Improving Agda's module system*

Ivar de Bruin
Bohdan Liesnikov
Jesper Cockx
Technical University Delft
Delft, Netherlands

ABSTRACT

Agda is a dependently typed programming language and proof assistant. It has a module system that provides namespacing, module parameters and module aliases, which can be used to write shorter and cleaner programs and proofs. However, the current implementation of the module system has several issues, including an exponential slowdown in the desugaring of module aliases. This paper shows how the implementation of the module system can be changed to address these issues. Instead of desugaring modules during type checking, we handle module parameters and aliases directly during name lookup in the scope checker, simplifying the implementation and eliminating the exponential behavior. Thus we allow users to make full use of the module system where doing so was previously too costly. Furthermore, our changes to the module system also pave the way to fix other issues in Agda's implementation of pretty-printing, records and `open public` statements.

1 INTRODUCTION

Throughout programming history, programmers have struggled with bugs. This is especially true in security-related programs where bugs can have drastic consequences. One possible solution is to verify the correctness of the program using a proof assistant such as Coq, Lean, or Agda. This has notably seen use in, for example, the creation of a C compiler for critical applications [31].

To create such proofs, programmers work in proof assistants [9]. Many of these proof assistants are also programming languages with strong type systems and termination checkers that enable writing proofs as programs through the Curry-Howard correspondence. 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 the Haskell back-end. This workflow produces code that is optimised for performance while maintaining the guarantee that the proven properties hold. Agda is also used to prove various mathematical properties or even formalize entire mathematical fields [13, 36].

To help programmers structure larger programs and proofs, Agda provides a module system. Modules allow for easy namespacing and grouping of definitions. In addition, there are two interesting features that can be of great help when writing proofs: module parameters and module aliases. Module parameters are declared once and are then in scope for all declarations in that module, while module aliases introduce new names for a module that instantiates these parameters with specific values. As an example, consider the module `CatProofs` that contains a variety of proofs generalised over a `Category` c, and a module alias `GroupCatProofs` specializing these proofs to the category of groups `groupCategory`.

```
module CatProofs (c : Category) where
  -- ...
module GroupCatProofs = CatProofs groupCategory
```

Structuring proofs this way increases their readability as well as ease of writing.

Unfortunately, the current implementation of Agda (version 2.6.4) does not perform well when many aliases are used. The problem is caused by how Agda desugars each module alias to a set of specialized functions, which is similar to how other compilers such as the Rust compiler deal with parametrized functions [18]. 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 error-prone due to the increase in transformations needed. Finally, any information about module aliases is removed by the type checker, so it is not available to the code generation back-end, making back-end features more difficult to implement.

In this paper we analyse different possible implementations of a simplified version of Agda's module system. The goal is to find an implementation strategy that removes the 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 the implementation of 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 demonstrate how these can be used to remove the need to desugar modules and module aliases during type checking, simplifying the implementation (Section 4).
- We implement three different type checkers for a **minimal version of Agda with parametrized modules and module aliases** and **evaluate their performance** on randomly generated Agda files, which shows that keeping aliases intact has far superior performance, with barely any downsides. We also verify the validity of our experiments by comparing the performance of our baseline implementation to the actual Agda implementation (Section 5).

*This is a shortened and updated version of the MSc thesis by the first author [11]. All code and results used in this paper can be found at: https://github.com/ivardb/AgdaModuleImprovement

- Based on our results as well as other factors such as the difficulty of refactoring Agda, we provide **advice for the future evolution of Agda's module system**, arguing that it should preserve module aliases (Section 6).

2 AGDA'S MODULE SYSTEM

This section will cover the required background information on Agda's module system. We assume that readers are already familiar with dependent typing [25] and its associated concepts such as weak head normal form (WHNF) [30] and telescopes [10]. A full description of Agda's module system can be found in chapter 4 of Norell's thesis [35].

Agda's module system. In addition to standard namespacing, modules in Agda can be nested and have parameters:

```
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
```

In the function call `M1.f true`, there is no need to provide a value for the parameter `x` as it is still in scope. However, when a declaration is used outside of its module, the module parameters have to be supplied just like normal function parameters, so we do need to provide an argument corresponding to the parameter `y`.

When using the same module with the same module arguments in different places, it can be useful to define a module alias:

```
module N (X : Type) where
  id : X → X
  id x = x
module NBool = N Bool
```

A module alias creates a copy of the module that instantiates zero or more of the parameters. Module parameters and module aliases can be extremely useful when writing a collection of proofs that all depend on a number, a category, or some other value.

Desugaring parametrized modules. The desugaring of parametrized modules happens in three steps. First, Agda replaces each name by its fully qualified version. Next, it lifts all declarations to the top level, turning module parameters into regular function parameters. Third, each function call is given additional arguments corresponding to the module parameters that are in scope for both the function call and the definition. For example, the module `M` above is desugared into the following:

```
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
```

The call to `M.M1.f` needs to be explicitly passed the `x` argument as it is no longer automatically in scope.

Desugaring module aliases. To desugar a module alias, new definitions are introduced: for each definition in the original module, we need a definition in the new module that redirects to the old definition, passing the appropriate arguments. For example, the module `N` above is desugared into:

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

NBool.id : Bool -> Bool
NBool.id = N.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.

Interface files. In order to avoid having to re-check a file that has already been checked, Agda stores information about the type-checked code in an *interface file*. The information stored in these files is then used whenever that file is imported, avoiding the need to type-check it again.

Such an interface file consists of two parts: the sections and the declarations. Each section corresponds to a module with its full set of parameters, but without any declarations. This information is needed to type-check module aliases. For example, the interface file generated from the module `M` above will contain the following sections:

```
section M (x : Bool)
section M1 (x : Bool) (y : Bool)
section M2 (x : Bool) (z : Bool)
```

After the sections come declarations of the functions and datatypes together with their given or inferred type signatures. The declarations are all using fully qualified names and take all module parameters as additional parameters.

3 PROBLEM DESCRIPTION

The implementation of Agda's module system has several problems, many of which are reported as issues on GitHub[1]. Before diving into these, we first describe a more general problem: the loss of information caused by the desugaring of modules and module aliases.

Agda allows for the implementation of custom back-ends which can be used to compile Agda to different languages. However, by the time the program is given to the back-end, its modules are already removed and the module parameters are moved to the declaration level. This means that the back-end is not able to decide itself how to handle these features. This rules out certain kinds of back-ends, for example a back-end that applies a transformation and then returns valid Agda code. As a real-world example, the `agda2hs` backend has to go through great lengths to remove the parameters that Agda adds to each function declaration in a `where` block[2].

Apart from this conceptual problem, there is a number of issues related to the module system that have been reported on the Agda

[1] https://github.com/agda/agda/issues?q=label%3Amodules+sort%3Aupdated-desc+
[2] https://github.com/agda/agda2hs/blob/32cdc891d1190bb6ca86a5f16add241f1d0f58b7/src/Agda2Hs/Compile/Function.hs#L247-L281

issue tracker on GitHub. We divide these issues into three major groups:

(1) **Lack of module structure in the typing environment:** The first class of issues is caused by the lifting of declarations from (parametrized) modules to the top level. In the past, there have been bugs in this lifting due to wrong renaming [17] or parameters missing from function arguments [22, 28, 2]. While these bugs have been fixed, others are still open. One such issue is issue #6359, which is caused by a declaration that should only exist locally being added to the global scope, thereby breaking the tracking of specific data and ultimately resulting in an illegal function call being accepted [34]. These issues show that the current approach of Agda is both complex and restrictive.
(2) **Performance problems:** The second class of issues is caused by the desugaring of module aliases. This desugaring causes poor performance either by inserting large numbers of module parameters [27] or more often by the large numbers of declarations that are generated by the desugaring of module aliases [1, 12]. The current behavior of the `open public` statement is also considered to be unintuitive by many users [20, 16] due to its behavior being different depending on whether module parameters are provided or not. However, making the behavior consistent would require every open public statement to be handled as a module alias, which is considered to be be too big of a performance sacrifice with the current implementation [21].
(3) **Pretty-Printing problems:** The final class of problems is related to the pretty-printing of calls to functions that come from a (parametrized) module. Either Agda forgets from which module alias a definition originated [24, 3] or it loses track of module parameters, making infix operators especially confusing to read [23, 14].

4 A STRUCTURED MODULE SYSTEM

Making changes to the actual Agda code base is not a quick process. To be able to prototype a number of different approaches we introduce a simplified version of Agda, called Simple Agda (Section 4.1). To handle parametrized modules, Simple Agda has two important components: it uses term-qualified names in the syntax (Section 4.2) and a structured signature that preserves both modules and module aliases (section 4.3). Finally, we define the signature lookup operation that is required to look up term-qualified names in the structured signature (section 4.4).

4.1 Simple Agda

Simple Agda supports dependent function types, a single universe `Type` with `Type : Type`, and primitive types `Unit` and `Bool`. The full syntax of Simple Agda can be found in Grammar 1. It does not have support for implicit arguments, universe levels, or inductive data types, as these do not impact the module system, only the complexity of the implementation.

As Simple Agda is dependently typed, we need to be able to evaluate terms to WHNF during type checking. The evaluation rules can be found in figure 1. The typing rules of Simple Agda are standard and can be found in figure 2.

A, B, u, v	$::=$	x	*Variable*
		$\mid$ `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*
		$\mid \lambda x.\ u$	*Lambda*
		$\mid u\ v$	*Application*
Δ, Γ	$::=$	$\epsilon \mid (x : A)\Delta$	*Contexts*

Grammar 1: Simple Agda grammar, without modules

$$\frac{}{\Sigma \vdash (\lambda x.\ u)\ v \longrightarrow u[x := v]} \qquad \frac{\Sigma \vdash u \longrightarrow u'}{\Sigma \vdash u\ v \longrightarrow u'\ v}$$

$$\frac{}{\Sigma \vdash \texttt{if true then } v \texttt{ else } w \longrightarrow v}$$

$$\frac{}{\Sigma \vdash \texttt{if false then } v \texttt{ else } w \longrightarrow w}$$

$$\frac{\Sigma \vdash u \longrightarrow u'}{\Sigma \vdash \texttt{if } u \texttt{ then } v \texttt{ else } w \longrightarrow \texttt{if } u' \texttt{ then } v \texttt{ else } w}$$

Figure 1: Small-step evaluation of Simple Agda terms

So far Simple Agda has no way to introduce new functions or modules. This is our focus for the rest of this section.

$$\frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x : A} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{Type} : \texttt{Type}}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{Bool} : \texttt{Type}} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{Unit} : \texttt{Type}}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{false} : \texttt{Bool}} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{1} : \texttt{Unit}}$$

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

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

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

$$\frac{\Sigma; \Gamma \vdash u : A \qquad \Sigma \vdash A \longrightarrow^* C \qquad \Sigma \vdash B \longrightarrow^* C}{\Sigma; \Gamma \vdash u : B}$$

Figure 2: Typing rules for Simple Agda terms

$$\frac{\Sigma;\Gamma \vdash A : \texttt{Type}}{\Sigma;\Gamma \vdash f : A} \qquad \frac{\Sigma;\Gamma \vdash u : A}{\Sigma;\Gamma \vdash f = u}$$

$$\frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma;\Gamma\Delta \vdash \Sigma'}{\Sigma;\Gamma \vdash \texttt{module}\ M\ \Delta\ \texttt{where}\ \Sigma'} \qquad \frac{\Sigma;\Gamma \vdash \Delta \qquad \Sigma;\Gamma\Delta \vdash \Sigma?\alpha}{\Sigma;\Gamma \vdash \texttt{module}\ M\ \Delta = \alpha}$$

Figure 3: Typing rules for declarations in Simple Agda

4.2 Term-qualified names

Looking back at the desugaring of the module `M` in section 2, the inserted parameters are always exactly those belonging to the modules whose names are added to create fully qualified names. So instead of transforming module parameters into function arguments, we introduce *term-qualified names* to the syntax in grammar 2.

A, B, u, v	::=	$\cdots$	
	$\mid$	$\alpha.f$	*Term-qualified name*
α, β	::=	ι	*No qualifier*
	$\mid$	$(M\ \bar{u}).\alpha$	*Module qualifier*

Grammar 2: Term-qualified names

Term-qualified names extend the syntax of qualified names to allow passing arguments directly to the module. For example, this allows us to write `(M x).M1.f true` instead of `M.M1.f x true`, keeping module arguments next to the module they belong to.

Since the scope checker already replaces each name with its fully qualified version, it is easy to add the appropriate module arguments at the same time. The type checker now no longer needs to deal with the moving around and insertion of module parameters.

4.3 A structured signature

To facilitate working with term-qualified names, we switch to a *structured signature* that preserves modules and their parameters. The syntax for structured signatures is given in grammar 3.

decl	::=	$f : A$	*Type declaration*
	$\mid$	$f = u$	*Function definition*
	$\mid$	$\texttt{module}\ M\ \Delta\ \texttt{where}\ \Sigma$	*Module definition*
	$\mid$	$\texttt{module}\ M\ \Delta = \alpha$	*Module alias*
Σ	::=	$\epsilon \mid \mathit{decl}, \Sigma$	*Declaration signature*

Grammar 3: Structured signature

We define the typing rules for declarations in Simple Agda in figure 3. Checking a type declaration or a function definition amounts to checking that respectively the type or the body is well-typed. There is no check that each symbol is only declared and defined once or that its declaration and definition are consistent, but such a check can easily be added. For typing a module, we check that the telescope is well-formed and that the declarations in it are valid. For module aliases, we also first check the telescope and then use the typing rules for qualifier from figure 7 to check that the alias is well-formed (see section 4.4).

$$\frac{(\texttt{module}\ M\ \Delta = \Sigma') \in \Sigma \qquad (\Delta_1, \Delta_2) = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma_0 \vdash \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta', \Sigma'')}{\Sigma_0 \vdash \Sigma!(M\ \bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta', \Sigma'')}$$

$$\frac{(\texttt{module}\ M\ \Delta = \beta) \in \Sigma \qquad (\Delta_1, \Delta_2) = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma_0 \vdash \Sigma_0!\beta \rightsquigarrow (\Delta', \Sigma') \qquad \Sigma_0 \vdash \Sigma'[\Delta_1 := \bar{a}]!\alpha \rightsquigarrow (\Delta'', \Sigma'')}{\Sigma_0 \vdash \Sigma!(M\ \bar{a}).\alpha \rightsquigarrow (\Delta_2\Delta'\Delta'', \Sigma'')}$$

$$\frac{}{\Sigma_0 \vdash \Sigma!\iota \rightsquigarrow (\epsilon, \Sigma)}$$

Figure 4: Signature lookup

$$\frac{\Sigma \vdash \Sigma!\alpha \rightsquigarrow (\Delta, \Sigma') \qquad f = u \in \Sigma'}{\Sigma \vdash \alpha.f \longrightarrow \lambda\Delta.\ u}$$

Figure 5: Evaluation of term-qualified names

In contrast to the current specification of Agda's module system [35], these typing rules do not need to insert any parameters, nor do we need to create or modify any declarations for dealing with module aliases.

4.4 Signature lookup

Term-qualified names need to be handled properly during evaluation of expressions. For example, the term `(M false).(M2 false).g` evaluates in one step to `(M false).(M1 true).f`, which evaluates further to `false`.

Signature lookup is the process of finding the declaration corresponding to a given (term-qualified) name. To formally define signature lookup, we use the syntax $\Sigma_0 \vdash \Sigma!\alpha \rightsquigarrow (\Delta, \Sigma')$ to say that we are looking for the signature belonging to the module α relative to Σ and that this results in the signature Σ'. Here the telescope Δ represents the remaining module parameters in case any module in α is underapplied, and the signature Σ_0 represents the root signature that is used for looking up module aliases.

The definition of signature lookup can be found in figure 4. In the case where we encounter a module application, we split the module telescope into two (potentially empty) parts: the part of the parameter telescope for which arguments are provided, and the remaining part. Next, we substitute the arguments into the module's signature and look up the remaining qualifier in it. In an actual implementation the substitution on the signature would be implemented lazily as there is no need to perform it on the whole signature. When we encounter a module alias during lookup, we need to find the signature belonging to the aliased module. As we use fully qualified names, this lookup is relative to the root signature instead of relative to the current signature. We also split the telescope of the module being aliased, as it is not required to provide an argument for each of the parameters in an alias. The final remaining telescope consists of the unused parameters of the module alias (Δ_2), the unused parameters of the aliased qualifier (Δ'), and the unused parameters of the remaining qualifier (Δ'').

$$\frac{(\texttt{module}\ M\ \Delta = \Sigma') \in \Sigma \qquad (\Delta_1, \Delta_2) = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma_0; \Gamma \vdash \bar{a} : \Delta_1 \qquad \Delta_2 = \epsilon \text{ OR } \texttt{noArgs}(\alpha) \qquad \Sigma_0; \Gamma \vdash \Sigma'[\Delta_1 := \bar{a}]?\alpha}{\Sigma_0; \Gamma \vdash \Sigma?(M\ \bar{a}).\alpha}$$

$$\frac{(\texttt{module}\ M\ \Delta = \beta) \in \Sigma \qquad (\Delta_1, \Delta_2) = \texttt{split}(\Delta, \bar{a}) \qquad \Sigma_0; \Gamma \vdash \bar{a} : \Delta_1 \qquad \Delta_2 = \epsilon \text{ OR } \texttt{noArgs}(\alpha) \qquad \Sigma_0 \vdash \Sigma_0!\beta \rightsquigarrow (\Delta', \Sigma') \qquad \Sigma_0 \vdash \Sigma'[\Delta := \bar{a}]?\alpha}{\Sigma_0; \Gamma \vdash \Sigma?(M\ \bar{a}).\alpha}$$

$$\frac{}{\Sigma_0; \Gamma \vdash \Sigma?\iota}$$

Figure 6: Typing rules for qualifiers

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

Figure 7: Typing of term-qualified names

Signature lookup is used for evaluation of term-qualified names, as shown in figure 5. The unused parameters Δ become additional arguments to the function by introducing an iterated lambda $\lambda\Delta$.

When evaluating a term we can typically assume that the term is well-typed. However, when type checking a qualified name $\alpha.f$, we should also check that all the terms embedded in the qualifier α are well-typed as arguments to their respective modules. This is done through the auxiliary judgement $\Sigma_0; \Gamma \vdash \Sigma?\alpha$ as defined in figure 6. If we encounter an underapplied module (i.e. Δ_2 is not empty), we forbid any further module parameters from appearing in the rest of the qualifier with the side condition $\texttt{noArgs}(\alpha)$. This is because the types of these parameters could depend on the missing parameters, which would be problematic. Since the current implementation of Agda requires parameters to be given sequentially, this is not removing any expressivity from the language.

Finally, the typing rule for term-qualified names is given in figure 7. The unused parameters Δ here become an iterated function type $\Delta \to$.

5 EXPERIMENTAL EVALUATION

This section will cover the experiments used to measure the performance improvement of our proposed changes to the implementation of Agda's module system. The experiments were executed for our implementations of Simple Agda as well as on the current implementation of Agda itself to verify the accuracy of our baseline. We will only cover four of our experiments in this paper, the remaining experiments can be found in chapter 7 of the full thesis [11].

5.1 The implemented type checkers

For the experimental evaluation, we implement three versions of a type checker for Simple Agda, differing only in their handling of modules:

- **Version 0** uses the approach Agda currently takes and lifts all declarations to the top level, removing all module features. This version will serve as the baseline of our experiments.
- **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.
- **Version 2** makes 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.

In addition to the three versions, we implemented a **version 3** that also exposes term-qualified syntax to the programmer. This does not change the performance, but does increase expressivity and allows us to fix the pretty-printing issues with infix and mixfix operators [23]. An analysis of how other pretty-printing issues related to the module system could be addressed in future work is discussed in section 5.3 of the full thesis [11].

5.2 Experiment setup

For our evaluation, we need to be able to to isolate the impact of specific aspects of the module system on the performance, and we need to make sure that our test files do not use Agda features that are not supported in Simple Agda. Hence we create a generator that can generate files according to some parameters, such as the size of the arguments provided to a module alias. Using this generator, we generate 15 files for each of our experiment configurations. These files are then converted from Simple Agda to Agda to allow them to be checked by Agda as well.

The Simple Agda experiments were timed using the timestats library for Haskell [19], 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.

5.3 Performance comparison

The first experiment uses only declarations with no module parameters or aliases. This experiment, found in figure 8, shows that the later type checkers perform slightly better, but that all implementations have roughly the same asymptotic behaviour. This makes sense as there are no large differences between the implementations in this case. However, it is useful to observe that the simplication of the code 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: it is a much more complex language with many checks not included in Simple Agda (e.g. universe levels and termination checking) and thus the type checker will always be slower.

The second experiment, shown in figure 9, shows the first clear difference between the implementations. In this experiment, we have created a module with 40 declarations and aliased it. We then

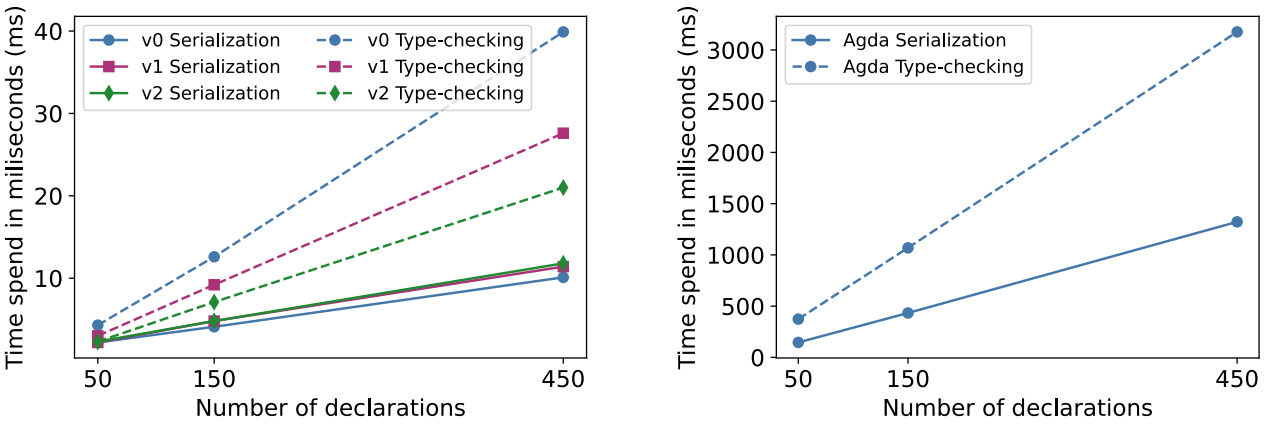

Figure 8: Experiment 1: Only declarations

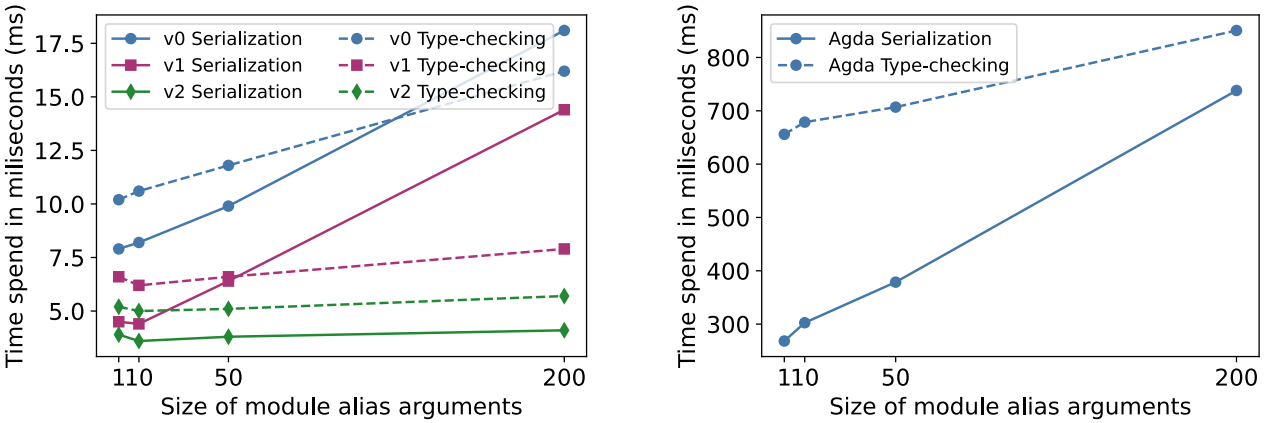

Figure 9: Experiment 2: Module argument size

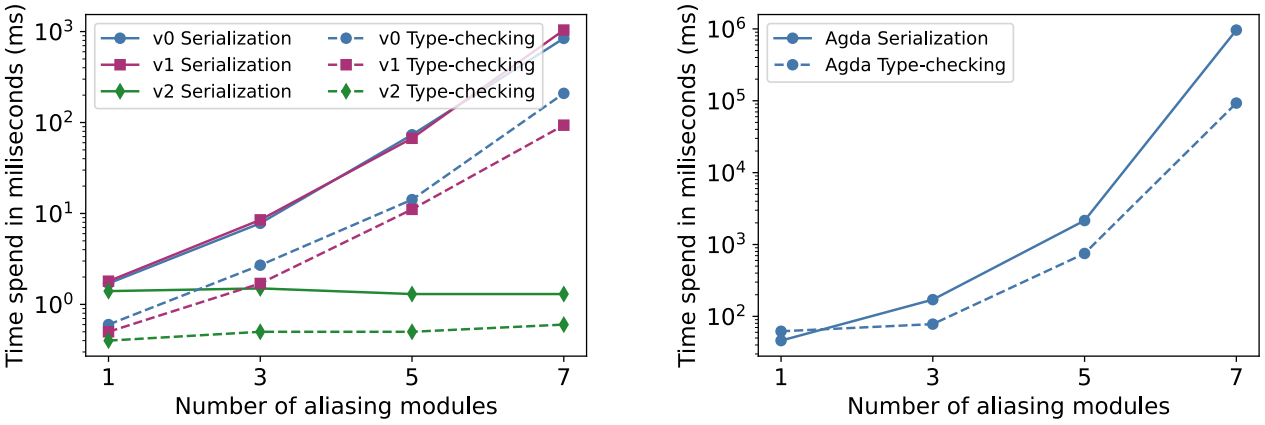

Figure 10: Experiment 3: Nested module aliases

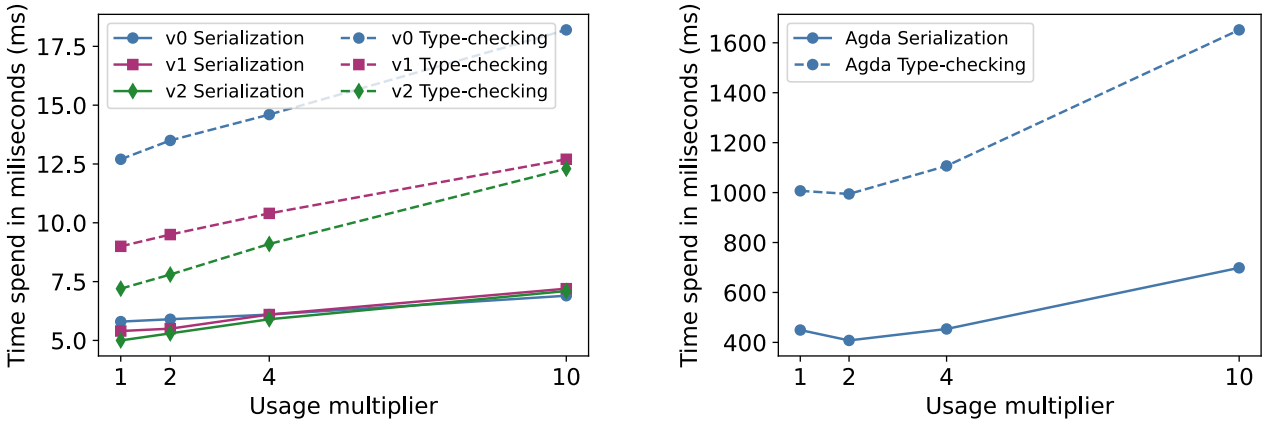

Figure 11: Experiment 4: Increased usage of module aliases

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 10, 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 M0 (x : Bool) where -- ...

module M1 (x : Bool) where
  module P = M0
  module Q = P (f1 x)
  module R = P (f2 x)

module M2 (x : Bool) where
  module P = M1
  module Q = P (f1 x)
  module R = P (f2 x)

-- ...
```

This creates a number of nested aliases which produces an exponential number of declarations when aliases are expanded. This is the case for all implementations except for version 2: as we can see from the graph (note the exponential scale!), this 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. To measure the impact of this, the fourth experiment, shown in figure 11, uses an increasing number of aliased declarations, in the most extreme case generating expressions 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))))))))))))))
  (if False then d120 else d212))))))
```

From the results, we can see that this scenario is indeed less favorable to version 2, but even in this worst-case scenario it is still the fastest.

6 DISCUSSION

From the results in the previous section, it is clear that version 2 shows a significant improvement in performance compared to version 0. It has no exponential growth in code size 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 is reasonable to expect that this will also lead to significant improvements.

The performance argument is not the only reason to switch to the new implementation. The system is simpler to implement and preserves more information that can be used by different backends. This preserved information allows us to improve the pretty-printing of names from module aliases [11], and it also allows us to resolve an interesting problem with pretty-printing of mixfix operators. To understand the latter, let us take a look at the following Agda file:

```
module If (A : Type) where
  if_then_else_ : Bool → A → A → A
  if true  then x else y = x
  if false then x else y = y
if-lemma : (b : Bool) →
  If.if_then_else_ Bool b true false ≡ b
```

```
if-lemma b = {! !}
-- Goal : (If.if Bool then b else false) true ≡ b
```

At the moment, Agda can not distinguish parameters that belong to the module and those belonging to the function, leading it to pretty-print the type of the goal as:

```
(If.if Bool then b else false) true ≡ b
```

This shows that Agda confuses the module parameter for the first argument to the `if` statement. With the additional knowledge about which arguments are module parameters, this bug can be prevented, pretty-printing the goal instead as:

```
If.if_then_else_ Bool b false true
```

By exposing term-qualified names to the user, as implemented in version 3, we can do even better by printing the goal as:

```
(If Bool).if b then false else true
```

The main argument against changing the current implementation of Agda's module system to the one we propose is the time and effort it will take. Since Agda is a much larger language than Simple Agda and the module system intersects with many other of its features, the implementation will certainly be time-consuming. In particular, qualified names were always considered to be a simple value, while now they will have to contain terms. This will require refactoring the representation of names used internally, but it should have no performance effects as those parameters are already present as function arguments in the current implementation. Similar sorts of refactors are required for many other parts of the Agda type checker that assume that all declarations are global, while now there is a local state as well.

Ultimately, we expect the final result of such an effort will reduce the complexity rather than adding to it. In particular, the clearer separation between module parameters and regular function arguments will greatly reduce the risk of bugs related to module parameters in the future. Given the performance benefits and the greater clarity of the implementation, we believe switching the module system of Agda to the implementation proposed by version 2 is a worthwhile endeavour.

7 RELATED WORK

In this section, we discuss a number of other module systems and their implementations, and explain how their problems and solutions differ from the ones in Agda.

ML modules. While modules in Agda are parametrized over values, *functors* in ML and ML-like languages are modules that are parametrized over other modules [32, 33]. There have been attempts to unify the module and term-level languages [37], which would also allow modules parametrized by terms as in Agda. Compared to ML, Agda modules themselves are not first-class, but ML-style functors can be simulated by using records [5]. When there have been problems with exponential growth in code size in ML [39, 38], this has been caused by the compiler actually duplicating full definitions. Agda does not have this problem, as it only ever generates aliases to existing definitions, which can be quickly generated on-the-fly when needed, as demonstrated by our implementation.

Rust generics. Generic functions in Rust [29, 6] and other languages are also specialized to concrete types during compilation. This specialization requires some extra work and produces larger code, but this is an acceptable trade-off as long as it has benefits during the execution of the compiled code. In contrast, in a type checker like Agda we are more concerned with the speed of the type checking itself, and we can leave optimization of the code to the back-end compiler.

Lean namespaces and sections. The Lean proof assistant has support for nested, hierarchical namespaces as well as sections parametrized over variables [41]. Together these can simulate Agda modules, but not module instantiations or module aliases. Since there is no way to specialize a section to specific values, there is also no risk of code size explosion.

Coq modules. The Coq proof assistant has an extensive module system including module types, nested modules, higher-order functors and module subtyping [40, 15]. The module structure is preserved in the kernel of Coq so there is no code generation and hence no risk of exponential code explosion. There is no way to directly parametrize a module by a term-level variable, so arguments to functors are typically smaller than module arguments in Agda.

Isabelle locales. Locales are Isabelle's approach for dealing with parametric theories [8, 7]. They provide a way to define a shared set of parameters and assumptions about those parameters. Each locale can then be *extended* with definitions and theorems. This can happen at any point in the program, making locales more flexible than Agda modules at the cost of making name lookup more complicated. A locale can also extend another locale by *importing* it, which corresponds to a nested module in Agda. Finally, a locale can be *interpreted* by instantiating its parameters, which creates qualified copies of each definition of the locale. Ballarin describes how locales are implemented by means of a *development graph* [26], which serve a similar role to the structured signature we introduced in our implementation. Since Isabelle uses classical logic and does not by default produce proof terms, there are no issues with exponential growth.

8 CONCLUSION AND FUTURE WORK

The main goal of this paper was to demonstrate a clear path towards improving 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 (version 0), keeping modules and module parameters intact (version 1) and keeping modules and module aliases both intact (version 2). We have introduced the concept of term-qualified names, easing the implementation of these later versions by resolving much of the difficulty of Agda's module system already during scope checking.

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 has the best performance in all evaluated scenarios and completely eliminates the exponential complexity of Agda's current system when aliases are nested.

Furthermore, this change allows for several other problems to be addressed as well. The improved module system uses 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 bottleneck.

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

There are two major areas related to Agda's module system that should be looked at in future work: evaluation and record types. For evaluation, it will need to be decided how this should handle qualified names. Do we keep the alias qualifier when reducing terms, 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 analyzed in much more detail.

The second major area for future work are Agda's record types. Currently, each declaration of a record type in Agda generates a new module. Apart from the fields, this module can also contain some other declarations (such as simple function definitions) but not others (such as datatypes). At the moment, when creating an alias to an application of a record module, these declarations are not printed correctly. More generally, it is unclear how record types should interact with the module system. Once the module system has been cleaned up, the next step is to do the same for the record system too.

REFERENCES

[1] Andreas Abel. *Exponential module chain leads to infeasible scope checking*. 2022. URL: https://github.com/agda/agda/issues/1646 (visited on 03/31/2023).

[2] Andreas Abel. *Not a splittable variable*. 2016. URL: https://github.com/agda/agda/issues/2181 (visited on 03/31/2023).

[3] Andreas Abel. *Printer prefers (longer) qualified over (shorter) unqualified name*. 2019. URL: https://github.com/agda/agda/issues/3240 (visited on 03/31/2023).

[4] 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. *Record types*. 2023. URL: https://agda.readthedocs.io/en/latest/language/record-types.html (visited on 05/04/2023).

[6] Brian Anderson. *Generics and Compile-Time in Rust*. PingCAP. June 15, 2020. URL: https://www.pingcap.com/blog/generics-and-compile-time-in-rust/ (visited on 02/09/2023).

[7] Clemens Ballarin. "Interpretation of Locales in Isabelle: Theories and Proof Contexts". In: *Mathematical Knowledge Management, 5th International Conference, MKM 2006, Wokingham, UK, August 11-12, 2006, Proceedings*. Ed. by Jonathan M. Borwein and William M. Farmer. Vol. 4108. Lecture Notes in Computer Science. Springer, 2006, pp. 31–43. ISBN: 3-540-37104-4. DOI: 10.1007/11812289_4. URL: http://dx.doi.org/10.1007/11812289_4.

[8] Clemens Ballarin. "Tutorial to locales and locale interpretation". In: *Contribuciones científicas en honor de Mirian Andrés Gómez*. Universidad de La Rioja. 2010, pp. 123–140.

[9] Henk Barendregt and Herman 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).

[10] N. G. de Bruijn. "Telescopic mappings in typed lambda calculus". In: *Information and Computation* 91.2 (Apr. 1, 1991), pp. 189–204. ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90066-B. URL: https://www.sciencedirect.com/science/article/pii/089054019190066B (visited on 02/03/2023).

[11] Ivar de Bruin. "Improving Agda's module system". Msc thesis. Delft, The Netherlands: TU Delft, 2023. URL: http://resolver.tudelft.nl/uuid:98b8fbf5-33f0-4470-88b0-39a9d526b115.

[12] Jacques Carette. *Switch to a structured signature?* 2022. URL: https://github.com/agda/agda/issues/4331 (visited on 03/31/2023).

[13] Jacques Carette and Jason Hu. "Formalizing Category Theory in Agda". In: (2021). DOI: 10.1145/3437992.3439922.

[14] Liang-Ting Chen. *Qualified names are printed if introduced by 'open M ...'* 2022. URL: https://github.com/agda/agda/issues/5632 (visited on 03/31/2023).

[15] Jacek Chrzaszcz. "Implementing Modules in the Coq System". In: *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*. Ed. by David A. Basin and Burkhart Wolff. Vol. 2758. Lecture Notes in Computer Science. Springer, 2003, pp. 270–286. ISBN: 3-540-40664-6. URL: http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2758&spage=270.

[16] Nils Anders Danielsson. *Record constructors sometimes in record modules, sometimes not*. 2019. URL: https://github.com/agda/agda/issues/4189 (visited on 03/31/2023).

[17] Nils Anders Danielsson. *Shadowing parameters are sometimes renamed*. 2020. URL: https://github.com/agda/agda/issues/2018 (visited on 03/31/2023).

[18] Rust developers. *Monomorphization*. Rust Compiler Development Guide. Nov. 30, 2023. URL: https://rustc-dev-guide.rust-lang.org/backend/monomorph.html (visited on 11/30/2023).

[19] Facundo Domínguez. *timestats*. Version 0.1.0. July 13, 2022. URL: https://hackage.haskell.org/package/timestats-0.1.0.

[20] Paolo G. Giarrusso. *Regression with open public*. 2018. URL: https://github.com/agda/agda/issues/1985 (visited on 03/31/2023).

[21] Google Code Exporter. *Change the semantics of open public in parameterised module*. 2018. URL: https://github.com/agda/agda/issues/892 (visited on 03/31/2023).

[22] Google Code Exporter. *Copatterns do not work in parametrized modules*. 2015. URL: https://github.com/agda/agda/issues/940 (visited on 03/31/2023).

[23] Google Code Exporter. *Printing of infix/mixfix operators defined in parametrized modules*. 2022. URL: https://github.com/agda/agda/issues/632 (visited on 03/31/2023).

[24] Google Code Exporter. *Undeclared name accepted in fixity declaration*. 2015. URL: https://github.com/agda/agda/issues/329 (visited on 03/31/2023).

[25] Martin Hofmann. "Syntax and Semantics of Dependent Types". In: *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and P. Dybjer. Cambridge University Press, 1997, pp. 79–130. DOI: 10.1017/CBO9780511526619.004. URL: https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/syntaxandsemanticsof-dependenttypes.pdf.

[26] Dieter Hutter. "Management of Change in Structured Verification". In: *Fifteenth IEEE International Conference on Automated Software Engineering*. ASE. IEEE, 2000, p. 23. DOI: 10.1109/ASE.2000.873647.

[27] Arjen Jonathan. *Unnecessary conversion checking due to parameterized module slows type-checking (a lot)*. 2022. URL: https://github.com/agda/agda/issues/4517 (visited on 03/31/2023).

[28] Wolfram Kahl. *Regression: Module parameters lost*. 2015. URL: https://github.com/agda/agda/issues/1701 (visited on 03/31/2023).

[29] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 1st Edition. San Francisco: No Starch Press, June 26, 2018. 552 pp. ISBN: 978-1-59327-828-1.

[30] "The λ-Calculus". In: *Abstract Computing Machines: A Lambda Calculus Perspective*. Ed. by W. Kluge et al. Texts in Theoretical Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 51–88. ISBN: 978-3-540-27359-2. DOI: 10.1007/3-540-27359-X_4. URL: https://doi.org/10.1007/3-540-27359-X_4 (visited on 02/03/2023).

[31] Xavier Leroy et al. *CompCert C verified compiler*. 2022. URL: https://compcert.org/compcert-C.html (visited on 04/10/2023).

[32] David MacQueen. "Modules for Standard ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036. URL: https://doi.org/10.1145/800055.802036.

[33] David MacQueen, Robert Harper, and John Reppy. "The History of Standard ML". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386336. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3386336.

[34] Orestis Melkonian. *Unsafe(?) irrelevant projections by 'open'ing*. 2023. URL: https://github.com/agda/agda/issues/6359 (visited on 03/31/2023).

[35] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Göteborg, Sweden: Chalmers University of Technology and Göteborg University, 2007. 166 pp. URL: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[36] Egbert Rijke et al. *Univalent mathematics in Agda*. URL: https://github.com/UniMath/agda-unimath/.

[37] ANDREAS ROSSBERG. "1ML – Core and modules united". In: *Journal of Functional Programming* 28 (2018), e22. DOI: 10.1017/S0956796818000205.

[38] Yuhi Sato and Yukiyoshi 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*. GPCE '21: Concepts and Experiences. Chicago IL USA: ACM, Oct. 17, 2021, pp. 184–196. ISBN: 978-1-4503-9112-2. DOI: 10.1145/3486609.3487209. URL: https://dl.acm.org/doi/10.1145/3486609.3487209 (visited on 02/07/2023).

[39] Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. "Module generation without regret". In: *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 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. URL: https://doi.org/10.1145/3372884.3373160 (visited on 02/07/2023).

[40] The Coq Development Team. *The Coq Proof Assistant, version 8.17.1*. 2023.

[41] The Lean Development Team. *The Lean 4 Proof Assistant*. 2023.