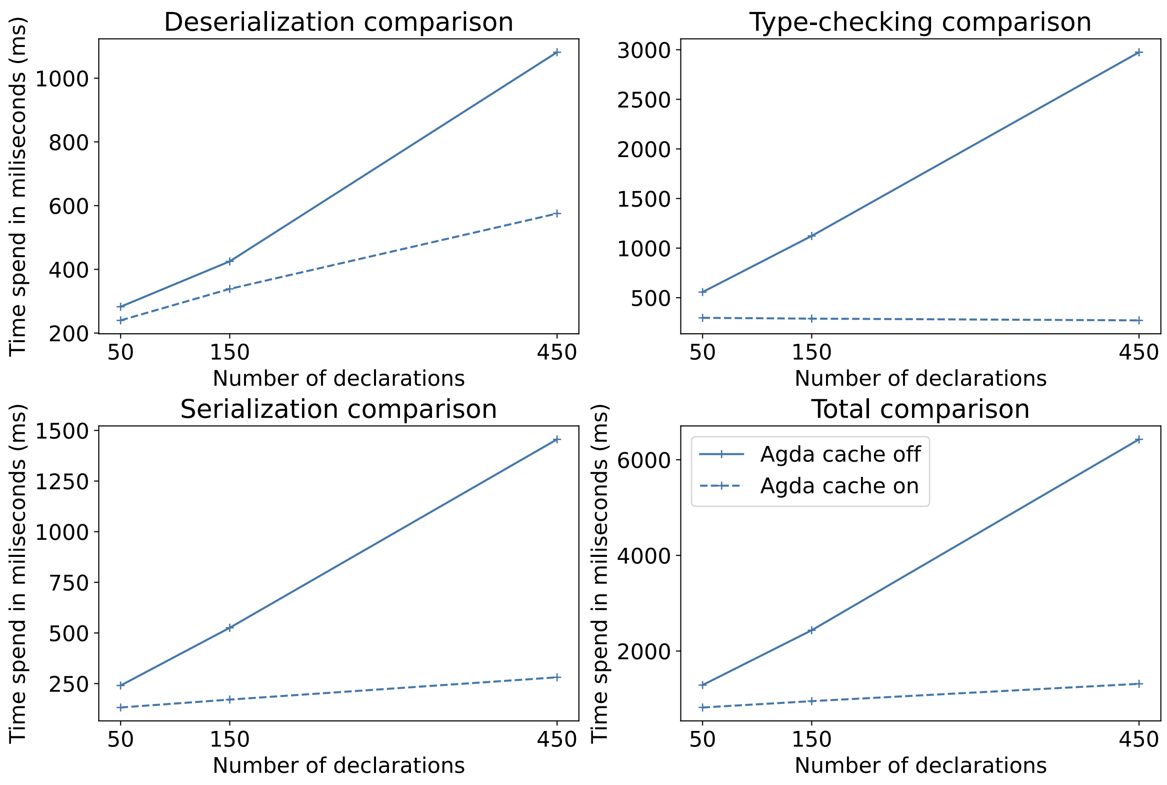

Figure 7.17: Agda multi-file experiment 1: Only declarations

Experiment:	agdai file size
Experiment 1: 50 declarations	127.4 KB
Experiment 1: 150 declarations	353.6 KB
Experiment 1: 450 declarations	1078.2 KB
Experiment 2: 2 parameter	160.3 KB
Experiment 2: 4 parameters	158.0 KB
Experiment 2: 8 parameters	177.3 KB
Experiment 2: Size 5 parameters	161.1 KB
Experiment 2: Size 10 parameters	159.9 KB
Experiment 2: Size 20 parameters	161.8 KB
Experiment 4: Size 1 arguments	190.5 KB
Experiment 4: Size 10 arguments	191.3 KB
Experiment 4: Size 50 arguments	198.0 KB
Experiment 4: Size 200 arguments	207.2 KB
Experiment 5: Alias after 40 declarations	757.7 KB
Experiment 5: Alias after 120 declarations	755.6 KB
Experiment 5: Alias after 300 declarations	713.6 KB
Experiment 6: 1 module with aliases	28.0 KB
Experiment 6: 3 modules with aliases	72.0 KB
Experiment 6: 5 modules with aliases	527.0 KB
Experiment 6: 7 modules with aliases	5724.2 KB

Table 7.18: File sizes for Agda experiments

Agda multi-file experiments For the multi-file experiments, we will only look at the first experiment so that we can see how much Agda benefits from using its interface files. We can see from figure 7.17 that in Agda the serialization and de-serialisation times are equal. This makes sense as it uses a custom algorithm for it. This means that the increase in de-serialization time for versions 1 and 2 will most likely not affect Agda as the serialization can easily be adapted for this. We also see that serialization and de-serialization is roughly half of the run-time. This means that shorter core files will also have a nice effect on the total runtime, especially given how common imports are in Agda.

Chapter 8

Discussion

This chapter will discuss and analyse the results of the experiments on both Simple Agda and Agda. Section 8.1 will first go through the Simple Agda experiments. After which section 8.2 will discuss how the Agda experiments affect the conclusions from the Simple Agda experiments. Next, section 8.3 will discuss any remaining limitations of the experiments. Finally, section 8.4 will explain what we believe should be done to improve Agda.

8.1 The Simple Agda experiments

This section will make a number of observations from the Simple Agda experiments starting with the observation that version 0 is always a bit slower when type-checking than version 1 which in return is slower than version 2. This shows that the type-checking phase is impacted by the complexity of the type-checker even if that complexity is not necessary. For example, version 0 needs to be able to move around parameters. Because of this, it is slower than version 1 even if there are no parameters, simply because the code becomes more complex.

The second observation is that version 0 is significantly affected by the number of module parameters present. From experiments 2 and 5, we can see that there is a significant difference in serialization time between version 0 and version 1 showing the impact of module parameters. From experiment 2 we can also see that the number of module parameters is more impactful than the size of the parameters. This makes sense as we need to wrap more parameters around declarations which requires more variable bindings and lambda nodes.

The third observation is that the number of declarations created through aliasing is not too impactful until you start nesting aliases. In experiment 5 the difference between version 1 and version 2 is very small, showing that the alias expansion was not very impactful. However, from experiment 6 we can see that if we create a proper nesting of aliases the difference between version 0 and version 1 becomes minimal as they are both far slower than version 2. This shows that not messing around with module parameters can lead to some positive effects in small cases but that the worst cases are caused by the explosion of the number of declarations.

The fourth observation we can make is related to experiment 4 which shows that module alias arguments are very impactful on the performance. This makes a lot of sense, while we previously mentioned that the generated declarations for aliases are very simple, this depends heavily on the size of the module arguments as these make up the majority of the declaration. While we know from the analysis performed in section 6.4 that module arguments tend to be simple variables, this might be because users have realised the performance impact of using anything else.

From the multi-file experiments, we can observe that the simpler core files of the later versions are better than the core files of version 0. They take less time to serialise and deserialise and perform just as well for type-checking.

The final and most important observation to make is that version 2 performs better in all experiments and across all statistics. This shows that it is a clear improvement over version 0. While the constant time improvement in type-checking is useful, the most important improvement is that version 2 only scales with the number of declarations in the input and not the type of declaration.

8.2 Impact of the Agda experiments

In this section, we will analyse the experiments run on Agda and see how these affect our previous observations.

The first observation we can make on the Agda experiments is that type-checking in Agda takes much longer. We previously observed that version 0 is slower in type-checking partly because it is performing more complex manipulations as it needs to move around module parameters. Agda is far more complex compared to version 0 and as such it needs much longer to deal with normal declarations as can be seen in experiment 1 where it already needs a couple of seconds for just type-checking the larger files.

The second observation is that the number of parameters is still slightly more problematic than the size of the parameters as can be seen in experiment 2. However, the difference for serialization is much smaller than for Simple Agda while the difference in type-checking is larger. One explanation is that the type-checking is more affected because Agda has different ways of dealing with variable binding that cause additional slowdowns. However, the precise reason is hard to pin down due to Agda's complexity. The reason that Agda does not have a similar explosion in serialization time is more easily explained. Agda uses a custom compression algorithm when serializing that is meant to prevent slowdowns due to excessive duplication of values. This could be very effective at removing the problems that version 0 experienced in this experiment.

The third observation we can make is that Agda seems to be less sensitive to the size of module aliases as can be seen in experiment 5. The most likely reason for this is that the time increase from having more declarations is much smaller than the time decrease due to fewer complex function calls. This is a similar reason as to why the performance of version 2 improved with larger aliased modules. Agda is still however very sensitive to the exponential explosion that happens with nested aliases as can be seen in experiment 6. Its performance in this experiment is even worse than that of version 0 as its runtime is exponential even when plotted on a logarithmic scale.

The fourth observation is that Agda is still sensitive to module argument size as can be seen in experiment 4. Whatever the reason is that Agda became less sensitive to the impact of the number of module parameters this does not apply to all forms of copying. This is a problem that is present in both version 0 and 1 but not in version 2 and would thus likely be fixed if Agda switched to this module system.

The final observation relates to the multi-file experiments. Previously we noted that Simple Agda is too quick at type-checking compared to its serialization performance to notice the benefits of caching. This is absolutely not the case for Agda.

8.3 Remaining limitations of the experiments

While some of the limitations of the experiments have been mitigated by verifying them on Agda itself there are still some clear limitations remaining. This section will go over them and discuss their impact on the conclusions of this thesis.

8.3.1 Limitations of using generated files

One of the major limitations of the experiments are the files used in the comparison. While generated files are quite good at isolating a specific modification they are far from realistic files. So while these files allow us to analyse the behaviour of the different versions under different circumstances, these circumstances are not necessarily realistic. For example, many of the terms generated will not be in WHNF while human-written code is almost always in WHNF. This already caused problems with lambdas being automatically beta-reduced, requiring us to introduce a $ operator to prevent this optimization. Other such optimizations might still be present which will affect the accuracy of the results.

8.3.2 Limitations of Simple Agda

Another limitation of the experiments is that there could be a feature of Agda that is affected aversely by the changes, but that is not present in Simple Agda. For example, the Agda rewrite rules could potentially be affected because looking up definitions takes more time and substitutions in the changed versions. While we do not believe that there are any such side-effects with a big enough impact to matter, it could happen. Although if it happens, it could probably be solved by implementing a form of caching that keeps any generated declarations for later use. This would solve any potential performance issues without immediately generating an exponential number of declarations.

8.3.3 Limitations of my Haskell knowledge

The final problem to discuss is the quality of the Haskell code of the type-checkers. My experience and knowledge of optimizing Haskell code is rather limited. This means that some of the versions could perform worse than they should due to mistakes in programming them. This could be an explanation for why Agda sometimes behaves better than version 0. On the other hand, the complexity of implementing version 0 properly is another good argument for switching to a module system such as version 2 which is much easier to work with and of course, no amount of programming mistakes can compensate for the asymptotic complexity advantages of version 2 when it comes to dealing with module aliases.

8.4 Advice for improving Agda's module system

This section will cover two different topics. First we will explain why Agda should switch to a module system similar to version 2 as soon as possible. Then, we will explain what we think should be done afterwards to continue improving the module system and Agda in general.

8.4.1 Module system changes outlined in this thesis

We believe that Agda should switch as soon as possible to a module system that is close to version 2. From the experiments we have seen that it generally performs better than the other versions we have tried and avoids the exponential growth problems that the other versions have with nested module aliases.

Besides the performance argument, switching to version 2 will also simplify the handling of module parameters which in the past has led to quite a few bugs [15–17]. It will also allow for open public statements to desugar to module aliases without a performance decrease and finally, it could benefit the pretty-printing issues Agda experiences as the pretty-printer has access to a signature with more information as the modules and aliases are kept intact.

The disadvantages of version 2 are quite minimal. From the experiments, we saw that serialization and de-serialization were impacted by the increased complexity of using term-qualified names. We don't think Agda will have this issue, as it already uses a custom-made serialization algorithm which should minimize the impact of more complicated names.

There are also a few feature interactions that need to change when switching to version 2. As modules are no longer lifted to the top level the handling of instances found in modules might need to be refactored as well to allow the algorithm to search through a nested structure. This should not be more difficult, it simply needs to be redesigned.

The same goes for erased modules. We can no longer handle the erasing of module aliases by applying the erasure to the generated declarations. This will now have to be done during lookup. This should not be too difficult either.

A feature that might be more difficult for version 2 is an open proposal for allowing modules to be extended [50]. As this proposal has already been put in the icebox, we don't believe this to be a problem either as it is unlikely this feature will ever be added to Agda.

Finally, the system could be updated to version 3 as well to allow for slightly better pretty printing and some nice shorthand for users to use. If term-qualified names are used to implement version 2 then switching to version 3 requires no big changes to the type-checker and this change could thus happen later if it needs to be discussed first.

8.4.2 Future steps

We believe there are three remaining steps needed to fully fix Agda's module system. We will go over all three.

8.4.2.1 Breaking module system changes

There are two features of Agda's module system that spark questions again and again. Open public and anonymous modules. For open public the consensus seems to be that it needs to be changed to act the same as a module alias [29]. This change has not been made yet due to it being a breaking change and module aliases having terrible performance.

Anonymous modules behave weirdly at the moment in the sense that using an underscore as a module name has completely different behaviour compared to using an underscore as the name of a declaration. For a module, it will scope all declarations inside as if they exist in the outside scope while for declarations the declaration can simply not be referenced. In my opinion, anonymous modules should also be hidden from the outside scope. A new syntax should be introduced to support the current behaviour of anonymous modules.

As both of these changes are breaking, they should be performed at the same time to minimize the number of version updates with breaking changes. As open public is something that can be fixed once the performance refactor is complete, it would in my opinion be a good idea if anonymous modules were actively discussed at the moment so that their behaviour can be finalized by the time the performance refactor is finished.

8.4.2.2 Pretty-printing improvements

Once the inverse scope lookup and display form systems have been been updated to the new module system it can be investigated what the next steps related to pretty-printing are. A number of approaches has already been outlined in section 5.3.2.2 and after the performance refactor it can been analysed which of these features users want as well as what steps are still necessary to deal with the remaining bugs. At the moment this is hard to determine as pretty-printing issues could be caused by the poor module representation, bugs in the display forms, fundamental issues with the current display form algorithm etc.

8.4.2.3 Record changes

The last feature related to modules that is in need of further analysis is the record system. The record system supports many annotations to get the users their desired behaviour. However, many changes have been made to records and termination checking since these annotations were introduced and they are in need of careful analysis and documentation to see which annotations are still useful and how they interact with each other [38]. There are also some more fundamental problems with the record system which could perhaps be analysed at the same time [35] and finally, some features might have a more efficient implementation now that the module system has been changed. All of these issues should be analysed to see how records can be improved.

Chapter 9

Related work

This thesis started with an unpublished, early draft by my supervisor Jesper Cockx [43]. Besides this draft there are no papers that this work is directly based on. Instead this chapter will describe several different module systems and how Agda compares to them. After that, we will cover exponential growth problems experienced by other module systems similar to those experienced by Agda to explain why our problem is different and cannot use the same solutions. Finally, we will cover some research that could be relevant to improving Agda's module system in future work.

Module systems in general One of the most prevalent module systems in research is that of ML [51, 52]. ML modules are parametrised and can be typed using signatures. This allows for the definition of functors which are module definitions parametrised over a module matching a specific signature. This allows for the creation of new modules given an existing module with a specific signature.

It has been shown that almost any language can be extended with an ML-like module system [53] and while few languages have this exact system, almost any language has a system that fulfils similar needs. For example, object-oriented languages like Java provide classes and interfaces which similarly allow programmers to create types for classes by using interfaces that can then be used to define other classes through a constructor. Generics can also be used to create parametrised methods or classes although these tend to be much weaker than those in a language like Agda as there is no dependent typing.

Another approach to module systems is that of Haskell with type classes. Type classes might not seem like a module system as they focus almost purely on polymorphism but the system is extremely powerful and is in fact equivalent to ML's modules [54].

Yet another interesting set of module systems are Mixin systems or hole-based systems [55–57]. These systems allow programmers to define both concrete definitions and declarations that need to be imported from another module. Modules can then be combined to satisfy the needs of each other in a potentially cyclic manner. Like Haskell type classes, this system has a different focus compared to ML, but it can still be used to encode ML functors [56] or be combined with Haskell's system [57].

Agda's module system Agda's module system consists of two main features: modules and records. Records combine many different features into one. They are very similar to classes in object-oriented languages in that they have a constructor and fields with optionally some definitions. They also support instances and instance search, allowing records to behave like Haskell type classes [58].

Records can also be combined with modules to create a form of functors by specifying a module parameter of a specific record type. The module can then either directly define the resulting record or be used to instantiate a record. Alternatively, functors can be defined using the record instances similar to how type classes are used to encode functors in Haskell [54].

Exponential size problems encountered by module systems similar to that of Agda To deal with Agda's exponential growth problems we can look to other module systems to see if they encountered similar problems and if we can reuse their solutions. Two systems similarly expand the applications of a generic construct: generic functions in languages like Rust and functors in systems like ML. Of those two, Agda's module aliases are most similar to generics.

Both structs and functions can be generic over types in Rust [59]. For each instantiation of a generic construct, a copy is created during compilation that is specific for that instantiation. This results in a very fast run time, but slow compile time as well as larger binaries [6]. Rust also supports type aliases for structs that instantiate some or all of the arguments. While Agda's parameters and module aliases are of course much more powerful than generics, they are handled in the same way at the moment.

Agda's modules are quite different compared to functors as Agda's modules are not first-order, nor objects. Agda does not have to decide whether its modules should be generative or applicative as they are not objects [60]. Nevertheless, these systems also generate the modules created through functor applications to reduce the number of indirect calls, similar to Agda's current system [61, 62]. While doing so, the systems can also encounter the problem that they generate an exponential amount of code [61]. The exponential size is caused by code duplication in this case and can be solved through better code generation and clever use of let bindings.

Applicability of the solutions to Agda Agda's exponential size problem is not because of duplication, but because of nested module aliases creating an exponential amount of declaration names that need to be defined and it can thus not be solved in the same way.

The exponential expansion in languages such as Rust is not considered to be a problem at all and thus has no solution. Not everything will be a generic and thus not everything will have multiple copies. In contrast, almost every declaration in Agda can be aliased and thus require the creation of new declarations. The scale of the problem is thus larger.

Furthermore, the exponential growth with both functors and generics happens during compilation where some level of exponential growth is acceptable as long as it improves the runtime execution speed. In contrast, Agda's problems occur during type-checking which we do often without even executing the code. This means we want the type-checking to be fast while with compilation we care more about the speed of the resulting code. These reasons combined mean that we cannot use the solutions provided for these problems in Agda.

Problems that Agda shares with other module systems While Agda's exponential growth problem cannot easily be addressed using solutions from other systems, there are two areas of research that could be very relevant to future work on Agda's module system.

In many of the discussed languages, the module-level system is separate from the term-level system. ML is the closest to being unified with recent research proposing a redesign that would completely unify the term and module levels [63]. Future research on Agda might want to achieve a similar goal and try to unify Agda's modules and records into proper terms.

Other big topics for research are how to compile the various module systems efficiently [64, 65]. Once Agda has been refactored to no longer remove the module system during type-checking, the various compilers will have to solve the problem of compiling the modules effectively to their target language. At that point, it has to be evaluated whether compiler-techniques from Rust or ML or active research into compiling modules could be of benefit to those compilers.

Chapter 10

Conclusion

The main goal of this thesis was to improve the performance of Agda's module system. We have created a simpler language called Simple Agda to evaluate the performance of three different approaches: Agda's current approach, keeping modules and module parameters intact and keeping modules and module aliases both intact. We have introduced the concept of term-qualified names, which allows us to implement these later versions with ease, by realising that much of the difficulty of Agda's module system can be resolved by the scope-checker.

We have evaluated these approaches in a variety of scenarios using randomly generated files. These experiments showed that the best approach is to keep modules and module aliases intact during type-checking. This will perform better in all evaluated scenarios and completely eliminates the exponential complexity of Agda's current system when aliases are nested.

Furthermore, we have seen that this change will allow for several other problems to be addressed as well. The improved module system makes use of term-qualified names internally: `(M True).f`. Allowing this syntax to be used when programming in Agda will remove a significant number of pretty-printing problems. The improved performance of module aliases also means that open public statements can be changed to a more intuitive version. This was not yet possible due to the performance bottle-necks.

The performance benefits combined with the other benefits mean that making the proposed changes to Agda will massively improve the user experience as some long-standing problems are eliminated. The performance problems especially have hampered the development of, for example, category theory proofs as these benefit massively from module aliases, which so far, could not be used extensively.

However, the proposed changes are not yet the end of the improvements of Agda's module system. More research is still needed for both pretty-printing and records. With regards to pretty-printing it will still need to be decided as to how we want to qualify terms. Do we keep the alias qualifier when evaluating akin to a sort of dynamic dispatch or do we reduce it to the aliased term and start fully reducing terms? Now that we maintain aliases after type-checking, such questions can start to be analysed in much more detail.

Agda's record functionality has been extended multiple times in the past few years but this often occurred in an isolated manner. This means that it is unclear what the various interactions between the record features are. Furthermore, there is also a lack of consensus on how records should interact with the module system. Now that the module system has been cleared up more, it is time to do the same for the record system and see how it should interact with itself and with modules.

Bibliography

[1] X. Leroy *et al.* “Compcert c verified compiler.” (2022), [Online]. Available: https://compcert.org/compcert-C.html (visited on 04/10/2023).

[2] H. Barendregt and H. Geuvers, “Proof-assistants using dependent type systems,” in *Handbook of automated reasoning*, NLD: Elsevier Science Publishers B. V., Jan. 1, 2001, pp. 1149–1238, ISBN: 978-0-444-50812-6. (visited on 02/03/2023).

[3] Agda Community, *Agda*, version 2.6.2.2, Mar. 27, 2022. [Online]. Available: https://github.com/agda/agda/tree/v2.6.2.2.

[4] J. Carette and J. Hu, “Formalizing Category Theory in Agda,” 2021. DOI: 10.1145/3437992.3439922.

[5] E. Rijke, E. Bonnevier, J. Prieto-Cubides, F. Bakke, *et al.*, *Univalent mathematics in Agda*. [Online]. Available: https://github.com/UniMath/agda-unimath/.

[6] B. Anderson. “Generics and compile-time in rust,” PingCAP. (Jun. 15, 2020), [Online]. Available: https://www.pingcap.com/blog/generics-and-compile-time-in-rust/ (visited on 02/09/2023).

[7] M. Hofmann, “Syntax and Semantics of Dependent Types,” in *Semantics and Logics of Computation*, A. M. Pitts and P. Dybjer, Eds., Cambridge University Press, 1997, pp. 79–130. DOI: 10.1017/CBO9780511526619.004. [Online]. Available: https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/syntaxandsemanticsof-dependenttypes.pdf.

[8] N. G. de Bruijn, “Telescopic mappings in typed lambda calculus,” *Information and Computation*, vol. 91, no. 2, pp. 189–204, Apr. 1, 1991, ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90066-B. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089054019190066B (visited on 02/03/2023).

[9] “The λ-calculus,” in *Abstract Computing Machines: A Lambda Calculus Perspective*, ser. Texts in Theoretical Computer Science, W. Kluge, W. Brauer, G. Rozenberg, and A. Salomaa, Eds., Berlin, Heidelberg: Springer, 2005, pp. 51–88, ISBN: 978-3-540-27359-2. DOI: 10.1007/3-540-27359-X_4. [Online]. Available: https://doi.org/10.1007/3-540-27359-X_4 (visited on 02/03/2023).

[10] U. Norell, “Towards a practical programming language based on dependent type theory,” Ph.D. dissertation, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2007, 166 pp. [Online]. Available: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[11] Agda Language Reference. “Coverage checking.” (2023), [Online]. Available: https://agda.readthedocs.io/en/v2.6.3/language/coverage-checking.html (visited on 04/22/2023).

[12] Agda Language Reference. "Termination checking." (2023), [Online]. Available: `https://agda.readthedocs.io/en/v2.6.3/language/termination-checking.html` (visited on 04/22/2023).

[13] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism, Volume 149* (*Studies in Logic and the Foundations of Mathematics*). USA: Elsevier Science Inc., 2006, isbn: 978-0-444-52077-7.

[14] N. A. Danielsson. "Shadowing parameters are sometimes renamed." (2020), [Online]. Available: `https://github.com/agda/agda/issues/2018` (visited on 03/31/2023).

[15] Google Code Exporter. "Copatterns do not work in parametrized modules." (2015), [Online]. Available: `https://github.com/agda/agda/issues/940` (visited on 03/31/2023).

[16] W. Kahl. "Regression: Module parameters lost." (2015), [Online]. Available: `https://github.com/agda/agda/issues/1701` (visited on 03/31/2023).

[17] A. Abel. "Not a splittable variable." (2016), [Online]. Available: `https://github.com/agda/agda/issues/2181` (visited on 03/31/2023).

[18] O. Melkonian. "Unsafe(?) irrelevant projections by 'open'ing." (2023), [Online]. Available: `https://github.com/agda/agda/issues/6359` (visited on 03/31/2023).

[19] Google Code Exporter. "Undeclared name accepted in fixity declaration." (2015), [Online]. Available: `https://github.com/agda/agda/issues/329` (visited on 03/31/2023).

[20] A. Abel. "Printer prefers (longer) qualified over (shorter) unqualified name." (2019), [Online]. Available: `https://github.com/agda/agda/issues/3240` (visited on 03/31/2023).

[21] Google Code Exporter. "Printing of infix/mixfix operators defined in parametrized modules." (2022), [Online]. Available: `https://github.com/agda/agda/issues/632` (visited on 03/31/2023).

[22] L.-T. Chen. "Qualified names are printed if introduced by 'open M ...'." (2022), [Online]. Available: `https://github.com/agda/agda/issues/5632` (visited on 03/31/2023).

[23] A. Abel. "Exponential module chain leads to infeasible scope checking." (2022), [Online]. Available: `https://github.com/agda/agda/issues/1646` (visited on 03/31/2023).

[24] J. Carette. "Switch to a structured signature?" (2022), [Online]. Available: `https://github.com/agda/agda/issues/4331` (visited on 03/31/2023).

[25] A. Jonathan. "Unnecessary conversion checking due to parameterized module slows type-checking (a lot)." (2022), [Online]. Available: `https://github.com/agda/agda/issues/4517` (visited on 03/31/2023).

[26] J. Carette. "Agda-categories wiki: Speed." (2023), [Online]. Available: `https://github.com/agda/agda-categories/wiki/speed` (visited on 06/20/2023).

[27] P. G. Giarrusso. "Regression with open public." (2018), [Online]. Available: `https://github.com/agda/agda/issues/1985` (visited on 03/31/2023).

[28] N. A. Danielsson. "Record constructors sometimes in record modules, sometimes not." (2019), [Online]. Available: `https://github.com/agda/agda/issues/4189` (visited on 03/31/2023).

[29] Google Code Exporter. "Change the semantics of open public in parameterised module." (2018), [Online]. Available: `https://github.com/agda/agda/issues/892` (visited on 03/31/2023).

[30] L. Diehl. "Closed Anonymous Modules." (2022), [Online]. Available: `https://github.com/agda/agda/issues/2293` (visited on 03/31/2023).

[31] N. A. Danielsson. "Private where modules." (2023), [Online]. Available: `https://github.com/agda/agda/issues/2593` (visited on 03/31/2023).

[32] U. Norell. “Internal error for local modules with refined parameters.” (2018), [Online]. Available: https://github.com/agda/agda/issues/2897 (visited on 03/31/2023).

[33] M. Arntzenius. “record module functions should get hidden parameters when copied by module application.” (2022), [Online]. Available: https://github.com/agda/agda/issues/2675 (visited on 03/31/2023).

[34] U. Norell. “TERMINATING pragma ignored in record.” (2022), [Online]. Available: https://github.com/agda/agda/issues/3008 (visited on 03/31/2023).

[35] A. Abel. “In record declarations, meta variables are messed up.” (2019), [Online]. Available: https://github.com/agda/agda/issues/2561 (visited on 03/31/2023).

[36] S. Levy. “Instance fields without eta-equality.” (2022), [Online]. Available: https://github.com/agda/agda/issues/5071 (visited on 03/31/2023).

[37] A. Abel. “Records: separate field names from binders.” (2022), [Online]. Available: https://github.com/agda/agda/issues/5361 (visited on 03/31/2023).

[38] ul. “Doubts regarding inductive records with eta.” (2022), [Online]. Available: https://github.com/agda/agda/issues/5842 (visited on 03/31/2023).

[39] S. Weirich, *Pi-forall*, version 2022 version, Oct. 18, 2022. [Online]. Available: https://github.com/sweirich/pi-forall.

[40] T. Coquand, “An algorithm for type-checking dependent types,” *Science of Computer Programming*, vol. 26, no. 1, pp. 167–177, 1996, ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(95)00021-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167642395000216.

[41] G. Allais. “Case-split on a datatype defined in a parametrised module produces needlessly-prefixed patterns.” (2022), [Online]. Available: https://github.com/agda/agda/issues/3209 (visited on 03/31/2023).

[42] A. Abel. “inverseScopeLookup chooses suboptimal name.” (2018), [Online]. Available: https://github.com/agda/agda/issues/1643 (visited on 03/31/2023).

[43] J. Cockx, “Mini modules: A structured module system with parametrized modules and dependent types,” 2020.

[44] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, May 2011, ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/1988042.1988046.

[45] N. Smallbone, K. Claessen, O. Grenrus, and B. Bringert, *Quickcheck*, version 2.14.2, 2023. [Online]. Available: https://hackage.haskell.org/package/QuickCheck.

[46] M. Daggitt, N. A. Danielsson, and G. Allais, *Agda-stdlib*, version 1.7.2, 2023. [Online]. Available: https://github.com/agda/agda-stdlibs.

[47] Haskell, *Deepseq*, version 1.4.8.1, Dec. 18, 2022. [Online]. Available: https://hackage.haskell.org/package/deepseq.

[48] F. Domínguez, *Timestats*, version 0.1.0, Jul. 13, 2022. [Online]. Available: https://hackage.haskell.org/package/timestats-0.1.0.

[49] J. Carette. “Feature request: aliases (or ’light’ modules).” (2022), [Online]. Available: https://github.com/agda/agda/issues/5499 (visited on 04/24/2023).

[50] anuyts. “Extending existing modules.” (2022), [Online]. Available: https://github.com/agda/agda/issues/6038 (visited on 04/06/2023).

[51] D. MacQueen, "Modules for standard ml," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84, Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 198–207, isbn: 0897911423. doi: 10.1145/800055.802036. [Online]. Available: https://doi.org/10.1145/800055.802036.

[52] D. MacQueen, R. Harper, and J. Reppy, "The history of standard ml," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, Jun. 2020. doi: 10.1145/3386336. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3386336.

[53] X. Leroy, "A modular module system," *Journal of Functional Programming*, vol. 10, no. 3, pp. 269–303, May 2000, Publisher: Cambridge University Press, issn: 1469-7653, 0956-7968. doi: 10.1017/S0956796800003683. [Online]. Available: http://www.cambridge.org/core/journals/journal-of-functional-programming/article/modular-module-system/A8D022C76CBFB0DD9EEA05458D5C662D# (visited on 03/10/2021).

[54] S. Wehr and M. M. T. Chakravarty, "Ml modules and haskell type classes: A constructive comparison," in *Programming Languages and Systems*, G. Ramalingam, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 188–204, isbn: 978-3-540-89330-1.

[55] A. Rossberg and D. Dreyer, "Mixin' up the ML module system," *ACM Transactions on Programming Languages and Systems*, vol. 35, no. 1, 2:1–2:84, Apr. 1, 2013, issn: 0164-0925. doi: 10.1145/2450136.2450137. [Online]. Available: http://doi.org/10.1145/2450136.2450137 (visited on 11/18/2022).

[56] T. Hirschowitz and X. Leroy, "Mixin modules in a call-by-value setting," in *Programming Languages and Systems*, D. Le Métayer, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 6–20, isbn: 978-3-540-45927-9.

[57] S. Kilpatrick, D. Dreyer, S. Peyton Jones, and S. Marlow, "Backpack: Retrofitting haskell with interfaces," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego California USA: ACM, Jan. 8, 2014, pp. 19–31, isbn: 978-1-4503-2544-8. doi: 10.1145/2535838.2535884. [Online]. Available: https://dl.acm.org/doi/10.1145/2535838.2535884 (visited on 02/24/2021).

[58] Agda Language Reference. "Record types." (2023), [Online]. Available: https://agda.readthedocs.io/en/latest/language/record-types.html (visited on 05/04/2023).

[59] S. Klabnik and C. Nichols, *The Rust Programming Language*, 1st Edition. San Francisco: No Starch Press, Jun. 26, 2018, 552 pp., isbn: 978-1-59327-828-1.

[60] Y. Sato and Y. Kameyama, "Type-safe generation of modules in applicative and generative styles," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, Chicago IL USA: ACM, Oct. 17, 2021, pp. 184–196, isbn: 978-1-4503-9112-2. doi: 10.1145/3486609.3487209. [Online]. Available: https://dl.acm.org/doi/10.1145/3486609.3487209 (visited on 02/07/2023).

[61] Y. Sato, Y. Kameyama, and T. Watanabe, "Module generation without regret," in *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM 2020, New York, NY, USA: Association for Computing Machinery, Jan. 20, 2020, pp. 1–13, isbn: 978-1-4503-7096-7. doi: 10.1145/3372884.3373160. [Online]. Available: https://doi.org/10.1145/3372884.3373160 (visited on 02/07/2023).

[62] Y. Sato and Y. Kameyama, "Type-safe generation of modules in applicative and generative styles," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, Chicago IL USA: ACM, Oct. 17, 2021, pp. 184–196, isbn: 978-1-4503-9112-2. doi: 10.1145/3486609.3487209. [Online]. Available: https://dl.acm.org/doi/10.1145/3486609.3487209 (visited on 02/07/2023).

[63] A. ROSSBERG, "1ml – core and modules united," *Journal of Functional Programming*, vol. 28, e22, 2018. doi: 10.1017/S0956796818000205.

[64] G. Kuan, “True higher-order module systems, separate compilation, and signature calculi,” Ph.D. dissertation, University of Chicago, Chicago IL USA, Jun. 2010. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=07258e13737477d9ca682db338325ea98189bdc9.

[65] K. Crary, “Fully abstract module compilation,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290323. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3290323.

Acronyms

WHNF	weak head normal form
AST	Abstract syntax tree

Appendix A

Full typing judgements for version 1

This appendix will provide a fully updated set of judgements for version 1, including all the judgements that were left out in section 5.1 because they are the same as in version 0. `extend` uses the fully qualified names in declarations to insert them in the proper places in the signature.

$$\frac{\textbf{module}\ M\ \Delta = \Sigma' \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta', \Sigma'')}$$

$$\frac{}{\Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure A.1: Signature lookup for version 1

$$\frac{\alpha.f = u \in \Sigma}{\Sigma \vdash a.f \xrightarrow{WHNF} u} \qquad \frac{}{\Sigma \vdash (\lambda x\ .\ u)\ v \xrightarrow{WHNF} u[x := v]} \qquad \frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash u\ v \xrightarrow{WHNF} u'\ v}$$

$$\frac{}{\Sigma \vdash \textbf{if True then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} v} \qquad \frac{}{\Sigma \vdash \textbf{if False then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash \textbf{if}\ u\ \textbf{then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} \textbf{if}\ u'\ \textbf{then}\ v\ \textbf{else}\ w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} v \qquad \Sigma \vdash v \xrightarrow{WHNF} w}{\Sigma \vdash u \xrightarrow{WHNF} w} \qquad \frac{}{\Sigma \vdash u \xrightarrow{WHNF} u}$$

Figure A.2: WHNF evaluation of version 1

$$\frac{}{\Sigma; \Gamma \vdash \epsilon} \qquad \frac{\Sigma; \Gamma \vdash A : \textbf{Type} \qquad \Sigma; \Gamma, x : A \vdash \Delta}{\Sigma; \Gamma \vdash (x : A), \Delta}$$

Figure A.3: Telescope typing rules for version 1

$$\frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x : A} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{Type} : \textbf{Type}} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{Bool} : \textbf{Type}} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{Unit} : \textbf{Type}}$$

$$\frac{}{\Sigma; \Gamma \vdash \textbf{True} : \textbf{Bool}} \qquad \frac{}{\Sigma; \Gamma \vdash \textbf{False} : \textbf{Bool}} \qquad \frac{}{\Sigma; \Gamma \vdash \mathtt{1} : \textbf{Unit}}$$

$$\frac{\Sigma; \Gamma \vdash u : \textbf{Bool} \qquad \Sigma; \Gamma \vdash v : A \qquad \Sigma; \Gamma \vdash w : A}{\Sigma; \Gamma \vdash \textbf{if } u \textbf{ then } v \textbf{ else } w \ : A}$$

$$\frac{\Sigma; \Gamma \vdash A : \textbf{Type} \qquad \Sigma; \Gamma, x : A \vdash B : \textbf{Type}}{\Sigma; \Gamma \vdash (x : A) \to B : \textbf{Type}} \qquad \frac{\Sigma; \Gamma, x : A \vdash u : B}{\Sigma; \Gamma \vdash \lambda x \,.\, u : A \to B}$$

$$\frac{\Sigma; \Gamma \vdash u : A \qquad \Sigma; \Gamma \vdash A \xrightarrow{WHNF} (b : B) \to C \qquad \Sigma; \Gamma \vdash v : B}{\Sigma; \Gamma \vdash u\ v : C[b := v]}$$

$$\frac{\Sigma!\alpha \rightsquigarrow (\Delta, \Sigma') \qquad f : A \in \Sigma'}{\Sigma; \Gamma \vdash \alpha.f : \Delta \to A}$$

Figure A.4: Term typing judgements for version 1

$$\frac{\Sigma; \Gamma \vdash A : \textbf{Type} \qquad \Sigma; \Gamma \vdash u : A}{\Sigma; \Gamma \vdash \alpha.f : A = u \rightsquigarrow \alpha.f : A = u}$$

$$\frac{\Sigma; \Gamma \vdash \Delta \qquad \Sigma; \Gamma M(\Delta) \vdash \mathit{decls} \rightsquigarrow \mathit{decls}'}{\Sigma; \Gamma \vdash \textbf{module } M\ \Delta \textbf{ where } \mathit{decls} \rightsquigarrow \textbf{module } M\ \Delta \textbf{ where } \mathit{decls}'}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \Delta \qquad \Sigma!\alpha \rightsquigarrow (\Theta, \Sigma') \qquad \textbf{module } M'\ \Theta' \textbf{ where } \mathit{decls} \in \Sigma' \\ \Sigma; \Gamma M(\Delta) \vdash \bar{u} : \Theta\Theta' \qquad \text{let } \mathit{decls}' = \mathtt{newDecls}(M, \alpha.M', \bar{u}, \mathit{decls}[\Theta\Theta' := \bar{u}])\end{array}}{\Sigma; \Gamma \vdash \textbf{module } M\ \Delta = \alpha.M'\ \bar{u} \rightsquigarrow \textbf{module } M\ \Delta \textbf{ where } \mathit{decls}'}$$

$$\frac{}{\Sigma; \Gamma \vdash [] \rightsquigarrow \epsilon} \qquad \frac{\begin{array}{c}\Sigma; \Gamma \vdash \mathit{decl} \rightsquigarrow \mathit{decl}' \qquad \text{let } \Sigma' = \mathtt{extend}(\Sigma, \mathit{decl}') \\ \Sigma'; \Gamma \vdash \mathit{decls} \rightsquigarrow \Sigma''\end{array}}{\Sigma; \Gamma \vdash \mathit{decl} :: \mathit{decls} \rightsquigarrow \Sigma''}$$

Figure A.5: Declaration typing judgements for version 1

$\mathtt{newDecls}(M, M', \bar{u}, D) = \mathtt{newDecl}(M, M', \bar{u}, d)$ for each $d \in D$

$\mathtt{newDecl}(M, M', \bar{u}, M'.f : A = u) = M.f = M'.f\ \bar{u}$

$\mathtt{newDecl}(M, M', \bar{u}, \textbf{module } M'.N\ \Delta \textbf{ where } \mathit{decls}) =$

$\qquad \textbf{module } M.N\ \Delta' \textbf{ where } \mathtt{newDecls}(M.N, M'.(N\ \Delta), \bar{u}, \mathit{decls})$

Figure A.6: `newDecls` definition for version 1

Appendix B

Full typing judgements for version 2

This appendix will provide a fully updated set of judgements for version 2, including all the judgements that were left out in section 5.2 because they are the same as in version 1. Because no changes are made to the program, Σ is initialised to the original program and never modified.

$$\frac{\textbf{module}\ M\ \Delta = \Sigma' \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma^R \vdash \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta', \Sigma'')}$$

$$\frac{\begin{array}{c}\textbf{module}\ M\ \Delta = M'\ \bar{u} \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma^R \vdash \Sigma^R!M' \rightsquigarrow (\Delta', \Sigma') \\ \Delta'_1, \Delta'_2 = \texttt{split}(\Delta', \bar{u}) \qquad \Sigma^R \vdash \Sigma'[\Delta'_1 := \bar{u}[\Delta_1 := \bar{a}]]!\alpha \rightsquigarrow (\Delta'', \Sigma'')\end{array}}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta'_2\Delta'', \Sigma'')}$$

$$\frac{}{\Sigma^R \vdash \Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure B.1: Signature lookup for version 2

$$\frac{\alpha.f = u \in \Sigma}{\Sigma \vdash a.f \xrightarrow{WHNF} u} \qquad \frac{}{\Sigma \vdash (\lambda x\ .\ u)\ v \xrightarrow{WHNF} u[x := v]} \qquad \frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash u\ v \xrightarrow{WHNF} u'\ v}$$

$$\frac{}{\Sigma \vdash \textbf{if True then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} v} \qquad \frac{}{\Sigma \vdash \textbf{if False then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} u'}{\Sigma \vdash \textbf{if}\ u\ \textbf{then}\ v\ \textbf{else}\ w \xrightarrow{WHNF} \textbf{if}\ u'\ \textbf{then}\ v\ \textbf{else}\ w}$$

$$\frac{\Sigma \vdash u \xrightarrow{WHNF} v \qquad \Sigma \vdash v \xrightarrow{WHNF} w}{\Sigma \vdash u \xrightarrow{WHNF} w} \qquad \frac{}{\Sigma \vdash u \xrightarrow{WHNF} u}$$

Figure B.2: WHNF evaluation of version 2

$$\frac{}{\Sigma;\Gamma \vdash \epsilon} \qquad \frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma, x : A \vdash \Delta}{\Sigma;\Gamma \vdash (x : A), \Delta}$$

Figure B.3: Telescope typing rules for version 2

$$\frac{x : A \in \Gamma}{\Sigma;\Gamma \vdash x : A} \qquad \frac{}{\Sigma;\Gamma \vdash \textbf{Type} : \textbf{Type}} \qquad \frac{}{\Sigma;\Gamma \vdash \textbf{Bool} : \textbf{Type}} \qquad \frac{}{\Sigma;\Gamma \vdash \textbf{Unit} : \textbf{Type}}$$

$$\frac{}{\Sigma;\Gamma \vdash \textbf{True} : \textbf{Bool}} \qquad \frac{}{\Sigma;\Gamma \vdash \textbf{False} : \textbf{Bool}} \qquad \frac{}{\Sigma;\Gamma \vdash \mathtt{1} : \textbf{Unit}}$$

$$\frac{\Sigma;\Gamma \vdash u : \textbf{Bool} \qquad \Sigma;\Gamma \vdash v : A \qquad \Sigma;\Gamma \vdash w : A}{\Sigma;\Gamma \vdash \textbf{if } u \textbf{ then } v \textbf{ else } w \ : A}$$

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma, x : A \vdash B : \textbf{Type}}{\Sigma;\Gamma \vdash (x : A) \to B : \textbf{Type}} \qquad \frac{\Sigma;\Gamma, x : A \vdash u : B}{\Sigma;\Gamma \vdash \lambda x \,.\, u : A \to B}$$

$$\frac{\Sigma;\Gamma \vdash u : A \qquad \Sigma;\Gamma \vdash A \xrightarrow{WHNF} (b : B) \to C \qquad \Sigma;\Gamma \vdash v : B}{\Sigma;\Gamma \vdash u\ v : C[b := v]}$$

$$\frac{\Sigma!\alpha \rightsquigarrow (\Delta, \Sigma') \qquad f : A \in \Sigma'}{\Sigma;\Gamma \vdash \alpha.f : \Delta \to A}$$

Figure B.4: Term typing judgements for version 2

$$\frac{\Sigma;\Gamma \vdash A : \textbf{Type} \qquad \Sigma;\Gamma \vdash u : A}{\Sigma;\Gamma \vdash \alpha.f : A = u} \qquad \frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma;\Gamma M(\Delta) \vdash decls}{\Sigma;\Gamma \vdash \textbf{module } M\ \Delta \textbf{ where } decls}$$

$$\frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma \vdash \Sigma!\alpha \rightsquigarrow (\Theta, \Sigma') \qquad \textbf{module } M'\ \Theta' \in \Sigma' \qquad \Sigma;\Gamma M(\Delta) \vdash \bar{u} : \Theta\Theta'}{\Sigma;\Gamma \vdash \textbf{module } M\ \Delta = \alpha.M'\ \bar{u}}$$

$$\frac{}{\Sigma;\Gamma \vdash []} \qquad \frac{\Sigma;\Gamma \vdash decl \qquad \Sigma;\Gamma \vdash decls}{\Sigma;\Gamma \vdash decl :: decls}$$

Figure B.5: Declaration typing judgements for version 2

Appendix C

Paper for IFL

This thesis was made into a paper for IFL. This paper will be finished after the submission deadline for the master thesis, which means that the version below is still a draft.

Improving Agda's module system*

Ivar de Bruin

ABSTRACT

Agda is a language used to write computer-verified proofs. It has a module system that provides namespacing, module parameters and module aliases. These parameters and aliases can be used to write shorter and cleaner proofs. However, the current implementation of the module system has several problems, such as an exponential desugaring of module aliases. This paper shows how the module system can be changed to address these problems. We have found that we do not need any desugarings during type-checking, but can instead handle module parameters and aliases during signature lookup by making a small change to the scope-checker, completely eliminating any exponential growth problems and unnecessary complexity. This will allow users to make more effective use of the module system, simplifying their proofs. Furthermore, the improvements to the module system will allow future research to fix the problems with Agda's implementation of pretty-printing, records and open public statements.

ACM Reference Format:
Ivar de Bruin. 2023. Improving Agda's module system. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Throughout programming history, programmers have struggled with bugs. This is especially a problem in security-related programs where bugs can have big consequences. One possible solution to this problem has been to create code that is proven to be correct. This has already seen some use in, for example, the creation of a C compiler [29].

To create such proofs, programmers work in proof-assistants [7]. Many of these proof-assistants are programming languages with strong type systems and additional properties such as guaranteed termination to allow for the creation of proofs. This paper focuses on Agda [4], a proof assistant that uses a syntax similar to Haskell, allowing users to write programs in Agda, prove them correct, then transform them to Haskell code using a Haskell back-end, thereby maintaining the guarantee that their code has the proven properties, while still using a programming language optimised for performance.

Agda is also used to prove various mathematical properties or even entire mathematical fields [9, 34]. To help programmers structure larger proofs, Agda provides a module system. Modules allow for easy namespacing and grouping of proofs. In addition, there are two interesting features that can be of great help when writing proofs: module parameters and module aliases.

Module parameters are in scope for all declarations in that module while module aliases allow programmers to create aliases that instantiate these parameters with specific arguments. This allows programmers to make proofs generic over some module parameters. Then, using a module alias, these proofs can be instantiated for a specific value as can be seen here, where `CategoryProofs` contains a variety of proofs generalised over some `Category c`:

```
postulate d e : Category
module CategoryProofs (c : Category) where
...
module DProofs = CategoryProofs d
module EProofs = CategoryProofs e
```

Structuring proofs this way greatly increases both the usability as well as the readability and ease of writing of the proofs. Unfortunately, the current version of Agda does not perform well when many aliases are used due to exponential growth problems. This occurs because Agda replaces the alias with specialised functions for each declaration being aliased, similar to what many compilers do when dealing with parametrised functions [6]. While this technique makes sense for a compiler, it is not necessarily a good idea for a proof-assistant, as we do not care for the speed of the type-checked code, but about the speed of the type-checker.

Furthermore, removing aliases makes the implementation of the type-checker more complex and bug-prone due to the increase in transformations needed and finally, any back-ends that are implemented for Agda will not have access to the aliases as the type-checker already removed them, limiting the potential compilations.

This paper will analyse different approaches to Agda's module system to remove this performance bottleneck while preserving the module features during type-checking. Concretely, we make the following contributions:

- We analyse the issues on the Agda GitHub issue tracker to identify the main problems with Agda's module system: The lack of structure, the performance problems with nested module aliases and a variety of issues related to pretty-printing (Section 3).
- We introduce the concept of term-qualified names and explain how we can use these to change Agda's approach to type-checking modules in a way that no longer requires us to change modules and module aliases to declarations, simplifying the implementation grealy (Section 4).
- We analyse the different type-checkers in a variety of scenarios, using randomly generated files, which shows that keeping aliases intact has far superior performance, with barely any downsides. We also verify the accuracy of our

*This is a shortened version of my Master thesis with the same title [16]. All code and results used in this paper can be found at: https://github.com/ivardb/AgdaModuleImprovement

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

experiments by running them on Agda to make sure that our baseline implementation matches Agda's performance (Section 5).
- We analyse the results as well as a number of other factors, such as the difficulty of refactoring Agda, to explain that Agda should switch its implementation to an approach that preserves module aliases (Section 6).

2 BACKGROUND

This section will cover the required background information on Agda. It is assumed that readers are already familiar with dependent typing [23] and its associated concepts such as weak head normal form evaluation [27] and telescopes [15]. A full description of Agda's module system can be found in Norell's thesis [33].

Agda's module system. The basis of Agda's module system are modules which can be nested. In addition to standard namespacing, modules can also have parameters:

```
module M (x : Bool) where
  f = x
  g = f
z : Bool
z = M.g True
```

The parameters are in scope for all declarations inside the module and when a declaration is used outside of its module, the module parameters have to be supplied just like normal function parameters.

When you often make use of the same module with the same module arguments, it can be useful to make use of a module alias instead:

```
module MBool = M True
z : Bool
z = MBool.g
```

Module aliases allow you to create an alternative module that supplies zero or more of the parameters of a module. Module aliases and parameters can be extremely useful when writing a set of proofs that depend on some category or some number etc.

Interface files. Before we cover how Agda's current implementation type-checks these module parameters and aliases, we need to quickly cover Agda's interface files. During type-checking Agda will simplify the program down to a core language that can then be stored in its interface files. These files are then used whenever a file needs to be imported to prevent it from having to be type-checked every time it is used.

Such a core language output consists of two parts. First, there are the sections, these are all the modules that exist together with their telescope. This information is needed to type-check module aliases. After that, we will have the declarations together with their given or inferred type signatures. The declarations are all lifted to the top level as that is how they would be used when imported. For example:

```
section M (x : Bool)

M.f : Bool -> Bool
M.f = \x . x
M.g : Bool -> Bool
M.g = \x . M.f x
```

Type-checking and elaboration of modules. Before type-checking, the scope-checker will insert fully-qualified names and deal with any visibility modifiers. This paper will ignore the visibility modifiers and assume that the scope-checker somehow inserts these fully-qualified names.

The current implementation of Agda lifts all declarations to the top level. This will change the types of function calls as the type of a qualified name depends on the module it is in as a module parameter that is shared by both the caller and the target declaration is not part of the type of the function call. When declarations are lifted to the top level, they lose these shared module parameters as they become function parameters.

In this example, the function `f` does not take any parameters as both `f` and `g` have `b` in scope.

```
module M (b : Bool) where
  f : Bool
  f = b

  g : Bool
  g = f
```

When we remove modules to get:

```
M.f = \b . b
M.g = \b a . M.f b
```

`M.f` needs to be explicitly passed the `b` argument as it is no longer automatically in scope.

To type-check a module alias, new definitions are introduced. For each definition in the original, we need a definition in the new module that redirects to the old definition, passing the appropriate arguments.

```
module M (X : Type) where
  M.id : X → X
  M.id x = x
module MBool = M Bool
```

will create

```
M.id : (X : Type) → X → X
M.id X x = x

MBool.id : Bool -> Bool
MBool.id = M.id Bool
```

This means that any arguments passed to an aliased module will be copied once for each declaration.

There are several problems with this way of type-checking and section 3 will explain what these problems are in more detail.

3 PROBLEM DESCRIPTION

Agda's module system has several problems, many of which are reported as issues on GitHub[1]. Before we go through the problems

[1] https://github.com/agda/agda/issues?q=label%3Amodules+sort%3Aupdated-desc+

found in these issues there is a more general problem. The Adga type-checker already performs a partial compilation.

Agda allows for the implementation of custom back-ends which can be used to compile Agda to different languages. However, by then Agda will already have removed its modules and moved the module parameters to the declaration level. This means that the back-end is not able to decide to do so for itself. If you want to make a back-end for Agda that applies a transformation and then returns valid Agda code, you will not be able to do so while preserving the original module structure.

This is a conceptual problem that harms Agda's usefulness but it is not relevant when working in Agda itself. Looking at the Agda issue tracker on GitHub we can see that there are three major groups of problems that affect Agda programmers:

(1) **Lack of module structure in the typing environment:** Agda has to make changes to the declarations to lift them to the top level and to move the module parameters to the declaration level. These changes can easily introduce bugs due to renaming to preserve unique names [14] or because it becomes harder to know if a parameter is part of the original declaration or if it came from a module [20, 25, 2]. While these bugs could all be easily fixed, some issues are still not fixed. One such issue is issue #6359 which is a problem that is caused by declarations being added to the global scope while they should only exist locally, thereby breaking the tracking of specific data resulting in buggy behaviour [32]. These issues show that the current approach of Agda is both complex and restrictive.

(2) **Performance problems:** The second and most serious of the problems is that of performance [1, 8, 24]. These problems are either caused by very large numbers of module parameters or more often by nested module aliases as this produces an exponential growth in the number of new declarations being created. Finally, many users are of the opinion that the current behaviour of the open public statement is unintuitive [18, 13]. However, this cannot be fixed at the moment as the current implementation is merely a scope-checking trick while fixing the behaviour would require the use of module aliases which would be too big of a performance sacrifice [19].

(3) **Pretty-Printing problems:** The final set of problems is related to pretty-printing. Agda has a variety of problems with pretty-printing values from different modules. Either because it cannot remember where a definition originated [22, 3] or because it loses track of module parameters making infix operators especially confusing to read [21, 10].

4 IMPROVEMENTS TO THE MODULE SYSTEM

This section will explain our main contribution of an updated module system. Section 4.1 will first explain the simplified version of Agda, used to implement these updated systems. Section 4.2 will then explain the core idea behind our changes. After this, section 4.3 will explain how we use this concept to preserve both modules and module aliases, after which section 4.4 will explain the different versions of the type-checker used for the experiments.

4.1 Simple Agda

Making changes to the actual Agda code base is not a quick process. To be able to prototype a number of different approaches we have implemented a simplified version of Agda, called Simple Agda, which supports pi-types, lambdas, primitive Unit and Bool values as well as if expressions to eliminate booleans and it supports module parameters and module aliases.

The language has no scoping features such as visibility modifiers nor features such as universe levels and inductive data types as these do not impact the module system, only the complexity of the implementation.

A full specification of Simple Agda can be found in the full thesis [16]. The remainder of this section will use Simple Agda to explain a number of changes required for dealing with the previously mentioned problems.

4.2 Term-qualified names

Lets take a look at how Agda handles module parameters. Take the following example:

```
module M (x : Bool) where
  module M1 (y : Bool) where
    f : Bool
    f = x
  module M2 (z : Bool) where
    g : Bool
    g = M1.f True
```

Which gets transformed to:

```
M.M1.f : Bool -> Bool -> Bool
M.M1.f = \x y . x
M.M2.g : Bool -> Bool -> Bool
M.M2.g = \x z . M.M1.f x True
```

We can see from this example that two things changed in the function call to `M1.f`. First, the scope-checker inserted the qualifier `M` to create a fully-qualified name, and then the type-checker inserted the parameter `x` to make sure the program is still correct after being lifted to the top level.

It is important to realise that the inserted parameters are always exactly the module parameters belonging to the modules whose names are added to create fully-qualified names. We have three module parameters in our example. `z` clearly does not need to be inserted into the function call as it is only in scope for `g`. `y` does not need to be inserted into the function call as the programmer already passed a value for it in the form of `True`. Finally, `x` does need to be inserted into the function call as the programmer does not have to provide it, while it is in scope for both `f` and `g`, meaning it needs to be passed through the function call to make sure that both `f` and `g` use the same parameter. This exact reasoning holds for which module names the scope-checker needs to insert to create fully-qualified names.

if we switch to a system preserving modules like so:

```
module M (x : Bool) where
  module M.M1 (y : Bool) where
    M.M1.f : Bool
    M.M1.f = x
  module M2 (z : Bool) where
    M.M2.g : Bool
    M.M2.g = M.M1.f True
```

we no longer need to insert these parameters for the generated code to make sense, as we are no longer performing any liftings. However, this code is still problematic. If we evaluate the term:

```
h = if M.M2.g False False then True else False
```

we get:

```
h = if M.M1.f True then True else False
h = if (\y . True) then True else False
```

This happens as we needed to lift the function call to `M1.f` out of the module `M2` as part of substituting the call to `M.M2.g`. This caused the `x` parameter and its value to completely disappear as they do not occur in the definition of g.

Instead we will use term-qualified names to preserve these parameters by transforming the example to:

```
module M (x : Bool) where
  module M.M1 (y : Bool) where
    M.M1.f : Bool
    M.M1.f = x
  module M2 (z : Bool) where
    M.M2.g : Bool
    M.M2.g = (M x).M1.f True
```

This transformation can be done entirely by the scope-checker with no additional work compared to only inserting qualified names. The type-checker now no longer needs to deal with the moving around and insertion of module parameters.

Evaluating the previous term will now correctly evaluate to:

```
h = if (M False).M1.f True then True else False
h = if False then True else False
h = False
```

4.3 Signature lookup

Using term-qualified names we can switch to a type-checking approach that preserves modules and their parameters and switch to a structured signature. To find the declaration belong to a specific module inside this signature we have to perform signature lookup.

We will use the syntax: $\Sigma^R \vdash \Sigma!(Mx).M2 \rightsquigarrow (\Delta, \Sigma')$ to say that we are looking for the signature belonging to $(Mx).M2$ relative to Σ and that this results in the signature Σ' with Δ representing the telescope of module parameters surrounding this signature. Σ^R then represents the root signature which we need when dealing with module aliases.

The definition of signature lookup can be found in figure 1 consisting of a base case and two relevant cases. First, we define lookup in the case where we encounter a normal module. In that case, we can split the module telescope into two (potentially empty) parts. A part of the telescope to which we provided arguments in the qualified name, and a part which still needs an argument. If we use our example of $(Mx).M2$ we can see that when looking up M we would get $(x : Bool), ()$ while when looking up $M2$ we would get $(), (y : Bool)$. Next, we look up the remaining qualifier in the signature of our newly found module and add the unmatched parameters to the returned telescope. Note that the substitution on the signature would be implemented lazily in an actual implementation as there is no need to perform it on the whole signature.

The second case of signature lookup occurs when we encounter a module alias during lookup. In that case, we perform all of the same steps as the previous case, except we now first need to find the signature belonging to the module being aliased. This lookup is relative to the root signature instead of relative to the current signature as we use fully-qualified names which are relative to the root. Note that we also have to split the telescope of the module being aliased, as it is not required to provide an argument for each of its parameters in an alias.

$$\frac{\textbf{module}\ M\ \Delta = \Sigma' \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \\ \Sigma^R \vdash \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2 \longrightarrow \Delta', \Sigma'')}$$

$$\frac{\textbf{module}\ M\ \Delta = M'\ \bar{u} \in \Sigma \qquad \Delta_1, \Delta_2 = \texttt{split}(\Delta, \bar{a}) \\ \Sigma^R \vdash \Sigma^R!M' \rightsquigarrow (\Delta', \Sigma') \qquad \Delta'_1, \Delta'_2 = \texttt{split}(\Delta', \bar{u}) \\ \Sigma^R \vdash \Sigma'[\Delta'_1 := \bar{u}[\Delta_1 := \bar{a}]]!\alpha \rightsquigarrow (\Delta'', \Sigma'')}{\Sigma^R \vdash \Sigma!(M\bar{a}).\alpha \rightsquigarrow (\Delta_2 \longrightarrow \Delta'_2 \longrightarrow \Delta'', \Sigma'')}$$

$$\frac{}{\Sigma^R \vdash \Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure 1: Signature lookup

Now that we have defined signature lookup, we can define the typing judgements for qualified names inside terms and for checking modules and module aliases. Typing qualified names is very easy with this new system as we barely have to do any parameter manipulations. We simply find the signature belonging to the module qualifier and the type of our definition inside this signature. We then have to wrap this type with the telescope of the modules surrounding the signature and we will have the correct type. Due to the parameters inserted by the scope-checker, we do not have to worry about the parameters surrounding both the function call and the target declarations. Agda's current implementation instead

$$\frac{\Sigma!\alpha \rightsquigarrow (\Delta, \Sigma') \qquad f : A \in \Sigma'}{\Sigma; \Gamma \vdash \alpha.f : \Delta \longrightarrow A}$$

$$\frac{\Sigma; \Gamma \vdash \Delta \qquad \Sigma; \Gamma M(\Delta) \vdash \mathit{decls}}{\Sigma; \Gamma \vdash \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls}}$$

$$\frac{\Sigma; \Gamma \vdash \Delta \qquad \Sigma \vdash \Sigma!\alpha \rightsquigarrow (\Theta, \Sigma') \\ \textbf{module}\ M'\ \Theta' \in \Sigma' \qquad \Sigma; \Gamma M(\Delta) \vdash \bar{u} : \Theta\Theta'}{\Sigma; \Gamma \vdash \textbf{module}\ M\ \Delta = \alpha.M'\ \bar{u}}$$

Figure 2: Typing judgements

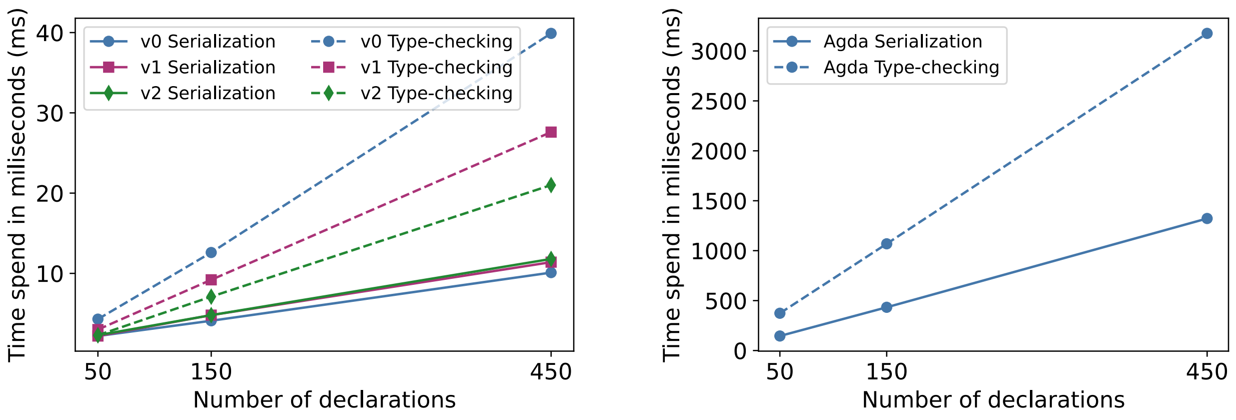

Figure 3: Experiment 1: Only declarations

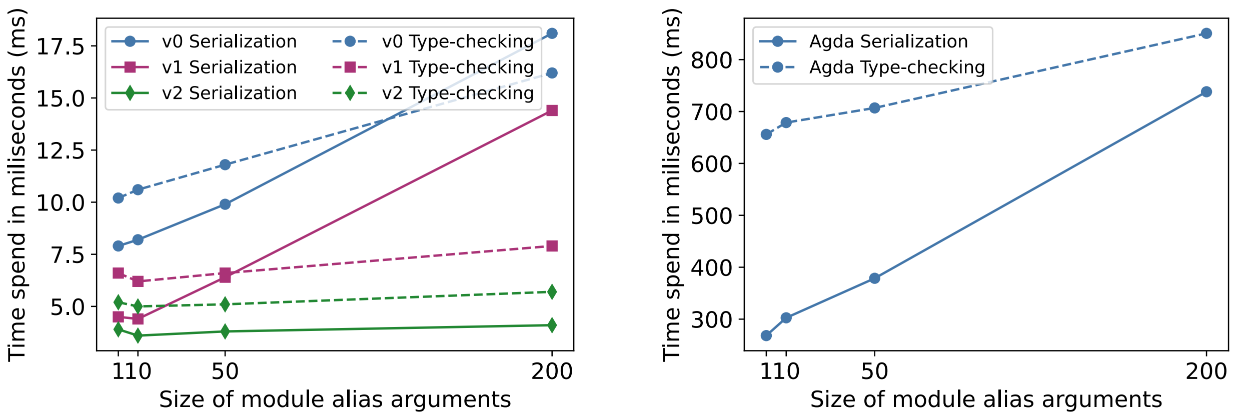

Figure 4: Experiment 2: Module argument size

has to deal with inserting the appropriate parameters during type-checking, finding these parameters as well as undoing any previous lifting of declarations [33].

Typing modules and module aliases is also quite easy as we do not have to worry about creating or modifying any declarations. For modules, we simply check that the telescope is well-formed and that the declarations are valid. For module aliases, we have to look up the telescope belonging to the aliased module to check that the provided arguments are valid. This is much easier than the current implementation of Agda that has to worry about lifting the declarations to the top level and generating the correct declarations when dealing with aliases [33].

4.4 The implemented type-checkers

A total of three different approaches to type-checking the module system have been implemented. The first type-checker uses the approach Agda takes and lifts all declarations to the top level at the same time, removing all module features. This version will serve as the baseline of our experiments and is referred to as version 0.

The next implementation, version 1, makes use of term-qualified names and uses a structured signature, preserving modules and module parameters. Module aliases will create new declarations, similar to version 0, except these declarations will now be a part of modules with module parameters.

The final version, version 2, will make use of term-qualified names and a structured signature supporting module aliases. This version therefore no longer has to make any changes during type-checking, completely preserving the original source syntax.

While the above versions all take different approaches and have to be evaluated for performance in the experiments, there is one other version implemented, version 3. This version does not change anything to the type-checker at all. Instead, it simply enables term-qualified syntax for programmers. This does not require further changes, but does allow us to fix the pretty-printing issues with infix and mixfix operators [21]. An analysis of how other pretty-printing issues could be addressed by the module system in future work is discussed in section 5.3 of the full thesis [16].

5 EXPERIMENT RESULTS

This section will cover the experiments used to measure the performance differences between the versions. The experiments were executed for our implementations as well as on Agda itself to verify the accuracy of our baseline. We will only cover four of our experiments in this paper, more can be found in chapter 7 of the full thesis [16].

5.1 Experiment setup

For the experiments, it is important to isolate specific features of the language, such as for example, the size of the arguments provided to a module alias. This is very hard to do with real-world code so instead we created a generator that can generate files according to some parameters, as explained in chapter 6 of the full thesis [16]. Using this generator, we have generated 15 files for each of our experiment configurations. These files were then converted from Simple Agda to Agda to allow them to be run by Agda as well.

The Simple Agda experiments were timed using the timestats library for Haskell [17], while the Agda experiments made use of the Agda-2.6.2.2 executable and its built-in benchmarking features [4].

The experiments were all executed on a 6-core Intel i7-8750H running at 2.20 GHz with 16 GB of RAM and each experiment was repeated 50 times, after which all times were averaged. The runtimes for the different files were also averaged to create a single set of timing data per experiment configuration.

5.2 Performance comparison

The first experiment to look at is an experiment using only declarations with no module parameters or aliases. This experiment, found in figure 3, shows that the later type-checkers perform slightly better, but that all implementations behave roughly the same. This makes sense as there are no large differences between the implementations in this case. However, it is useful to observe that the reduction in general complexity in the later versions has performance benefits, even in cases where the complexity is not necessary. This also explains why Agda needs multiple seconds to type-check the files. Agda is a much more complex language and thus the type-checker will always be slower.

The second experiment, shown in figure 4, shows the first clear difference between the implementations. In this experiment, we have created a module of 40 declarations and aliased it. We then increase the size of the argument provided to the module. Both versions 0 and 1, as well as Agda itself, see a significant increase in serialization time. This makes sense as they are all copying the module argument once for each declaration in the module. Version 2 is barely affected by the increasing size as it does not create these copies. Furthermore, we can see that both Agda and version 0 are also affected in their type-checking performance. The reasons for this are not as obvious, but this is likely related to the increased amount of parameter manipulations.

The third experiment, shown in figure 5, shows the clearest difference between the approaches. For this experiment, we have a base module `M0` with 5 declarations and a single module parameter. We then add an increasing number of modules that look like this:

```
module M1 (x : Bool) where
  module M = M0
  module N = M (f x)
  module O = M (g x)

module M2 (x : Bool) where
  module M = M1
  module N = M (f x)
  module O = M (g x)

...
```

This creates a number of nested aliases which will produce an exponential amount of declarations when aliases are expanded, as is the case in all implementations except for version 2, which we can see from the graph is the only version that does not show an exponential increase in both serialization and type-checking time.

Not generating the new declarations does have some disadvantages. When a declaration is used multiple times, the type will have to be generated multiple times as well, while if you generate all declarations immediately, they will only be generated once. This experiment, shown in figure 6, will increasingly make use of aliased declarations in the most extreme case generating declarations such as:

```
M'.d70 (M'.d59 (M'.d87 (M'.d91 (M'.d75 ((\x258. M'.d29
(M'.d40 (M'.d51 (M'.d60 (M'.d45 (M'.d85 (M'.d5 (M'.d68
(M'.d85 (M'.d55 (M'.d98 (M'.d75 (M.d5 (if d191 then x258
else True) (if True then d146 else False)))))))))))) :
Bool -> Bool) (if False then d120 else d212))))))
```

From the results, we can see that this does close the gap between version 1 and version 2 a bit, but even in the worst case version 2 is still faster.

6 DISCUSSION

From the results in the previous section, it should be clear that version 2 is a massive improvement compared to version 0. It has no exponential growth problems and even in its worst case it is still faster than the other versions. Furthermore, the comparison to Agda shows that version 0 behaves the same or better as Agda in the experiments. This means that if we implement the module system of version 2 for Agda, it should also lead to significant improvements.

The performance argument is not the only reason to switch to the new implementation. The system is much simpler to implement and preserves more information to be used by the pretty-printer or eventual compilers. This increase in information allows us to improve the pretty-printing as the display form system can be enhanced [16] and it also allows us to resolve an interesting problem with pretty-printing mixfix operators. At the moment, Agda does not know which parameters are module parameters and which parameters are normal parameters, leading it to pretty-print incorrect terms:

```
module M (A : Set1) where
  if_then_else_ : Bool -> A -> A -> A
x : (b : Bool) ->
```

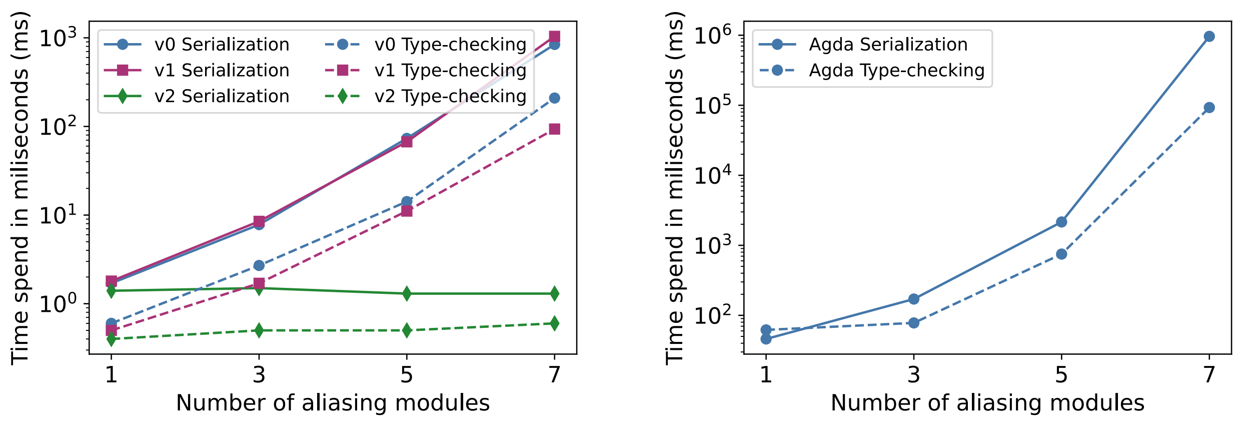

Figure 5: Experiment 3: Nested module aliases

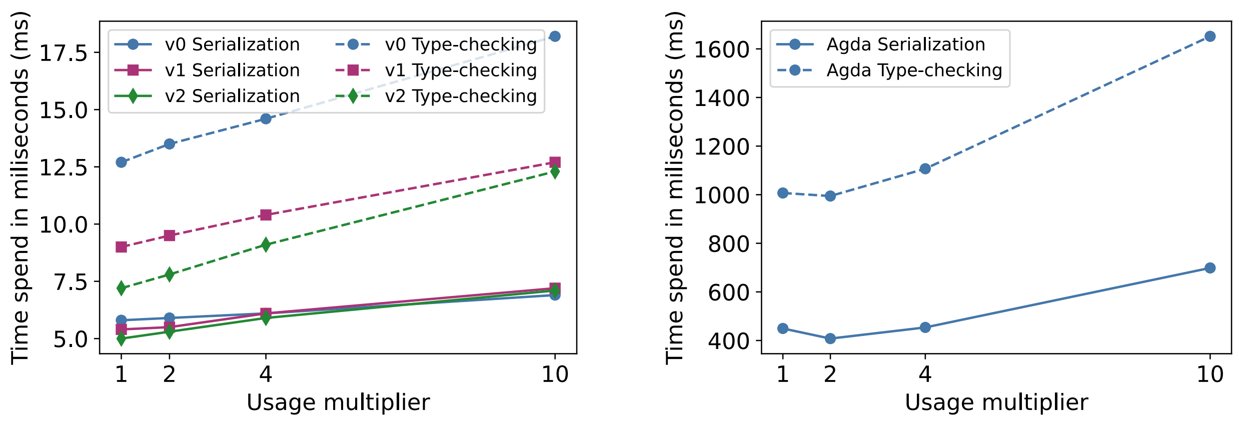

Figure 6: Experiment 4: Increased usage of module aliases

```
  M.if_then_else_ Set b Bool Bool
x b = ?
Goal : (M.if Set then b else Bool) Bool
```

Agda prints the module parameter as if it is the first argument provided to the if statement. With more knowledge, this bug could be prevented and by using term-qualified names, as implemented in version 3, we can even completely fix it:

```
module M (A : Set1) where
  if_then_else_ : Bool -> A -> A -> A
x : (b : Bool) ->
  M.if_then_else_ Set b Bool Bool
x b = ?
Goal: (M Set).if b then Bool else Bool
```

Finally, implementing the changed module system is not very difficult. It will be time-consuming, but for the most part, we are removing complexity in the type-checker not adding complexity. However, there are a lot of side-effects on the Agda codebase which will take time to implement. Qualified names were always considered to be a literal, while now they will contain terms. This will require a refactor, but it has no performance effects as those parameters used to occur somewhere else.

Similar sorts of refactors are required for many other systems which assume that all declarations are global, while now there is a local state as well. This is better, as it allows us to fix a number of bugs, but refactoring all the different systems is time-consuming.

Given these reasons, We believe it is absolutely worth it to switch the module system of Agda to the implementation proposed by version 2. It has large performance benefits with few disadvantages and the hardest part about the refactoring process is finding someone that has the time to do it, not actually doing it.

7 RELATED WORK

This work started with an unpublished, early draft by my supervisor Jesper Cockx [11]. Other than that there is no work that is directly related. Instead, in the remainder of this section, We will go over some other work that might seem related and explain how Agda's problems and our solutions differ.

Agda's current implementation of modules is very similar to how functors in ML-like languages behave [30, 31]. We have parametrised modules that can be instantiated with arguments to create new modules. We can even use records [5] to make a module parametrised over another module.

Active research on improving these kinds of module systems is looking to, for example, unify the module and term-level languages [35] or trying to implement better compiler techniques to ensure things like easy incremental compilation [28, 12].

Agda has different problems. Agda's module aliases can be used to create functor-like behaviour but they are quite different. Modules can be used to instantiate records [5], thereby becoming an object, but they themselves are not objects.

When exponential growth occurs with functors it is often due to code duplication [37, 36]. This can be solved through the clever insertion of let bindings to prevent generating duplicate code [37].

Agda technically has no code duplication as each of the declarations generated has a different name and is thus distinct. However, these generated declarations barely contain any information and there is thus no need to generate all these declarations when most will likely never be used and they can be generated extremely quickly when needed, as we only need to add or remove a couple of parameters.

We could also compare Agda to something like Rust generics [6]. Rust [26] also has to create specialised functions for each instantiation of a generic function. In Agda, there is no need to do this. Generics can be seen as a limited form of dependent-typing which Rust does not support. It, therefore, has to remove this dependency by creating the concrete instances. Agda is dependently typed and instantiating the module parameters is no different from calling a normal function. There is therefore no need to specialize anything.

From these two comparisons, it should be clear that Agda encounters a somewhat unique problem as it is currently treating a scope-checking issue in the type-checker, which is extremely inefficient as we are using many unnecessary techniques.

The final distinction between other work and this work is that we are working with the Agda type-checker whose output is only used to type-check other files, not a compiler that is optimising code to be executed. Some extra work and extra code being produced in a compiler is acceptable as long as it has benefits during the execution of the compiled code. For a type-checker, this is not the case. The type-checker itself should be very fast. The compiler afterwards can deal with the efficiency of the final outputted code.

8 CONCLUSION

The main goal of this work was to improve the performance of Agda's module system. We have created a simpler language called Simple Agda to evaluate the performance of three different approaches: Agda's current approach, keeping modules and module parameters intact and keeping modules and module aliases both intact. We have introduced the concept of term-qualified names, which allows us to implement these later versions with ease, by realising that much of the difficulty of Agda's module system can be resolved by the scope-checker.

We have evaluated these approaches in a variety of scenarios using randomly generated files. These experiments showed that the best approach is to keep modules and module aliases intact during type-checking. This will perform better in all evaluated scenarios and completely eliminates the exponential complexity of Agda's current system when aliases are nested.

Furthermore, we have seen that this change will allow for several other problems to be addressed as well. The improved module system makes use of term-qualified names internally: `(M True).f`. Allowing this syntax to be used when programming in Agda will remove a significant number of pretty-printing problems. The improved performance of module aliases also means that open public statements can be changed to a more intuitive implementation. This was not yet possible due to the performance bottle-necks.

The performance benefits combined with the other benefits mean that making the proposed changes to Agda will massively improve the user experience as some long-standing problems are eliminated. The performance problems especially have hampered the development of, for example, category theory proofs as these benefit massively from module aliases, which so far, could not be used extensively.

8.1 Future work

There are two major areas related to Agda's module system that could be improved in future work. The first such area is pretty-printing. For example, it will need to be decided how we want to qualify terms. Do we keep the alias qualifier when evaluating, akin to a sort of dynamic dispatch, or do we reduce it to the aliased term and start fully reducing terms? Now that we maintain aliases after type-checking, such questions and many others can start to be analysed in much more detail.

The second major area for future work are Agda's records. Agda's record functionality has been extended multiple times in the past few years but this often occurred in an isolated manner. This means that it is unclear what the various interactions between the record features are. Furthermore, there is also a lack of consensus on how records should interact with the module system. Now that the module system has been cleared up more, it is time to do the same for the record system and see how it should interact with itself and with modules.

REFERENCES

[1] Andreas Abel. 2022. Exponential module chain leads to infeasible scope checking. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/1646.

[2] Andreas Abel. 2016. Not a splittable variable. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/2181.

[3] Andreas Abel. 2019. Printer prefers (longer) qualified over (shorter) unqualified name. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/3240.

[4] [SW] Agda Community, Agda version 2.6.2.2, Mar. 27, 2022. URL: https://github.com/agda/agda/tree/v2.6.2.2.

[5] Agda Language Reference. 2023. Record types. Retrieved May 4, 2023 from https://agda.readthedocs.io/en/latest/language/record-types.html.

[6] Brian Anderson. 2020. Generics and compile-time in rust. PingCAP. (June 15, 2020). Retrieved Feb. 9, 2023 from https://www.pingcap.com/blog/generics-and-compile-time-in-rust/.

[7] Henk Barendregt and Herman Geuvers. 2001. Proof-assistants using dependent type systems. In *Handbook of automated reasoning*. Elsevier Science Publishers B. V., NLD, (Jan. 1, 2001), 1149–1238. ISBN: 978-0-444-50812-6. Retrieved Feb. 3, 2023 from.

[8] Jacques Carette. 2022. Switch to a structured signature? Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/4331.

[9] Jacques Carette and Jason Hu. 2021. Formalizing Category Theory in Agda. DOI: 10.1145/3437992.3439922.

[10] Liang-Ting Chen. 2022. Qualified names are printed if introduced by ‘open M ...‘. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/5632.
[11] Jesper Cockx. Mini modules: a structured module system with parametrized modules and dependent types. (2020).
[12] Karl Crary. 2019. Fully abstract module compilation. *Proc. ACM Program. Lang.*, 3, POPL, Article 10, (Jan. 2019), 29 pages. DOI: 10.1145/3290323.
[13] Nils Anders Danielsson. 2019. Record constructors sometimes in record modules, sometimes not. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/4189.
[14] Nils Anders Danielsson. 2020. Shadowing parameters are sometimes renamed. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/2018.
[15] N. G. de Bruijn. 1991. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91, 2, (Apr. 1, 1991), 189–204. DOI: 10.1016/0890-5401(91)90066-B.
[16] Ivar de Bruin. 2023. *Improving Agda's module system.* TODO. TODO, Delft, The Netherlands. TODO. TODO.
[17] [SW] Facundo Domínguez, timestats version 0.1.0, July 13, 2022. URL: https://hackage.haskell.org/package/timestats-0.1.0.
[18] Paolo G. Giarrusso. 2018. Regression with open public. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/1985.
[19] Google Code Exporter. 2018. Change the semantics of open public in parameterised module. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/892.
[20] Google Code Exporter. 2015. Copatterns do not work in parametrized modules. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/940.
[21] Google Code Exporter. 2022. Printing of infix/mixfix operators defined in parametrized modules. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/632.
[22] Google Code Exporter. 2015. Undeclared name accepted in fixity declaration. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/329.
[23] Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation.* Andrew M. Pitts and P. Dybjer, (Eds.) Cambridge University Press, 79–130. DOI: 10.1017/CBO9780511526619.004.
[24] Arjen Jonathan. 2022. Unnecessary conversion checking due to parameterized module slows type-checking (a lot). Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/4517.
[25] Wolfram Kahl. 2015. Regression: Module parameters lost. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/1701.
[26] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* (1st Edition ed.). No Starch Press, San Francisco, (June 26, 2018). 552 pp. ISBN: 978-1-59327-828-1.
[27] 2005. The λ-calculus. In *Abstract Computing Machines: A Lambda Calculus Perspective.* Texts in Theoretical Computer Science. W. Kluge, W. Brauer, G. Rozenberg, and A. Salomaa, (Eds.) Springer, Berlin, Heidelberg, 51–88. ISBN: 978-3-540-27359-2. DOI: 10.1007/3-540-27359-X_4.
[28] George Kuan. 2010. *True Higher-Order Module Systems, Separate Compilation, and Signature Calculi.* Ph.D. Dissertation. University of Chicago, Chicago IL USA, (June 2010). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=07258e13737477d9ca682db338325ea98189bdc9.
[29] Xavier Leroy, Sandrine Blazy, Zaynah Dargaye, Jacques-Henri Jourdan, Michael Schmidt, Bernhard Schommer, and Jean-Baptiste Tristan. 2022. Compcert c verified compiler. Retrieved Apr. 10, 2023 from https://compcert.org/compcert-C.html.
[30] David MacQueen. 1984. Modules for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (LFP '84). Association for Computing Machinery, Austin, Texas, USA, 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036.
[31] David MacQueen, Robert Harper, and John Reppy. 2020. The history of standard ml. *Proc. ACM Program. Lang.*, 4, HOPL, Article 86, (June 2020), 100 pages. DOI: 10.1145/3386336.
[32] Orestis Melkonian. 2023. Unsafe(?) irrelevant projections by ‘open‘ing. Retrieved Mar. 31, 2023 from https://github.com/agda/agda/issues/6359.
[33] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology and Göteborg University, Göteborg, Sweden. 166 pp. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.
[34] [SW] Egbert Rijke, Elisabeth Bonnevier, Jonathan Prieto-Cubides, Fredrik Bakke, et al., Univalent mathematics in Agda. URL: https://github.com/UniMath/agda-unimath/.
[35] ANDREAS ROSSBERG. 2018. 1ml – core and modules united. *Journal of Functional Programming*, 28, e22. DOI: 10.1017/S0956796818000205.
[36] Yuhi Sato and Yukiyoshi Kameyama. 2021. Type-safe generation of modules in applicative and generative styles. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* GPCE '21: Concepts and Experiences. ACM, Chicago IL USA, (Oct. 17, 2021), 184–196. ISBN: 978-1-4503-9112-2. DOI: 10.1145/3486609.3487209.
[37] Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. 2020. Module generation without regret. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (PEPM 2020). Association for Computing Machinery, New York, NY, USA, (Jan. 20, 2020), 1–13. ISBN: 978-1-4503-7096-7. DOI: 10.1145/3372884.3373160.