Expanding Coq with Type Aware Code Completion

Hjalte Dalland *hjda@itu.dk*, Jakob Israelsen *jais@itu.dk*, Simon Kristensen *sigk@itu.dk*
Course code: `S23KISPECI1SE349`, IT University of Copenhagen

ABSTRACT

Code completion is readily available for many general-purpose programming languages, but this is not the case for proof assistants, including the popular proof assistant Coq.

In this paper, we present, compare and evaluate algorithms for providing code completion for the Coq language server, specifically aimed towards improving the VsCoq extension for Visual Studio Code. We then provide a comparison of these algorithms based on Robbes and Lanza's benchmarking procedure. This comparison finds that Type Structure with Selective Split Unification performs the best among our algorithms. When no prefix is used for fuzzy matching in filtering, the algorithm provides the correct completion as one of the top ten results in 53.3% of cases. This result comes with the caveat of the strategies not working well with SSReflect and other external libraries.

June 1, 2023

I. INTRODUCTION

As a codebase grows, so does the complexity of adding functionality. Each new class or function adds new options for the developer to write code, but adds additional work, requiring the user to evaluate what options are applicable. This is an especially difficult problem for people who are new to a large codebase, who will need to spend time finding out which functions and classes exist, and what they do.

Code completion is done to reduce the cognitive burden on the user by presenting a list of likely options, meaning the user does not have to remember everything. These options are typically sorted, so as to present what is most likely to be the relevant completions in a given context.

Users of general-purpose programming languages have long enjoyed productivity-enhancing benefits from code completion when using tools such as IntelliSense[1] in Visual Studio or Content Assist[2] in the Eclipse Editor. In contrast, proof assistants have been lacking this code completion, leaving users with rudimentary code completion oblivious to language semantics. Such proof assistants include Isabelle, Agda and notably Coq.

Coq is one of the most popular proof assistants and allows for writing verified programs and formalising proofs. It has been used to write CompCert, a verified C-compiler and formalise the proof for the 4-colour theorem. However, despite Coq being one of the older and more popular proof assistants, no editor provides language-aware code completion for it. Some editors provide language-oblivious code completion, which for example suggests words from the current document, with no regard for syntax or validity. Only using existing words can easily create syntactic errors, as they are not aware of the syntax of Coq and therefore might suggest something invalid.

Prior work for providing code completion in Coq is essentially non-existent and consists of standard editor tools for providing completions of words from the current file [3]. One proof assistant, *Lean*, provides some code completion [4], by suggesting all valid constructs, including those outside the current file, but only sorting the results by lexicographic ordering.

Our work coincides with a rewrite of VsCoq [5], an extension to Visual Studio Code for working with Coq. The rewrite changes the extension to use Language Server Protocol (LSP) and to directly depend on Coq's source code. This tighter integration allows for potential code completion to make use of Coq's source code, essentially using it like a library. By having access to Coq's internal functions, the language server should also be able to extract a greater wealth of information. This information will allow code completion to provide more relevant completion results.
We thus present the following problem statement:

> *How can a system for providing useful code completion be created, in order to improve the process of writing proofs in Coq?*

To aid in answering the problem statement, we ask the following research questions:

- What does it mean for a completion to be useful?
- How can we benchmark and compare the usefulness of code completion algorithms?
- What strategies and heuristics are effective at finding useful completions?

We hypothesise it is possible to provide relevant code completions efficiently by using the type-information available in theorems and proof goals. Furthermore, we theorise that by using Coq's unification algorithm, we can further improve the usefulness of completions by checking if theorems can be applied. To simplify terminology, the word theorem is used as a stand-in for theorem, lemma, and hypothesis, due to their use being identical for completion purposes.

Creating code completion for the entirety of Coq is outside the scope of this paper. Coq has multiple modes of interaction, each requiring different types of code completion. These modes include defining new tactics, implementing functions, specifying theorem statements and constructing their proofs. An example of the syntax used in each mode can be seen in figure 1.

A separate method of providing completion would be necessary for each interaction mode, as they have distinct syntax.

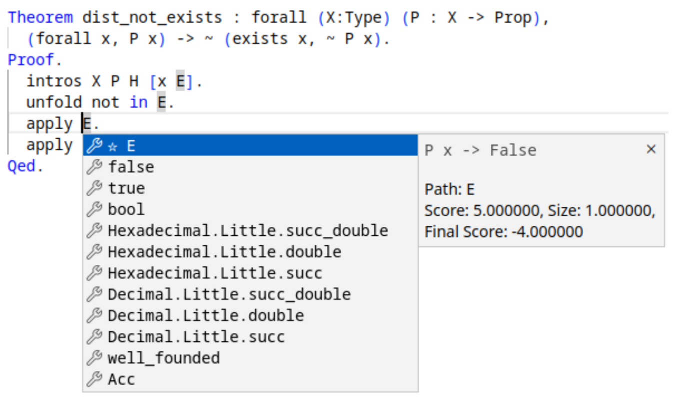
```
(* Defining a tactic *)
Ltac mega_destruct :=
  match goal with
  | [H: _ /\ _ |- _] => destruct H
  | [H: _ \/ _ |- _] => destruct H
  | [H: exists _ : _, _ |- _] => destruct H
  end.

(* Implemeting the rev function *)
Fixpoint rev {A:Type} (l:list A) : list A :=
  match l with
  | nil      => nil
  | cons h t => app (rev t) (cons h nil)
  end.

(* Specifying theorem statement for rev_rev *)
Theorem rev_rev :
    forall A (l l' : list A),
  l = rev l' ->
  l' = rev l.

(* Constructing proof for rev_rev *)
Proof.
  intros  A l l' H.
  symmetry.
  rewrite H.
  apply rev_involutive.
Qed.
```

Fig. 1 – The four modes of interaction in Coq.



```
Theorem dist_not_exists : forall (X:Type) (P : X -> Prop),
  (forall x, P x) -> ~ (exists x, ~ P x).
Proof.
  intros X P H [x E].
  unfold not in E.
  apply E.
  apply
Qed.
```

```
☆ E
false
true
bool
Hexadecimal.Little.succ_double
Hexadecimal.Little.double
Hexadecimal.Little.succ
Decimal.Little.succ_double
Decimal.Little.double
Decimal.Little.succ
well_founded
Acc
```

```
P x -> False

Path: E
Score: 5.000000, Size: 1.000000,
Final Score: -4.000000
```

Fig. 2 – Screenshot of code completion in Visual Studio Code. E is the top result.

In this paper we only implement and analyse code completion for one of the interaction modes, the construction of proofs. Furthermore, we focus on supporting backward reasoning through the tactics `apply` and `rewrite`. Coq also supports forward reasoning through the tactics `apply in` and `rewrite in`, but the algorithms in this paper are not optimised for these cases.

An example of using VsCoq to get code completions can be seen in figure 2, where one of the algorithms developed in this paper is used to correctly guess the next step.

To answer our research questions, we first describe Coq and its interaction models. This is followed by a description of Visual Studio Code, the Language Server Protocol, and VsCoq. To allow reproduction of our results we outline our methodology, describe our data sets, and provide source code used in running the experiments. We give a definition of usefulness for code completion and to quantify this usefulness, we use a method presented by Robbes and Lanza in their paper *Improving code completion with program history* [6]. We then present a variety of code completion algorithms, explaining the ideas behind them and the results yielded by running them on the data set. Finally, we discuss our work, including alternative criteria for evaluating code completion, and potential problems with our work, before concluding our report and mentioning ideas for further work.

A. *Acknowledgements*

We want to thank Maxime Dénès, Enrico Tassi and Romain Tetley for answering our many questions about Coq and helping us with implementing features in VsCoq.

We also want to thank Jesper Bengtson for his supervision and guidance.

II. Background

In this section, we describe the fundamental technologies for this paper. We describe the proof assistant Coq, giving an overview of how it works, as well as describe Visual Studio Code (VS Code), the Language Server Protocol (LSP), and the concept of code completion.

A. *Coq*

Coq is a proof assistant, initially developed in 1984 and implemented in Caml-Light, before being converted to OCaml [7]. At present, the code base contains more than 300,000 lines of OCaml code [8]. Coq has been used to verify famous proofs such as the Four Colour theorem [9], a proof too large for humans to feasibly verify by hand. Other famous proofs verified in Coq include the Odd-Order theorem [10] and Abel-Ruffini theorem [11]. In addition to being used for mathematical proofs, Coq has been used to implement the trusted and verified compiler CompCert for a subset of C [12].

Verasco, a verified static analyser which establishes the absence of run-time errors in C code, has also been created for this subset of C compiled by CompCert [13]. The correctness of complicated data structures have also been formally verified using Coq such as a persistent union-find data structure [14]. Coq has also been extended to use external automatic theorem prover tools by checking the correctness of the output of the tool, such as with A3PAT [15].

```
Theorem rev_rev :
    forall A (l l' : list A),
  l = rev l' ->
  l' = rev l.
Proof.
  intros  A l l' H.
  symmetry.
  rewrite H.
  apply rev_involutive.
Qed.
```

Fig. 3 – An example of a Theorem in Coq. The theorem `rev_involutive` is used as an argument to `apply`.

Users have several modes of interaction with Coq. They can define functions using the built-in functional language for Coq. The users may then construct propositions about the functionality of these programs, that can be proven using theorems. An example of such a proof can be seen in figure 3, where functionality is proven for the reverse list function `rev`. Theorems are proven using tactics which modify the goal and hypotheses until every goal of the proof is solved. The final mode of interaction is defining custom tactics to more easily prove goals.

After a user has finished a proof, the kernel of Coq can check the validity of the entire proof. Coq's kernel is a trusted implementation of the Calculus of Inductive Constructions, which is small enough to be checked for correctness by humans [16]. To ensure correctness, Coq also has an extensive unit test suite which further increases confidence. Tactics will be transformed to terms Coq's kernel can understand, and if these terms are invalid, the proof will fail at verification. Therefore, the implementation of tactics do not have to be trusted, as any fault is caught by the kernel. This allows Coq to have a large library of complex external tactics that speed up the development of proofs, while only having to trust a very small codebase in the form of the kernel.

1) Workflow: The workflow of Coq is different from most other programming languages. Where most programming languages evaluate entire files at a time, in Coq the evaluation state inside a file is generally controlled by the user. Coq code is made up of *sentences*, the smallest fragments of code that can be interpreted, generally terminated by periods. The evaluation state is modified by sending commands to step forwards and backwards in increments of a sentence, usually with a keyboard shortcut.

While writing a proof in Coq, the evaluation state is clearly visible, as the current goal and the available hypotheses are shown in a proof view window. The goal and hypotheses are derived from the *proof state*, which is the part of the evaluation state containing the state of the proof [17]. The proof state contains all remaining goals, the available hypotheses and introduced variables for each goal.

Upon entering proof mode by using the `Proof` keyword, a proof state is created, which contains the theorem statement as the only goal. The user can then create a hypothesis in the proof state by the use of tactics such as `intros`, which can introduce variables from `forall` and move premises from the goal into hypotheses. New subgoals may also be introduced, where each subgoal uses the proof state at the point they were created. One tactic that can introduce new subgoals is `split`. Used on a conjunction $P \wedge Q$, `split` creates the new subgoals P and Q.

The proof state stands in contrast to the *global environment*, which contains all previously defined variables, theorems, and functions [18]. All definitions in the proof state and global environment can, when applicable, be used in conjunction with tactics to complete the proof.

2) Types in Coq: In Coq, types are implemented as the recursively defined OCaml type `Constr` [19]. Consequently, any instance of a Coq type can be interpreted as a tree of types with each node being an instance of `Constr`. `Constr` contains several pieces of information about the type, including its *kind*. This kind describes how the `Constr` node relates to its child nodes in the type tree. The kind of `Constr` can be extracted using the function `kind` which returns a `kind_of_term`, an algebraic data type.

This paper is mainly concerned with the following subset of constructors in `kind_of_term`:

- Constants are fully qualified types, like `nat`. The constant kinds include inductive types `Ind`, constructors to inductive types `Construct`, variables `Var`, and existential variables `Evar`.
- `App of Constr * Constr array` is a type that takes further types as arguments, and is guaranteed to have at least one type argument contained in the array [19]. An example of this is `list`, which is type polymorphic and as such takes the type of elements as an argument in the array. The array of types is where the tree branches into multiple subtrees.
- `Sort of Sorts` introduces a type of types and is created when using generic types, such as a type argument.
- `Prod of Name * Type * Type` creates a named variable with a type and continues the tree in the third argument. The second argument is usually a constant or an application of constants. Inside the rest of the tree, the introduced variable can be referenced by De Bruijn Index [20].
- `Rel of Int` refers to an earlier introduced bound variable by way of De Bruijn index. The variables are introduced by `Prod`. In our internal representation for completion, we use the index to bind the type of Sorts. An example of this would be a function taking two arguments of the same type and checking their equality. The types of the arguments would be `Rel` referring to a `Sort` type argument. By binding the type to the sort at the time of ranking, the likelihood of type constraints being met is higher. This allows for better ranking of generic types, as theorems that have no chance of matching the goal are prioritised lower.

```
Theorem rev_rev : forall A (l l' : list A),
  l = rev l' ->
  l' = rev l.
Proof.

Prod (A, Sort Type,
  Prod (l, App(list, {| Rel 1 |}),
    Prod (l', App(list, {| Rel 2 |}),
      Prod (->,
        App (@eq, {|
          App (list, {| Rel 3, Rel 2 |}),
          App (@rev, {| Rel 3, Rel 1 |})
        |}),
        App (@eq, {|
          App (list, {| Rel 4, Rel 2 |}),
          App (@rev, {| Rel 4, Rel 3 |})
        |})
      )
    )
  )
)
```

Fig. 4 – The goal's type before `intros` in `rev_rev` shown as the `kind_of_term` constructors. The indentation level is the level in the tree, `Rel` values are De Bruijn indices

These type constructors, along with the rest of `kind_of_term`, are combined to construct all types in Coq.

Since the application of theorems requires the type of the theorem to match the type of the goal, understanding the structure of the types is essential to provide useful code completions in Coq. An example of a type in terms of its `kind_of_term` constructors can be seen in figure 4, which displays the type of `rev_rev` from figure 3.

3) Unification: Unification is the process of reconciling different types in a programming language or type system. Higher-order unification extends this concept to handle types that involve functions or higher-order constructs, allowing for the unification of more complex type expressions. Using higher-order unification is potentially more expensive in terms of computations, causing it to be slower.

In Coq, unification forms the basis for the `apply` tactic. `apply` will attempt to unify the type of the theorem with the type of the goal. If the unification is successful, the theorem can be used to prove the goal. If it fails, `apply` will also try to *conditionally* apply the theorem, meaning it will attempt to unify the conclusion of the theorem with the goal. If this succeeds, the goal is proven, but the premises of the theorem are then added to the list of goals. For example, if you are trying to prove the proposition C, and there is no theorem proving C, but instead the theorem $A \rightarrow B \rightarrow C$ exists. Then, this theorem can then be conditionally applied, leaving the task of proving A and B.

Unification is also used for forward reasoning through the `apply in` tactic. This tactic still takes a theorem to be applied and a hypothesis to apply the theorem in. The `apply in` tactic will try to unify the first premise of the theorem with the given hypothesis and, if successful, replace the hypothesis with the rest of the theorem. This can be used to change the hypothesis until it can be used to solve the current goal, in contrast to backwards reasoning where the goal is changed to match the hypotheses.

B. Visual Studio Code

Visual Studio Code (VS Code) is a freeware and partially open-source code editor released in 2015 by Microsoft. It is the most popular editor among software developers [21], and it ships with advanced editor features for web technologies like HTML, JavaScript/TypeScript and JSON.

Through extensions, it also supports editor features for most commonly used programming languages, such as C#, Java, Python, C, and many more. These extensions are part of a vast ecosystem, which gives VS Code access to advanced editor features in most programming languages or data formats a programmer might encounter. This is further expanded by Microsoft's efforts to develop open protocols for interacting with debuggers and language infrastructure, such as the Language Server Protocol.

C. Language Server Protocol

Language Server Protocol (LSP) [22] is a protocol specification created by Microsoft, made to facilitate communication between Language Servers and editors. A language server is a program that provides programming language-specific services such lookup of symbols, highlighting of files, or code completion. As a result, it is usually necessary to implement many of the features that exist in compilers to extract information about functions, types and references to variables.

Prior to the introduction of LSP, if a language server wanted to provide these functionalities in a new editor, they would essentially have to duplicate their efforts, usually in the form of a plugin. LSP was created to address this issue and act as a standard interface between language servers and editors. This means that if a language server adheres to the LSP, it can be used for many different editors, and similarly, an editor can support any language that provides a language server communicating via LSP. An illustration of this can be seen in figure 5.

In this figure, every arrow represents an implementation that needs to be written and maintained. On the left, there needs to be an implementation for each combination of language and editor, while on the right, every language and editor can make do with a single implementation that adheres to LSP.

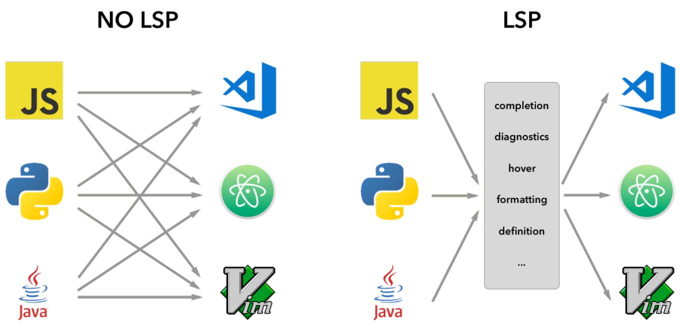


Fig. 5 – An illustration of the problem LSP solves. Taken from VS Code's documentation [23].

The code written for this paper is part of the new Coq language server, which as of writing is only used by the VsCoq extension for VS Code. Since the new Language Server follows the LSP specification, any editor supporting LSP, such as Emacs or Vim, would be able to use it. Only the Coq-specific parts that are not part of the LSP specification, such as the proof view, Commands and manual interpretation stepping, would require an editor-specific implementation to add support for a new editor.

D. VsCoq

VsCoq is an extension for VS Code to support Coq and was originally released in 2016 [24], later being handed off to the Coq community [25]. Starting in late 2022, work began on a rewrite by Dénès, Tassi, and Tetley [5], creating VsCoq version 2.0. The focus of VsCoq 2.0 was on removing a dependency on the legacy language server *CoqIDETop*.

CoqIDETop was made to implement the Coq XML protocol for CoqIDE [26]. However, it also became a source of bugs. The Coq XML protocol is stateful, and this state management caused problems during development. CoqIDETop also requires locking the interpreted state, another aspect that led to development being difficult. It was therefore decided to use LSP in VsCoq 2.0, as this protocol is implemented VS Code, which meant implementing a language server, instead of a client that talked through CoqIDETop.

VsCoq 2.0 interacts directly with Coq, using Coq's implementation of parsing, types, evaluation state and so on. As it works directly with Coq, some difficulties arise due to Coq being a large code base with over 30 years of development by many different developers. The code base's quality, readability, and level of documentation varies greatly from file to file. Some files were never meant to be read by anyone outside the Coq project and contain functions with warnings about safety [27]. Therefore, the largest hurdle in creating new features is often finding the correct function to call, along with what arguments it should be called with to achieve the desired results.

At the time of writing, the rewrite is close to reaching feature parity with the old CoqTop version, and has the following features implemented:

- A proof view showing the current proof state with hypotheses and the current goals.
- Hovering over variables with the cursor shows their type.
- Search view with a separate tab showing the results of running Search commands.

In addition to these features implemented by Dénès, Tassi, and Tetley, we implemented very rudimentary code completion, which just uses the results of a Search command to return every theorem that exists in the current context.

E. Code Completion

Code completion is a common feature of modern editors that can suggest code to be written at the user's cursor. Code completion algorithms vary in complexity and effectiveness, from suggesting existing words in the current document, to analysing types of variables. For example, in an object-oriented language, a code completion algorithm may be able to suggest the methods of a class by inspecting the type of a variable and presenting the relevant class methods.

Commonly, completions are made in three steps:

1) *Discovery* is finding all relevant completions for the current context, which is provided by the language server. For example, this could consist of suggesting all possible methods belonging to a given class.
2) *Sorting* consists of sorting the completions based on relevance to the user. There is no concrete measure of relevance, and code completion systems will handle this in different ways. Following our previous example, if the current context expects a string to be returned, the algorithm might put methods returning strings at the top of the list.
3) *Filtering* is the process of filtering these completions, based on what the user has written so far. This section of already written text is the *prefix*, since it is the prefix of what the user may end up writing. This process of filtering is usually done by the editor and commonly consists of filtering out completions that do not start with the prefix. Some editors, such as VS Code, go further and provide *fuzzy matching*, which also matches on infix, suffix, or sub-sequences.

The implementation of these steps can differ from editor to editor. Some language servers do not implement filtering, because it is common for editors to have it implemented instead. Leaving filtering to the editor also has the advantage of the language server only having to create a list of sorted results once for the current context. The editor can then repeat filtering after each letter entered by the user, without having to wait for the language server. This creates the user experience of immediate feedback after getting the first results, as the relatively slow call to the language server is only made once.

Writing code completion algorithms for proof assistants will involve differences compared to writing for general-purpose programming languages. Some of the differences include:

- Some proof assistants, such as Coq and Isabelle [28], contains several modes of interaction. This means there are multiple contexts, each of which will require completely different code completion algorithms.

- When writing code in a general-purpose programming language, the intent of the user will be unknowable to the language server, as it will have no way of semantically knowing what the user is trying to accomplish with their code, other than getting it to compile by matching types. When writing proofs, the intent is always to prove the theorem statement. This means an advanced code completion algorithm for writing proofs could attempt a limited form of proof synthesis.
- Given the complexity of types in proof assistants, code completion based on types is more resource intensive, due to having more data to check. Therefore, using heuristics to choose candidates might be necessary, rather than running more complex and slow code completion algorithms.

In short, there are several aspects that make development of code completion for proof assistants different to code completion for general-purpose programming languages.

III. Methodology

In this section, we discuss how we evaluate and benchmarked the usefulness of completion algorithms. We then describe our experimental setup, and how we compare algorithms to each other.

A GitHub repository of the code used for this project can be found with the following link, `https://github.com/Jakobis/vscoqComparison`. A zipped copy of the code can also be found in the appendix.

A. Algorithmic Structure of Code Completion

To make clear exactly what is being evaluated, we first define the structure of the code completion algorithms. Every algorithm described in this paper uses the same processes for the steps of discovery and filtering described in section II-E. Discovery is handled through the internal function that is also used by the Search command in Coq. This function returns an arbitrarily ordered list of all theorems available in the current context. Filtering is handled by VS Code, which runs the fuzzy matching algorithm based on the prefix.

The only difference in how the algorithms are implemented is the sorting step. The algorithms will sort the theorems using the proof state's current goal as a reference for what the user is trying to achieve. By comparing the theorems and the current goal, an ordering based on relevance to the goal is created. The ordering is then used to tell VS Code how it should sort the results when presenting them to the user. All the algorithms take the same theorems as input and output the same completions, the only difference being the ordering.

B. Evaluation Criteria

In order to evaluate the usefulness of different completion algorithms, we must first define what usefulness is. Code completion is ultimately meant to be used by users, and the best evaluation method would likely be user studies. However, such studies are time-consuming, expensive, and make rapid iteration harder to accomplish. They are therefore outside the scope of this paper.

Prefix length (i)	0	1	2	3
Prefix	-	B	Ba	Baz
List	Foo, Bar, Baz	**B**ar, **B**az	**Ba**r, **Ba**z	**Baz**
Correct answer position	3	2	2	1
$G(i)$	1/3	1/2	1/2	1

TABLE 1 – A table showing an example with gradings $G(i)$ for different values of i.

Instead, we define usefulness as the ability to provide the completion the user wants, as early in the list of completions as possible. To this end, we argue that what the user wants can be approximated by looking at existing proofs. Here, we assume that what the user wanted to write, is what they ultimately wrote. Thus, by looking at existing proofs, this definition of usefulness is quantifiable.

For every applied theorem in these existing proofs, a call for completions can be made, and the theorem written by the user can be used as the correct answer. The position of the correct answer can then be looked up in the list of completion results.

To evaluate and compare the usefulness of our algorithms, we use a formula based on the work of Robbes and Lanza [6] which uses a variation of Mean Reciprocal Ranking (MRR) [29]. MRR is a formula for ranking processes that produce a list of results, where the desired outcome is that a given result appears as early in the list as possible. This makes MRR useful to score code completion algorithms. Robbes and Lanza's variation consists of looking at prefixes of the correct answer, using this prefix to produce a list of results and calculating the MRR for each such list of completions. A formula is then used to calculate the combined score based on the scores for each prefix length, with a bias for doing well on smaller prefixes. This gives weight to the filtering of results done by fuzzy matching as explained in section II-E.

$$G(i) = \frac{\sum_{j=1}^{t} \frac{\mathit{results}(i,j)}{j}}{\mathit{attempts}(i)} \tag{1}$$

For a specific prefix length i, the grade $G(i)$ is the sum of occurrences of the correct completion item within each top t positions in the completion list weighted by position j. The value $\mathit{results}(i,j)$ is the number of occurrences of the correct answer for prefix i at position j in the list, while t corresponds to the lowest list position to be considered. The sum of occurrences is divided by $\mathit{attempts}(i)$, the total amount of completion attempts for prefix i.

In this paper, t is chosen to be 10, meaning that if the correct answer is not among the first ten in the list of results, it is equivalent to not being present at all.

An example of the usage of the formula can be seen in table 1. Here, our example algorithm outputs the list *Foo, Bar, Baz*, and the desired result is *Baz*. The filtering process consists of only including completions that start with the prefix. This example contains only 1 attempt per prefix, and every score is therefore divided by 1.

To give a total score across all prefix lengths, we use the following formula to give a weighted average of the values of $G(i)$:

$$S = \frac{\sum_{i=m}^{p} \frac{G(i)}{i+1}}{\sum_{k=m}^{p} \frac{1}{k+1}} \times 100 \tag{2}$$

The score S for an algorithm is a weighted sum of the grades $G(i)$ for prefix lengths m through p, all divided by the hypothetical perfect score for the prefix length, and scaled by 100 for legibility. The intent is to give a score to each algorithm, averaging the performance across different lengths of prefixes, but giving more weight to results using shorter prefixes. This means that algorithms that are better at giving good results with small prefixes, will do better than those who need larger prefixes.

This scoring formula is also based on the work of Robbes and Lanza, but their version only takes into account prefixes of length 2 to 8, using the values $m = 2, p = 8$. However, for the algorithms presented in this paper, it is relevant to look at prefixes of length 0 and 1 as well, leading to the values $m = 0, p = 8$. This is because of the unique circumstances around making a proof in Coq. The change of m to 0 means that the scores cannot be directly compared to Robbes and Lanza's scores.

In a general-purpose programming language, the language server can never know what the intent of the user is. There are simply too many options for what a user could semantically be trying to achieve. On the other hand, if a user is trying to write a proof in a proof assistant, there is essentially only one possible goal: to prove the theorem statement. Given the expressive type system of Coq and that the user's intent is encoded in the proof's goal, we believe our algorithms have relatively high chances of predicting the correct answer, even with 0 known characters.

To give an example of the calculation, consider the table 1. A value of 3 is used for p to simplify the example.

The sum of grades is calculated to be 1 in equation 3.

$$\begin{aligned} \sum_{i=0}^{3} \frac{G(i)}{i+1} &= \frac{1/3}{1} + \frac{1/2}{2} + \frac{1/2}{3} + \frac{1/1}{4} \\ &= \frac{24}{24} \\ &= 1 \end{aligned} \tag{3}$$

Then, in order to find what the perfect grades would be, equation 4 calculates this sum of prefect grades.

$$\begin{aligned} \sum_{k=0}^{3} \frac{1}{k+1} &= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \\ &= 25/12 \\ &\approx 2.08 \end{aligned} \tag{4}$$

To calculate the final score, these values are substituted in to the score formula, as seen in equation 5. The score is calculated to be 48.

$$\begin{aligned} S &= \frac{\sum_{i=0}^{3} \frac{G(i)}{i+1}}{\sum_{k=0}^{3} \frac{1}{k+1}} \\ &= \frac{1}{25/12} \times 100 \\ &= 48 \end{aligned} \tag{5}$$

With this formula, a score is given to every algorithm, allowing us to compare their respective usefulness. However, it is important to mention that the score is only really useful for comparison. It is not as simple as a score of 48 meaning the algorithm is right 48% of the time. Instead, the score is a weighted average of weighted averages.

For this reason, our algorithms will not only be compared based solely on their score. For each algorithm, we will also show a table with success rates based on the lengths of prefixes and the position of results. This will allow us to compare the algorithms in more depth, showing the influence of the length of prefix and the ranking of the correct answer. It also allows for a comparison of no prefix cases, which is of particular interest as it requires no input from the user.

C. Experimental Setup

The experimental procedure used for each algorithm is as follows: For every file, at every location where the tactics `apply` and `rewrite` are used, request for a list of completion items.

This list is then provided to the completion provider within VS Code, to filter for each prefix of the correct answer to generate a filtered completion item list. For each list, check the position of the correct answer in the filtered list. The position is used to perform grading and scoring based on the criteria from section III-B, grouped by completion algorithm and algorithm parameters across all files in the data set.

To more closely replicate the user experience of a person using VsCoq, the fuzzy matching in VS Code was used as the filtering step. This means that our results should accurately represent usage in VsCoq, but may differ if the Language Server is used with another editor which provides a different filtering algorithm or no such algorithm at all. Consequently, the results are not entirely governed by our sorting heuristics because the filtering influences the results.

D. Data Set

The algorithms were tested on a data set of three prior Coq projects, that were chosen based on a set of requirements. Due to a bug with importing external libraries, it was a requirement that the projects could not use any such imports. Therefore, we excluded any files or projects that required external dependencies to run.

Many projects in Coq are written with *SSReflect*, an alternative syntax that is fairly popular. However, since SSReflect behaves significantly differently from Coq, it was deemed to be outside the scope of this paper to support it. Therefore, there are no projects using SSReflect in the data set. This exclusion means that the resulting data is not representative of everyone's experience when using VsCoq. The problems this poses is further expanded on in the discussion section V-D.

The first project [30] was a student's solutions to the exercises within Software Foundations: Logical Foundations [31], up to the section on Simple Imperative Programs. We chose this project as it can represent the usage patterns of a person new to Coq. This is due to the fact that the book and its exercises are made as an introduction to Coq and proof assistants.

Coq 100 Theorems [32] is the second project and is a collection of the top 100 most famous proofs, as decided by Kahl [33], formalised in Coq. Most of these use only theorems from the Coq standard library, which fits well with the requirement of no external dependencies, while still offering a variety of projects written by different users.

CoqGym[34] is the last project and is itself a collection of projects intended to provide a large data set and machine learning environment and has been used for benchmarking proof synthesis. Because CoqGym was meant to be used a machine learning environment, it made the setup of its various sub-projects easier. This made it easier to include a lot of projects without spending a lot of time on getting each project to work.

Both Coq 100 Theorems and CoqGym consist of multiple sub-projects, where some of these sub-projects did not meet our requirements and were therefore excluded from the data set.

IV. Code Completion Algorithms

In order to provide good code completions, an algorithm for sorting any provided completions is needed. This section describes the different algorithms we implemented and the results of their benchmarks. For each algorithm, we provide an overview of the idea behind it, a table showing the results of the algorithm, and an analysis of the results.

To better understand these tables, we here provide an explanation of table 2, which illustrates our results for the Baseline algorithm.

Each column corresponds to the length of the prefix used for obtaining completions. Prefix length 0 indicates that the completion was obtained without any knowledge of what the correct completion should be, while prefix lengths 1-8 indicate that the first 1-8 characters of the correct theorem are used to filter the completion items.

For each row, the label indicates where the correct item is placed in the completion item list. The results are listed in percentages, so for example, it can be read that the Baseline algorithm placed the correct theorem for no prefix as the first completion item 0.1% of the time, as the second completion 0.0% of the time, and as the 4th-10th completion 0.2% of the time.

Prefix	0	1	2	3	4	5	6	7	8
1st	0.1%	8.5%	29.8%	35.0%	38.6%	41.9%	49.1%	50.2%	53.8%
2nd	0.0%	4.1%	10.0%	10.4%	9.9%	11.5%	12.8%	15.1%	14.0%
3rd	0.0%	3.5%	5.2%	5.7%	6.3%	6.5%	5.9%	5.3%	6.2%
4-10th	0.2%	14.3%	17.4%	18.9%	17.9%	16.2%	10.8%	11.4%	10.2%
Pass	0.3%	30.4%	62.3%	70.0%	72.7%	76.1%	78.7%	82.0%	84.2%
Fail	99.7%	69.6%	37.7%	30.0%	27.3%	23.9%	21.3%	18.0%	15.8%

TABLE 2 – Test results from Baseline, showing for each prefix length how large a proportion of results appeared at each position in the completions.

There is a summary of how well each prefix did below the dividing line. The *pass* row is a sum of the values above and indicates that the correct answer was in the top ten results, meaning it would show up in the UI for the user to choose immediately. The *fail* row indicates how large a proportion of completion attempts either outright failed, or did not contain it in the first ten results, making it not immediately visible in the UI.

For example, in table 2, it can be seen that the Baseline algorithm failed 99.8% of completion attempts with no prefix information, but managed to get the correct theorem 78.7% of the time when a prefix of 6 characters was used to create completions.

A. Baseline (Score: 26.03)

The Baseline algorithm outputs the completions in random order by shuffling them. Shuffling is done to remove any impact the discovery step had on the order of theorems. As the name implies, it is intended to form a baseline for the sake of comparing the remaining algorithms. With Baseline, we can compare the algorithms to the results of random chance, giving a better understanding of how well they perform. It also give a better understanding of the impact of filtering through fuzzy matching in VS Code.

From the test results in table 2, it can be seen that randomising the completions produces almost no passing results when no prefix can be used, succeeding only 0.3% of the time, or once every ~330 attempts. Filtering on even a single character makes the Baseline pass 30.4% of the time, all the way up to 84.2% of the time for 8 characters. The score ends up being 26.03 points. This relatively high pass rate of 30.4% might be due to use of short names often used in Coq, such as H or H1 for hypotheses.

B. Type Intersection (Score: 49.68)

Type Intersection is a simple algorithm, based on the idea of flattening the type trees of types into sets, and then comparing these sets via set intersection. The flattening is done by extracting types from the type tree and inserting them into a set. These extracted types are *atomic*, as they are the smallest types that can be meaningfully compared on their own. This is in contrast to types that require the rest of the type tree to be meaningful. The set intersection between the atomic types of a theorem and the atomic types of the goal is used as the heuristic for ordering.

```
let typeIntersectionCompare goal a b =
    match (a ∩ goal), (b ∩ goal) with
    | a1, b1 when |a1| = |b1| ->
        compare |a \ goal| |b \ goal|
    | a1, b1 ->
        compare |a1| |b1|
```

Fig. 6 – Pseudo-code for the type intersection comparison function.

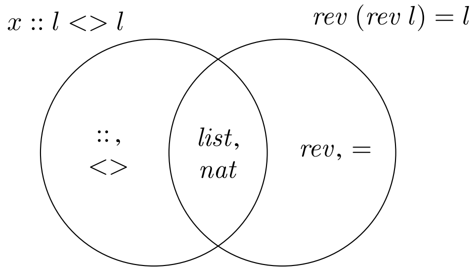


Fig. 7 – An illustration of the type intersection algorithm. Atomic types of the outer expressions are listed within the circles, the overlap indicating atomic types from both types.

To explain our heuristic, we define the following two terms:

1) *Type overlap* $(A \cap B)$ is the size of the intersection between two sets of types. The bigger the type overlap, the more similar the types are.
2) *Type difference* $(A \setminus B)$ is the number of atomic types present in a type, but not in the intersection with another type. The smaller the type difference, the more similar the types are.

Pseudocode describing the algorithm can be seen in figure 6.

Our main heuristic for Type Intersection consists of sorting by type overlap between theorems and the goal. Bigger intersections result in more overlap in types and therefore indicate higher similarity between the goal and the theorem. If two different theorems have the same type overlap, then sort them by the smallest type difference. If two different theorems have the same set of types, then Type Intersection has no way of distinguishing between them, and they would therefore be considered equal.

An example can be seen in figure 7 where the atomic types of the following propositions are compared:

$$\begin{gathered} forall\ x : nat,\ l : nat\ list,\ x :: l <> l \\ forall\ l : nat\ list,\ rev\ (rev\ l) = l \end{gathered} \tag{6}$$

The two propositions contain four atomic types each, and an overlap of two atomic types, *nat* and *list*.

Prefix	0	1	2	3	4	5	6	7	8
1st	21.5%	26.2%	50.4%	51.2%	55.9%	58.2%	62.7%	63.0%	66.0%
2nd	7.8%	27.2%	13.8%	14.4%	12.4%	11.9%	10.1%	11.6%	10.0%
3rd	5.6%	9.5%	7.0%	6.8%	5.7%	5.4%	4.5%	3.7%	3.9%
4-10th	18.0%	12.9%	9.9%	8.2%	7.4%	6.4%	6.7%	5.8%	5.0%
Pass	52.9%	75.8%	81.1%	80.6%	81.3%	82.0%	83.9%	84.2%	85.0%
Fail	47.1%	24.2%	18.9%	19.4%	18.7%	18.0%	16.1%	15.8%	15.0%

TABLE 3 – Test results from Type Intersection, showing for each prefix length how large a proportion of results appeared at each position in the completions.

The results for the Type Intersection algorithm can be seen in table 3. It is capable of outright guessing the correct theorem 21.5% of the time, placing the correct theorem in the top ten 52.9% of the time. This also translates into the results for known prefixes, where it passes 75.8% of the time for a single character prefix, a twofold increase in pass rate compared to Baseline. Type intersection performs even better than Baseline at getting the correct answer as the first result. As expected, this improvement drops off as more of the prefix is revealed, and the filtering in VS Code dominates the completion item list. The overall score turns out to be 49.68, a 23-point improvement over the Baseline algorithm.

C. Split Type Intersection (Score: 50.40)

This heuristic is based on combining simple type intersection with conditional application.

We define *splitting* a theorem type as follows. A theorem is made up of a series of premises and a conclusion, separated by implications. It can be split at any of these implications, removing all premises on the left side of the split, leaving only the right side in the form of a smaller theorem with fewer premises. We call this smaller theorem a *suffix* of the original theorem.

Split Type Intersection splits the theorems at every implication, finding the split where the suffix has the biggest type overlap with the goal. It also considers the overlap of the full theorem without any splits. If multiple suffixes have the same amount of overlapping types, the smallest suffix is chosen. The set of types for the chosen suffix is then used to sort the completions, in the exact same way as in Type Intersection.

An example can be seen in figure 4 where the theorem L of the form $A \to B \to C$ is compared to the goal $B \to C$. When comparing L to other theorems, the heuristic will try splitting the theorem into $A \to B \to C$, $B \to C$ and C. For each of these, the algorithm will then compute the intersection of types with the goal, with the intersections for the above example being 2, 2 and 1 respectively. In case of a tie, the split with the fewest types will be used, in this case, $B \to C$. This sub-type will then be used to compare L against other theorems.

Suffix	# of overlapping types
$A \to B \to C$	2
$B \to C$	2
C	1

TABLE 4 – A table displaying an example of how Split Type Intersection compares a goal $B \to C$ and with a theorem $A \to B \to C$.

Prefix	0	1	2	3	4	5	6	7	8
1st	22.8%	26.1%	50.2%	50.2%	55.9%	58.8%	63.8%	64.0%	67.4%
2nd	8.2%	27.7%	13.6%	14.7%	12.7%	11.9%	9.3%	11.2%	9.2%
3rd	6.4%	9.5%	7.0%	6.8%	5.9%	5.2%	4.8%	3.3%	3.4%
4-10th	18.2%	13.0%	10.5%	9.4%	7.0%	6.2%	6.3%	6.0%	5.3%
Pass	55.6%	76.3%	81.3%	81.1%	81.4%	82.2%	84.2%	84.5%	85.4%
Fail	44.4%	23.7%	18.7%	18.9%	18.6%	17.8%	15.8%	15.6%	14.6%

TABLE 5 – Test results from Split Type Intersection, showing for each prefix length how large a proportion of results appeared at each position in the completions.

As can be seen in table 5, this approach offers minor improvements over the standard Type Intersection algorithm. For the case with no prefix, a 1.3 percentage points improvement was observed for the first position, and it passes 55.6% of the time, a 2.7 percentage point improvement. This improvement can also more or less be seen for the rest of the prefix lengths, which also translates into the slightly higher overall score of 50.40 points, an improvement by just under a single point.

D. Type Structure (Score: 50.87)

The idea of the Type Structure strategy is similar to the previous Type Intersection in section IV-B. However, instead of only using the set of atomic types, the structure of the type tree is also used to compare types. This allows for using the information contained in Sorts to match a specific goal with more generic theorems. As the structure of the goal and theorem type might not be an exact match, each sub-tree of the theorem type tree is also compared with the goal, using the best match to sort the theorems. An example of the type tree of a theorem can be seen in figure 8.

Scoring is done by descending down the type tree of the goal and theorem simultaneously, trying to match the nodes between the two trees. If at any point the nodes in a branch cannot be matched, then the descent does not go further down that branch. Depending on the kind of match, different scores are assigned. Matching on two identical atomic types is given a higher score than matching a sort with a type. The rationale behind this is that it is more specific for the same atomic type to be mentioned in both types. If the theorem is more specific to the current goal, it is assumed to be more relevant. To avoid large proofs always having the best score, a penalty is given for the size of the type tree, as Type Intersection does for its sets.

```
Theorem rev_involutive :
  forall A (l : list A), rev (rev l) = l.
```

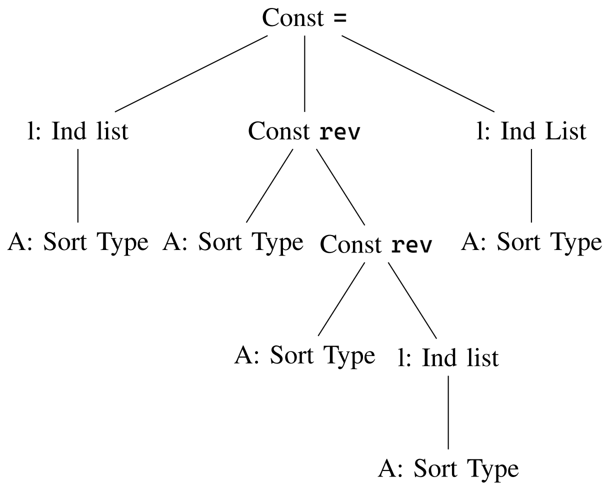


Fig. 8 – The type structure of the theorem `rev_involutive` used in figure 3. The left child of `eq` and `rev` is a type parameter.

To control how this algorithm performs, it accepts two parameters. One is the *Atomic Bias*, which increases the score for matching atomic types compared to matching sorts. The other is the *Size Bias*, which controls how much the size of the theorem affects the final score, higher number meaning size has less influence.

The final score is calculated by the equations:

$$\begin{aligned} atomicScore &= atomicMatches \times atomicBias \\ matchScore &= atomicScore + sortMatches \\ finalScore &= size - (matchScore \times sizeBias) \end{aligned} \tag{7}$$

In the above equations, *atomicMatches* is the amount of matched atomic types and *sortMatches* is the amount of matched sorts. The theorems are sorted, based on their *finalScore*.

To test this algorithm, it was run with all combinations of 1, 2, and 5 for both parameters. The results from all runs can be seen in the appendix (tables 37 through 46), but here the focus will be on the best performing of the matrix, where Atomic Bias is 5 and Size Bias is 5.

As biases 5 and 5 performed the best out of all combinations, a single experiment was also run with an Atomic Bias of 10 and Size Bias of 10, but the performance was worse than using the value 5. The value of 10 was not included in the combinations due to it increasing the total test run time too much, but the single experiment can be seen in the appendix in table 46.

As can be seen in table 6, Type Structure generally performs better than Split Type Intersection, placing the correct theorem first for no prefix 26.1% of the time, a 3.3 percentage point increase over the latter. This does however come at a cost, as it generally places the correct theorem in the top ten less frequently than Split Type Intersection, reducing the hit rate by 2.5 percentage points for the no prefix case. This translates into an overall score of 50.87, which is still a half-point improvement over the previous algorithm.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.1%	26.6%	51.5%	51.9%	57.0%	60.2%	64.9%	65.5%	68.5%
2nd	6.9%	27.1%	11.5%	13.2%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.5%	6.3%	4.8%	3.4%	4.2%	3.7%	3.6%	2.9%	3.2%
4-10th	15.6%	12.3%	10.8%	10.8%	6.7%	6.2%	5.2%	5.2%	3.7%
Pass	53.1%	72.2%	78.5%	79.3%	79.5%	81.2%	83.3%	84.1%	85.0%
Fail	46.9%	27.8%	21.5%	20.7%	20.5%	18.8%	16.7%	15.9%	15.0%

TABLE 6 – Test results from Type Structure, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

	1	2	5
1	37.36	45.50	48.43
2	46.55	49.87	50.55
5	50.40	50.83	**50.87**

TABLE 7 – Scores for each test run in the parameter matrix. The horizontal axis represents Atomic Bias, and the vertical axis represents Size Bias. The best performing, (5, 5), is written in bold.

In table 7, the scores of the test runs in the matrix of values are presented. It can be seen that the optimal values from the matrix are an Atomic Bias of 5 and a Size Bias of 5 (5, 5), followed close behind by Atomic Bias set to 2 (2, 5).

Comparing the two directly, as seen in table 8, they perform very similarly, to the point where any deviance in the data can't be considered all too significant. If anything, Atomic Bias 2 and Size Bias 5 (2, 5) performs a little bit better in no-prefix cases, but worse overall in filtered cases.

Prefix	0	1	2	3	4	5	6	7	8
1st	-0.1	-0.3	0.0	+0.1	+0.4	+0.2	+0.2	+0.2	+0.2
2nd	+0.1	+0.7	+0.5	+0.2	+0.1	+0.2	+0.3	+0.2	+0.2
3rd	-0.1	0.0	-0.1	+0.1	+0.2	+0.2	+0.1	-0.1	-0.2
4-10th	-0.3	+0.1	0.0	+0.1	-0.5	-0.4	-0.4	-0.4	-0.2
Pass	-0.5	+0.5	+0.3	+0.5	+0.3	+0.2	+0.1	-0.1	0.0
Fail	+0.5	-0.5	-0.3	-0.5	-0.3	-0.2	-0.1	+0.1	0.0

TABLE 8 – Difference in percentage points between the two best performing parameter sets for Type Structure, obtained by subtracting the percentages of (2, 5) from (5, 5).

Prefix	0	1	2	3	4	5	6	7	8
1st	26.7%	26.9%	51.5%	51.9%	57.0%	60.2%	64.9%	65.5%	68.5%
2nd	6.7%	26.6%	11.4%	13.3%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.5%	6.3%	4.9%	3.5%	4.2%	3.7%	3.6%	2.9%	3.3%
4-10th	15.3%	12.2%	10.7%	10.6%	6.7%	6.2%	5.2%	5.2%	3.6%
Pass	53.1%	72.2%	78.5%	79.3%	79.5%	81.2%	83.3%	84.1%	85.0%
Fail	46.9%	27.8%	21.5%	20.7%	20.5%	18.8%	16.7%	15.9%	15.0%

TABLE 9 – Test results from Type Structure with Unification, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

E. Selective Unification (Score: 51.05)

This algorithm is somewhat different from the previous ones, as it depends on another algorithm to make an ordering, which could be any of the ones described so far. It takes the ordering of theorems, selects the 100 highest-ordered elements, and tries to unify each of them with the goal. Since unification is the same process used by the `apply` tactic, if the unification succeeds, then the theorem can be applied successfully. The results are split into two groups, consisting of theorems that unify, and theorems that do not. The theorems retain their relative ordering within their groups, but all theorems that unify are placed above those that cannot be unified.

The optimal method from the perspective of finding the best theorem to apply would be to attempt unification on all theorems. However, unification can be time consuming, especially for more complex types and expressions. To ensure a responsive editing environment, only a limited amount of theorems can go through the unification attempt. We chose a limit of 100, which is few enough to not suffer performance problems, but finding the best compromise between efficiency and completion usefulness is outside the scope of this report.

Since there is only time to run unification on some of the possible completion candidates, unification should only be run on the candidates that have the highest chance of succeeding. For this reason, it makes sense to first sort with another algorithm to improve the odds of the correct answer being in the first 100 results, and then try unifying these results.

For comparison's sake, we applied Selective Unification to the results of Baseline, Type Intersection, Split Type Intersection and Type Structure with the entire set of parameters.

The results from all of these tests can be found in the appendix, and a summary of the results can also be found in section IV-G. Of all the algorithms, the one that performed best with Selective Unification was Type Structure with biases (5, 5), and the results of this algorithm can be seen in table 9.

As can be seen in the results, Selective Unification places the correct theorem at the first position 26.7% of the time, a 0.6 percentage point improvement over not using unification. However, this better placement comes at the cost of placing within the rest of the top ten less often, resulting in an identical top ten hit rate as Type Structure. This better placement within the top ten results in a score of 51.05, an improvement of 0.2 points.

Prefix	0	1	2	3	4	5	6	7	8
1st	+0.6	+0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2nd	-0.2	-0.4	-0.1	+0.1	0.0	0.0	0.0	0.0	0.0
3rd	0.0	+0.1	+0.1	+0.1	0.0	0.0	0.0	0.0	+0.1
4-10th	-0.3	0.0	0.0	-0.2	0.0	+0.1	0.0	0.0	-0.1
Pass	0.0	0.0	0.0	0.0	0.0	+0.1	0.0	0.0	0.0
Fail	0.0	0.0	0.0	0.0	0.0	-0.1	0.0	0.0	0.0

TABLE 10 – Difference in percentage points between Type Structure with and without Selective Unification, obtained by subtracting the percentages of the algorithm without unification from the one with unification.

Algorithm	Score
Baseline	26.26
Type Intersection	49.77
Split Type Intersection	50.54
Type Structure (5, 5)	51.05

(a) Scores from running the algorithms with Selective Unification.

	1	2	5
1	37.44	45.64	48.60
2	46.71	50.05	50.68
5	50.57	50.98	51.05

(b) Matrix of scores from Type Structure with Selective Unification. Atomic Bias is on the horizontal axis, and Size Bias on the vertical.

TABLE 11 – All scores from the different runs with Selective Unification.

Table 10 reflects this slight improvement well, showing the change in percentage points from adding Selective Unification to the Type Structure algorithm. Very few score changes occurred, with the most significant change being for short prefixes.

As can be seen in table 11, Type Structure is the best algorithm to use with Selective Unification. The others benefit from unification too, all producing scores 0.1-0.2 points higher than the base algorithms.

F. Selective Split Unification (Score: 51.13)

This algorithm is similar to Selective Unification mentioned above, but also tries to emulate the effects of conditional application. This is similar to the change from Type Intersection to Split Type Intersection. Specifically, the algorithm removes premises from the theorem one at a time and tries to unify the remaining theorem with the goal. If any of these conditional applications are successful, then it means the theorem can be applied, but will result in one or more new goals that must be proven. Like Selective Unification, the algorithm splits the given theorems into groups which retain the relative ordering. However, Selective Split Unification adds a third group, consisting of theorems that can be conditionally applied. The final ordering first contains the complete unification, followed by the conditional unifications, and lastly the rest of the completions.

The version that has been benchmarked still defaults to 100 theorems being selected for unification.

Prefix	0	1	2	3	4	5	6	7	8
1st	27.0%	26.9%	51.6%	51.9%	57.1%	60.2%	64.9%	65.5%	68.5%
2nd	6.6%	26.6%	11.4%	13.3%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.4%	6.3%	4.9%	3.5%	4.2%	3.7%	3.6%	2.9%	3.3%
4-10th	15.3%	12.2%	10.7%	10.6%	6.7%	6.2%	5.2%	5.2%	3.6%
Pass	53.3%	72.2%	78.6%	79.4%	79.6%	81.2%	83.3%	84.1%	85.0%
Fail	46.7%	27.8%	21.4%	20.6%	20.4%	18.8%	16.7%	15.9%	15.0%

TABLE 12 – Test results from Type Structure with Split Unification, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	+0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2nd	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3rd	-0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4-10th	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1-10th	+0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Fail	-0.1	0.0	0.0	0.0	-0.1	0.0	0.0	0.0	0.0

TABLE 13 – Difference in percentage points between Selective Split Unification and Selective Unification on Type Structure with biases 5 and 5.

Again choosing to look at Type Structure, table 12 shows the results of applying Selective Split Unification to Type Structure with parameters 5 and 5. This version guesses the correct theorem from no prefix 27% of the time and manages to hit the top ten 53.3% of the time.

Looking at table 13, it shows once again that improvements are marginal. Applying Selective Split Unification results in a 0.3 percentage points increase in correct guesses with no prefix when compared to Selective Unification. When comparing to Type Structure the improvement is 0.9 percentage points for no prefix and first position.

When looking at all the scores in table 14, it can be seen that Selective Split Unification provides minor improvements to every algorithm. It also improves marginally improves on the results from Selective Unification as shown in table 15.

Algorithm	Score
Baseline	26.33
Type Intersection	49.97
Split Type Intersection	50.61
Type Structure (5, 5)	51.13

(a) Scores from running the algorithms with Selective Split Unification.

	1	2	5
1	37.48	45.71	48.68
2	46.76	50.13	50.76
5	50.66	51.06	51.13

(b) Matrix of scores from Type Structure with Selective Split Unification. Atomic Bias is on the horizontal axis, and Size Bias on the vertical.

TABLE 14 – All scores from the different runs with Selective Split Unification.

Algorithm	Score	Score with Unification	Score with Split Unification
Baseline	26.03	26.26	26.33
Type Intersection	49.68	49.77	49.97
Split Type Intersection	50.40	50.54	50.61
Type Structure (5, 5)	50.87	51.05	51.13

TABLE 15 – Summary of the experiment results. Parameters for Scored Structure Evaluation are a ranking factor of 5 and size factor of 5 in all three cases, as it performed the best. Full results can be found in the appendix, table 56.

Algorithm	Score	Score with Unification	Score with Split Unification
Baseline	0.11	0.69	0.84
Type Intersection	30.42	30.58	31.11
Split Type Intersection	32.18	32.47	32.66
Type Structure (5, 5)	33.96	34.37	34.57

TABLE 16 – Scores based exclusively on prefix size 0, a p-value of 0 in the scoring formula 2 defined in section III-B. Results for all algorithm parameters can be found in the appendix, table 57.

G. Summary and Analysis

To summarise, we present all algorithms and their scores, along with the scores of augmenting with Selective Unification and Selective Split Unification. We also present the scores that correspond to only considering prefixes of length 0, removing the impact that fuzzy matching has. To compare the results for whether the correct theorem is related to `apply` or `rewrite`, we provide results for each keyword individually. Lastly, a summary of the time performance to discover and sort for each algorithm is given. In all cases, Type Structure uses Atomic Bias 5 and Size Bias 5, as the algorithm performed best using these values.

A summary of the scores from the different algorithms with different levels of unification applied, can be seen in table 15. All type-aware algorithms perform better than the baseline, by a significant margin. The data shows that Type Structure is the best-performing base algorithm and that Selective Unification and Selective Split Unification for all algorithms perform better than the algorithm without unification. It is also clear that the type-aware algorithms perform better than the baseline. There isn't any extreme difference in performance between the type-aware algorithms, as they only deviate in score by up to 2 points.

Algorithm	`apply`	`rewrite`	`apply`, Split Unification	`rewrite`, Split Unification
Baseline	26.69	24.22	27.11	24.26
Type Intersection	50.99	46.34	51.39	46.41
Split Type Intersection	52.03	46.42	52.33	46.44
Type Structure (5, 5)	56.31	38.68	56.58	38.89

TABLE 17 – Scores based on whether completion was for the `apply` or `rewrite` tactic. Results for Selective Unification are omitted for brevity, as they are similar to Selective Split Unification, much like all other results.

Algorithm	Time (ms)	Time, Unification	Time, Split Unification
Baseline	686	687	688
Type Intersection	709	698	700
Split Type Intersection	716	709	709
Type Structure	699	696	697

TABLE 18 – Summary of how long each algorithm takes to provide completions, in milliseconds.

It is also interesting to look at what the score would look like considering the algorithms without fuzzy matching. In table 16, scores calculated from only a prefix of length 0 are listed. The ordering of results is the same as when including prefixes, meaning the best-performing algorithm is still Type Structure with Selective Split Unification. There is an increase in the differences between scores, as the distance between the best and worst type aware algorithms is only around 2 points, while without prefixes, this difference grows to about 4. It is interesting to see the behaviour of Baseline, which has plummeted to a score of 0.11.

As with the results in table 15, the type-aware algorithms are similar in performance but perform noticeably better than Baseline. These results also shows how reliant the Baseline score are on the fuzzy matching in VS Code.

In table 17, scores for each algorithm are listed, isolated to the `apply` and `rewrite` tactics respectively. It is clear from these results that each algorithm works better for `apply` than for `rewrite`. Interestingly, it can also be seen that Split Type Intersection is better than Type Structure at discovering completions for `rewrite`. This might be due to Type Structure following a process that is similar to unification and therefore `apply`.

In table 18, average response times for completions with the different algorithms are listed. These times do not deviate from each other all that much from an average of 700 milliseconds, showing that requests average out to taking the same time for each algorithm.

These times are averages, which does mean that some situations will take longer to complete than others. In a single case, our test suite had a request for Unification which had to time out after 10 seconds, as it seemingly never returned a result. These timeouts are not included in the timing results. The algorithms augmented by unification are not visibly slower than the rest, and their limit of 100 theorems to be unified could therefore likely be increased.

As all the response times are quite similar, it is likely that something other than the algorithm dominates the total run time. Possibilities for this include serialising all the results from the language server, discovering all theorems, or something in the tests which adds a large overhead.

V. Discussion

In this section, we discuss alternative potential criteria for evaluating code completion, followed by the consequences of how our algorithms handle conditional application. We also discuss the fact that development has focused on the `apply` tactic at the expense of other tactics, the lack of support for SSReflect, and the problem of partial sentences.

A. Alternative Evaluation Criteria

The focus of this report has been the comparable usefulness of the algorithms, as observed through the ability to output better completions earlier in the list of results. However, there are of course other desired factors in code completion algorithms.

1) Performance: Code completion should be a fast process, ideally fast enough to feel instant to a user. This criterion has been tested for as part of development, but no formal evaluation of time performance has been made, other than timing the benchmarking for each algorithm in table 18. This is however very relevant, especially to the unification algorithms, since the point of being selective about which theorems to unify is to increase performance to acceptable levels.

A way to ensure that unification completes in a reasonable time, would be to adjust the amount of theorems to unify, based on how fast or slow the process is. The speed of unification is based on the goal and theorems, where larger and more complex types will be slower to unify. This could be done by doing unification on theorems until a reasonable time limit is reached, instead of doing unification on a constant amount of theorems. Using a time limit allows for more unifications in the case of uncomplicated types, while still responding quickly if unification is slow.

2) Resiliency to Errors: Another desirable trait in code completion is that it is reliable, instead of frequently crashing or causing errors. For our algorithms, the only algorithms that have had problems with errors are the ones related to unification. During testing, both unification algorithms regularly either crashed or ended up running indefinitely on proofs, with no obvious pattern as to which proofs provoked these.

To maintain the focus on the quality of completions, errors were handled in the following ways. If unification crashed, it simply returned the original list of completions, meaning it would not perform worse than the algorithm used to perform the initial ordering. The cases where unification ran indefinitely were not included in the data set. This gives unification a better score than it might have otherwise. If these errors had been considered as cases where the algorithm simply failed to produce the correct answer, then the score would have been worse. However, these errors are likely more due to faulty implementation in our algorithms or test suite, and we therefore believe not letting their reliability affect their benchmarking results is the right choice.

3) Long-Term Value: The algorithms in this paper mainly focus on finding which theorems might be successfully applied. Selective Unification goes a step further and checks if applying the theorem proves the goal in a single step or not. Selective Split Unification goes even further, by distinguishing between complete application and conditional application.

If several theorems are found that can be conditionally applied, then it would be valuable to find theorems that are more likely to result in the goal being proven later. One such method might be to look at the premises in theorems and check if they exist as hypotheses. If a theorem only has premises that already exist, then the goal can be proven by applying the theorem and afterwards applying the relevant hypotheses.

However, such a method is likely very computationally expensive, given that doing so would involve trying to unify each available hypothesis with every premise in every theorem. It is therefore uncertain if such an algorithm would be feasible.

4) User Experience: An additional relevant criterion for judging the algorithms would be user experience. User experience is affected by all of the other criteria. Code completion will feel worse if it is slow, crashes often, or gives bad completions.

However, there are other aspects of usability that are not encapsulated by any of the above. Most of the algorithms in this paper use heuristics to determine if a theorem can be applied, but are unable to say with certainty whether they actually can. However, Selective Unification can know with certainty if a theorem can be applied, while Selective Split Unification can know if a theorem can be conditionally applied. This can allow the language server to mark theorems if it detects that they can be applied. For example, theorems that can be completely applied can be marked with a star, while conditionally applicable theorems can be marked with an empty star. An example of this can be seen in figure 2 in section I By using a star to signify successful application could allow the user to avoid checking through theorems, as they can be sure that the theorem applies.

B. Matching on Theorem Conclusions

A common theme among the algorithms presented in this paper is to handle conditional application by splitting the theorem at every implication, and then looking at which split best matches the goal. This is explicitly the idea behind Split Type Intersection and Selective Split Unification. It also happens implicitly in Type Structure, which checks every subtree for the best match, and some of these subtrees correspond to the suffixes. This approach was used out of a desire to handle any case, including those where the goal contained an implication. In practice, it is very rare in Coq code to have implications in the goal when trying to apply a theorem. Instead, most developers when doing backwards reasoning will first introduce all premises as hypotheses, and then try to apply theorems to the goal.

This is especially problematic for Split Type Intersection, which uses the overlap of types as a heuristic to try and find the most similar type, using every split as a candidate for the most similar set of types. Consider an example where the goal contains the types A and B with no implications, and we want to compare it to a theorem where the conclusion contains only the type B, but a premise contains A. In this case, the theorem would be evaluated as having a complete type overlap with the goal, but it is unlikely to be applicable, given the conclusion does not have all the types in the goal.

While Type Structure and Selective Split Unification use a similar approach, they are not negatively impacted by it. For Selective Split Unification, the heuristic function for ordering is the same function that Coq uses for application, so it cannot erroneously give a theorem a higher rank. Type Structure uses an approach similar to unification, and should therefore also be unable to make the mistake. However, checking all suffixes and subtrees takes more time, than just checking the conclusion. Therefore, both of these algorithms might be changed to perform faster by only looking at the conclusion, at the expense of being unable to work well in rare cases where the goal contains implications.

C. Focus on apply Tactic

Since the algorithms are only meant to suggest theorems and hypotheses, the algorithms are only benchmarked at areas of the code where this is relevant, specifically right after `apply` and `rewrite`. However, just because our completions can potentially fit in these contexts, it does not mean that the algorithms will be equally optimised for all of them.

For example, the core idea of Selective Unification was that the end goal is to `apply` some theorem, and we therefore use unification, which is the mechanism used by the `apply` tactic to check what theorems are applicable. On a technical level, Type Structure is also similar to the process of unification, causing it to excel in the same places while also having the same shortfalls. The idea of using unification as a code completion heuristic only makes sense if we are trying to optimise completions for `apply`. Looking at the results, the Type Structure algorithm for `rewrite` indeed performs worse when compared to the two intersection algorithms.

To remedy the problem of unification not working well for `rewrite`, an algorithm inspired by Selective Unification could be made. This new algorithm could use the function that `rewrite` uses, to improve results.

It would also be possible to optimise for the `apply in` tactic, supporting forward reasoning, but this is more difficult. Instead of comparing theorems to a single goal, they would have to be compared to all the hypotheses of the proof state, multiplying the computation time by the number of hypotheses. This is similar to the idea and problem presented in section V-A3.

The best code completion is likely achieved by using different strategies for different tactics. For example, using Type Structure for `apply` and Split Type Intersection for `rewrite`. Using this combination would give better results than using either algorithm exclusively.

D. Missing Support for SSReflect

As a significant portion of Coq projects utilises SSReflect, its exclusion presents a problem for our results [35].

Historically, the SSReflect implementation has been provided through MathComp, but as of Coq 8.7, SSReflect was added to the standard library [36]. This means that many significant proofs created using SSReflect use the MathComp version, as they were created before its inclusion in Coq's standard library. As a result, all of these proofs were unavailable for inclusion in our data set.

SSReflect is a set of tactics for small-scale reflection methodology for proofs. Many of these tactics have similar or exact matches in the standard tactics library of Coq. As SSReflect is focused on forward reasoning instead of the backwards reasoning our ranking algorithms are intended for, it is likely that our algorithms perform worse on SSReflect. All of our ranking algorithms rely on matching the entire goal against the theorems and with a larger goal, the matches are likely to be less relevant. Using a different method of ranking when working in SSReflect would therefore likely be fruitful.

E. Partial Sentences

VsCoq, and therefore our auto-completion, only executes code in whole sentences. To reiterate the description from section II-A1, a sentence is the smallest fragment of code that can be interpreted, generally terminated by a period. In many cases, this is not a problem, as a sentence will only correspond to a single use of a tactic.

However, multiple steps may be applied inside the same sentence, such as with the chaining operator (`;`). The chaining operator makes it possible to `apply` tactics to multiple goals at the same time if any previous tactic created sub-goals. From the perspective of VsCoq, a sentence with chaining will keep the same proof state throughout the whole sentence, but when the kernel later validates the proof, the proof state changes with each tactic. Therefore, any subsequent tactic application after the first would be provided completions only relevant to the first application, and be of little relevance to the user.

VI. Conclusion

In this paper, we set out to answer the following problem statement:

> How can a system for providing useful code completion be created, in order to improve the process of writing proofs in Coq?

To do this, we defined usefulness as the ability to put completions desired by the user early in the list. To measure and compare this usefulness, a test suite consisting of several large Coq projects was created. We also created a formula for converting the results into a comparable score, reflecting the ability of algorithms to place the correct answer early in the list, using fewer letters of the prefix.

With our system for testing usefulness, we presented a range of algorithms ranging from a baseline approach that lacked meaningful ordering to simple algorithms based on types, to more sophisticated ones that analysed type structure. We then explored the concept of enhancing the algorithms through Selective Unification, and further expanding their capabilities with conditional application.

In conclusion, Type Structure augmented with Selective Split Unification is the most useful algorithm based on our scoring results. When using no prefix to fuzzy match, the correct answer is the top answer 27.0% of the time, and it is in the top ten 53.3% of the time.

While this conclusion is based on data that does not include SSReflect, it is still representative of the user experience of a large part of the Coq user base. There are also many other criteria for evaluating code completion, such as performance, reliability and user experience, but these are outside the scope of this paper. However, we argue that the most important aspect is still the usefulness of code completion.

Through our efforts, we have improved VsCoq and provided several code completion algorithms, with a comparison showing their relative usefulness, ready to be integrated into VsCoq. This will allow users of VsCoq to more easily make proofs by giving them access to code completion and bringing the experience of program verification closer to the standard of general-purpose programming languages.

A. *Further Work*

An obvious avenue of further work would be creating more completion algorithms for Coq. There are probably still many possibilities for better completion algorithms for theorems. For example, machine learning has had promising results in general-purpose programming languages[37], and has also been investigated in other areas related to Coq[38][39]. One example of this is Diversity-Driven Automated Formal Verification (DIVA) [40] by Sanchez-Stern et al., which uses a diverse set of machine learning models to synthesise proofs. The model works by choosing the most likely next step, so should be possible to use as a heuristic for ranking possible completions.

There are also several other unexplored dimensions of code completions in Coq development that warrant further investigation. The algorithms described here only suggest theorems for application, but code completion is relevant for other tactics as well, and for proving in general. The other modes of interaction, writing functions, specifying theorems, and defining tactics, each requires distinct completion algorithms. Function definition will likely be able to pull ideas from existing completion algorithms for general-purpose programming languages, given its similarity to various functional programming languages. However, both tactics definition and theorem specification will likely require novel work to create useful completion algorithms, due to only having equivalents in other proof assistants that are also missing code completion.

Any code completion implemented for these other interaction modes, could use the benchmarking procedures of this paper with only minor changes required. This would require changes to the test suite to detect more locations which are suitable for completion, other than just `apply` and `rewrite`.

An alternative avenue of inquiry could be looking closer at the results of the algorithms presented in this paper, and especially, trying to identify what scenarios the algorithms handle poorly. It might be possible to look at cases where the algorithms are bad at finding the correct completion and see if patterns exist in the types of goals it fails on. This knowledge could then be used to improve the algorithms and improve the usefulness of completions even further.

One avenue of further research could be replicating the ideas of Robbes and Romain [6], and using more information than just the proof currently being written. In their paper, they use functions that have been recently changed as a basis for completions. It is likely worth investigating whether this approach is useful for the development of proofs.

Lastly, an obvious next step is to merge one of the algorithms proposed in this paper into VsCoq, such that it can be used by a wider audience than this paper can reach. Currently, there are plans for this to happen sometime during the summer of 2023, but will depend on some important improvements being made, like having a timeout for unification and cleaning up the code.

References

[1] *Intellisense*, 2023. [Online]. Available: https://code.visualstudio.com/docs/editor/intellisense (visited on May 20, 2023).

[2] *Content assist*, 2023. [Online]. Available: https://www.eclipse.org/pdt/help/html/working_with_code_assist.htm (visited on May 20, 2023).

[3] H. Dalland, J. Israelsen, and S. Kristensen. “A survey of development tools for proof assistants.” (2023), [Online]. Available: https://github.com/Jakobis/vscoqComparison/blob/main/A_Survey_of_Development_Tools_for_Proof_Assistants.pdf (visited on May 25, 2023).

[4] leanprover. “Lean 4 VSCode Extension.” (2022), [Online]. Available: https://github.com/leanprover/vscode-lean4 (visited on May 29, 2023).

[5] C. Community. “Coq-community/vscoq: VsCoq extension for vs code.” (2023), [Online]. Available: https://github.com/coq-community/vscoq (visited on May 25, 2023).

[6] R. Robbes and M. Lanza, “Improving code completion with program history,” *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, Feb. 2010. DOI: 10.1007/s10515-010-0064-x.

[7] Inria, CNRS and contributors. “Early history of Coq.” (2021), [Online]. Available: https://coq.inria.fr/refman/history.html (visited on May 29, 2023).

[8] Coq. “Coq/coq.” (2023), [Online]. Available: https://github.com/coq/coq (visited on May 26, 2023).

[9] G. Gonthier, “The four colour theorem: Engineering of a formal proof,” in *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, D. Kapur, Ed., ser. Lecture Notes in Computer Science, vol. 5081, Springer, 2007, p. 333. DOI: 10.1007/978-3-540-87827-8_28.

[10] G. Gonthier, A. Asperti, J. Avigad, *et al.*, “A machine-checked proof of the odd order theorem,” in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., ser. Lecture Notes in Computer Science, vol. 7998, Springer, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[11] S. Bernard, C. Cohen, A. Mahboubi, and P.-Y. Strub, “Unsolvability of the Quintic Formalized in Dependent Type Theory,” in *ITP 2021 - 12th International Conference on Interactive Theorem Proving*, Rome / Virtual, France, Jun. 2021. [Online]. Available: https://inria.hal.science/hal-03136002.

[12] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, “CompCert - A Formally Verified Optimizing Compiler,” in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, SEE, Toulouse, France, Jan. 2016. [Online]. Available: https://inria.hal.science/hal-01238879.

[13] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, “A formally-verified C static analyzer,” in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds., ACM, 2015, pp. 247–259. DOI: 10.1145/2676726.2676966.

[14] S. Conchon and J. Filliâtre, “A persistent union-find data structure,” in *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, C. V. Russo and D. Dreyer, Eds., ACM, 2007, pp. 37–46. DOI: 10.1145/1292535.1292541.

[15] E. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain, “A3PAT, an Approach for Certified Automated Termination Proofs,” in *2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, J. Gallagher and J. Voigtländer, Eds., ser. PEPM’10, ACM, Madrid, Spain: ACM, Jan. 2010, pp. 63–72. DOI: 10.1145/1706356.1706370. [Online]. Available: https://inria.hal.science/inria-00535655.

[16] Inria. “Core language.” (2021), [Online]. Available: https://coq.inria.fr/refman/language/core/index.html (visited on May 29, 2023).

[17] Coq Maintainers. “Proof mode.” (2022), [Online]. Available: https://coq.inria.fr/refman/proofs/writing-proofs/proof-mode.html (visited on May 29, 2023).

[18] Coq Maintainers. “Typing rules.” (2022), [Online]. Available: https://coq.inria.fr/refman/language/cic.html#id6 (visited on May 29, 2023).

[19] Coq. “Coq/coq – coq/constr.mli line 239 to line 289.” (2023), [Online]. Available: https://github.com/coq/coq/blob/41f8a12ef14803fef6dcd925ca4df31fab4d204b/kernel/constr.mli#L239-L289 (visited on May 19, 2023).

[20] J. P. Seldin, “N. g. de bruijn. lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. koninklyke nederlandse akademie van wetenschappen, proceedings, ser. a vol. 75 (1972), pp. 381–392; also indagationes mathematicae, vol. 34 (1972), pp. 381–392.,” *The Journal of Symbolic Logic*, vol. 40, no. 3, pp. 470–470, 1975. DOI: 10.2307/2272213.

[21] StackOverflow. “2022 developer survey.” (2022), [Online]. Available: https://survey.stackoverflow.co/2022/#integrated-development-environment (visited on May 29, 2023).

[22] Microsoft. “Language Server Protocol.” (2022), [Online]. Available: https://microsoft.github.io/language-server-protocol/ (visited on May 29, 2023).

[23] Coq maintainers. “Language server extension guide.” (2023), [Online]. Available: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on May 29, 2023).

[24] C. J. Bell. “Projects.” (2016), [Online]. Available: https://people.csail.mit.edu/cj/projects.shtml (visited on May 31, 2023).

[25] M. Dénès. “VSCoq - Microsoft Marketplace.” (2022), [Online]. Available: https://marketplace.visualstudio.com/items?itemName=maximedenes.vscoq (visited on May 29, 2023).

[26] The Coq development team, INRIA, CNRS, and contributors. “The coq proof assistant, xml protocol server.” (2023), [Online]. Available: https://opam.ocaml.org/packages/coqide-server/coqide-server.8.17.0/ (visited on May 31, 2023).

[27] Coq. “Coq/coq – coq/evar.mli line 22 to line 24.” (2023), [Online]. Available: https://github.com/coq/coq/blob/f0def8e3cc680f4cb2e675d31096b434a1e31c9f/kernel/evar.mli#L22-L24 (visited on May 19, 2023).

[28] University of Cambridge and Technische Universität München et al. “Isabelle.” (2022), [Online]. Available: https://isabelle.in.tum.de/ (visited on May 29, 2023).

[29] N. Craswell, “Mean reciprocal rank,” in *Encyclopedia of Database Systems*, Springer US, 2009, pp. 1703–1703. DOI: 10.1007/978-0-387-39940-9_488.

[30] V. Kochnev. “Marshall-lee/software_foundations: Solutions to Software Foundations.” (2023), [Online]. Available: https://github.com/marshall-lee/software_foundations (visited on May 12, 2023).

[31] B. C. Pierce, A. A. de Amorim, C. Casinghino, *et al.* “Software foundations – logical foundations.” (2023), [Online]. Available: https://softwarefoundations.cis.upenn.edu/lf-current/index.html (visited on May 12, 2023).

[32] C. Community. “Coq-community/coq-100-theorems: 100 famous theorems proved using Coq.” (2022), [Online]. Available: https://github.com/coq-community/coq-100-theorems/ (visited on May 12, 2023).

[33] N. Kahl. “The hundred greatest theorems.” (2023), [Online]. Available: http://pirate.shu.edu/~kahlnath/Top100.html (visited on May 12, 2023).

[34] K. Yang and J. Deng, “Learning to prove theorems via interacting with proof assistants,” *CoRR*, vol. abs/1905.09381, 2019. arXiv: 1905.09381. [Online]. Available: http://arxiv.org/abs/1905.09381.

[35] G. Gonthier, A. Mahboubi, and E. Tassi, “A Small Scale Reflection Extension for the Coq system,” Inria Saclay Ile de France, Research Report, 2008. [Online]. Available: https://hal.inria.fr/inria-00258384.

[36] Inria, CNRS and contributors. “Coq Docs - The SSReflect proof language.” (2021), [Online]. Available: https://coq.inria.fr/refman/proof-engine/ssreflect-proof-language.html (visited on May 29, 2023).

[37] S. N. Maxim Tabachnyk. “Ml-enhanced code completion improves developer productivity.” (2022), [Online]. Available: https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html (visited on May 29, 2023).

[38] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer, *Passport: Improving automated formal verification using identifiers*, 2022. DOI: 10.48550/ARXIV.2204.10370. [Online]. Available: https://arxiv.org/abs/2204.10370.

[39] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, “Roosterize: Suggesting lemma names for coq verification projects using deep learning,” *CoRR*, vol. abs/2103.01346, 2021. arXiv: 2103.01346. [Online]. Available: https://arxiv.org/abs/2103.01346.

[40] E. First and Y. Brun, “Diversity-driven automated formal verification,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 749–761, ISBN: 9781450392211. DOI: 10.1145/3510003.3510138.

Appendix A
Test Results From All Test Runs

Prefix	0	1	2	3	4	5	6	7	8
1st	0.1%	8.5%	29.8%	35.0%	38.6%	41.9%	49.1%	50.2%	53.8%
2nd	0.0%	4.1%	10.0%	10.4%	9.9%	11.5%	12.8%	15.1%	14.0%
3rd	0.0%	3.5%	5.2%	5.7%	6.3%	6.5%	5.9%	5.3%	6.2%
4-10th	0.2%	14.3%	17.4%	18.9%	17.9%	16.2%	10.8%	11.4%	10.2%
Pass	0.2%	30.4%	62.3%	70.0%	72.7%	76.1%	78.7%	82.0%	84.2%
Fail	99.8%	69.6%	37.7%	30.0%	27.3%	23.9%	21.3%	18.0%	15.8%

TABLE 19 – Test results from Baseline, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	0.7%	8.8%	29.7%	35.0%	38.6%	41.9%	49.1%	50.2%	53.8%
2nd	0.0%	4.1%	10.0%	10.4%	9.9%	11.5%	12.8%	15.0%	14.0%
3rd	0.0%	3.4%	5.2%	5.7%	6.4%	6.5%	5.9%	5.3%	6.2%
4-10th	0.2%	14.1%	17.4%	18.8%	17.9%	16.2%	10.8%	11.4%	10.2%
Pass	0.8%	30.4%	62.3%	70.0%	72.7%	76.1%	78.6%	81.9%	84.1%
Fail	99.2%	69.6%	37.7%	30.0%	27.3%	23.9%	21.4%	18.1%	15.9%

TABLE 20 – Test results from Baseline with Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	0.8%	8.9%	29.7%	35.0%	38.6%	41.9%	49.1%	50.2%	53.8%
2nd	0.1%	4.1%	10.0%	10.4%	9.9%	11.5%	12.8%	15.0%	14.0%
3rd	0.0%	3.4%	5.2%	5.7%	6.4%	6.5%	5.9%	5.3%	6.2%
4-10th	0.2%	14.1%	17.4%	18.8%	17.9%	16.2%	10.8%	11.4%	10.2%
Pass	1.0%	30.4%	62.3%	70.0%	72.7%	76.1%	78.6%	81.9%	84.1%
Fail	99.0%	69.6%	37.7%	30.0%	27.3%	23.9%	21.4%	18.1%	15.9%

TABLE 21 – Test results from Baseline with Split Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	22.4%	26.4%	50.5%	51.2%	55.9%	58.4%	62.8%	63.1%	66.1%
2nd	7.7%	27.1%	13.7%	14.4%	12.3%	11.8%	10.1%	11.5%	9.9%
3rd	5.6%	9.5%	7.1%	6.9%	5.8%	5.5%	4.4%	3.8%	4.0%
4-10th	17.4%	12.9%	9.8%	8.1%	7.3%	6.4%	6.7%	5.8%	5.0%
Pass	53.0%	75.8%	81.2%	80.6%	81.3%	82.0%	83.9%	84.2%	85.0%
Fail	47.0%	24.1%	18.9%	19.4%	18.7%	18.0%	16.1%	15.8%	15.0%

TABLE 22 – Test results from Type Intersection with Split Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	21.5%	26.2%	50.4%	51.2%	55.9%	58.2%	62.7%	63.0%	66.0%
2nd	7.8%	27.2%	13.8%	14.4%	12.4%	11.9%	10.1%	11.6%	10.0%
3rd	5.6%	9.5%	7.0%	6.8%	5.7%	5.4%	4.5%	3.7%	3.9%
4-10th	18.0%	12.9%	9.9%	8.2%	7.4%	6.4%	6.7%	5.8%	5.0%
Pass	52.9%	75.8%	81.1%	80.6%	81.3%	82.0%	83.9%	84.2%	85.0%
Fail	47.1%	24.2%	18.9%	19.4%	18.7%	18.0%	16.1%	15.8%	15.0%

TABLE 23 – Test results from Type Intersection, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	21.8%	26.3%	50.5%	51.2%	55.9%	58.4%	62.8%	63.1%	66.1%
2nd	7.6%	27.1%	13.8%	14.5%	12.3%	11.8%	10.1%	11.5%	9.9%
3rd	5.6%	9.6%	7.1%	6.9%	5.8%	5.5%	4.4%	3.8%	4.0%
4-10th	18.0%	12.9%	9.8%	8.1%	7.3%	6.4%	6.7%	5.8%	5.0%
Pass	53.0%	75.8%	81.2%	80.6%	81.3%	82.0%	83.9%	84.2%	85.0%
Fail	47.0%	24.1%	18.9%	19.4%	18.7%	18.0%	16.1%	15.8%	15.0%

TABLE 24 – Test results from Type Intersection with Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	22.8%	26.1%	50.2%	50.2%	55.9%	58.8%	63.8%	64.0%	67.4%
2nd	8.2%	27.7%	13.6%	14.7%	12.7%	11.9%	9.3%	11.2%	9.2%
3rd	6.4%	9.5%	7.0%	6.8%	5.9%	5.2%	4.8%	3.3%	3.4%
4-10th	18.2%	13.0%	10.5%	9.4%	7.0%	6.2%	6.3%	6.0%	5.3%
Pass	55.6%	76.3%	81.3%	81.1%	81.4%	82.2%	84.2%	84.5%	85.4%
Fail	44.4%	23.7%	18.7%	18.9%	18.6%	17.8%	15.8%	15.6%	14.6%

TABLE 25 – Test results from Split Type Intersection, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	23.6%	26.4%	50.2%	50.2%	56.0%	58.9%	63.9%	64.1%	67.5%
2nd	7.7%	27.4%	13.6%	14.8%	12.6%	11.8%	9.2%	11.1%	9.0%
3rd	6.3%	9.5%	7.0%	6.8%	5.8%	5.2%	4.8%	3.3%	3.4%
4-10th	18.1%	13.0%	10.4%	9.3%	7.0%	6.2%	6.2%	5.9%	5.3%
Pass	55.7%	76.3%	81.3%	81.1%	81.3%	82.1%	84.1%	84.4%	85.3%
Fail	44.3%	23.7%	18.7%	18.9%	18.6%	17.9%	15.9%	15.6%	14.7%

TABLE 26 – Test results from Split Type Intersection with Split Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	23.3%	26.4%	50.2%	50.2%	56.0%	58.9%	63.9%	64.1%	67.5%
2nd	7.8%	27.4%	13.6%	14.8%	12.6%	11.8%	9.2%	11.1%	9.0%
3rd	6.4%	9.5%	7.0%	6.8%	5.8%	5.2%	4.8%	3.3%	3.4%
4-10th	18.2%	13.0%	10.4%	9.3%	7.0%	6.2%	6.2%	5.9%	5.3%
Pass	55.7%	76.3%	81.3%	81.1%	81.3%	82.1%	84.1%	84.4%	85.3%
Fail	44.3%	23.7%	18.7%	18.9%	18.6%	17.9%	15.9%	15.6%	14.7%

TABLE 27 – Test results from Split Type Intersection with Unification, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	18.1%	15.8%	36.4%	40.4%	42.7%	47.7%	56.7%	57.4%	61.3%
2nd	1.7%	10.2%	11.8%	10.0%	10.5%	11.2%	9.7%	11.2%	10.5%
3rd	0.7%	5.5%	5.1%	6.6%	7.4%	4.4%	4.0%	5.1%	4.2%
4-10th	3.3%	13.7%	17.5%	18.2%	16.0%	14.4%	9.8%	9.3%	8.0%
Pass	23.7%	45.1%	70.7%	75.2%	76.6%	77.7%	80.2%	83.1%	84.0%
Fail	76.3%	54.9%	29.3%	24.8%	23.4%	22.3%	19.8%	16.9%	16.1%

TABLE 28 – Test results from Type Structure with Split Unification, with Atomic Bias 1 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.5%	23.2%	45.4%	48.3%	51.9%	55.7%	60.8%	61.8%	65.3%
2nd	5.3%	21.6%	12.6%	11.4%	10.8%	10.4%	10.2%	11.0%	10.0%
3rd	3.4%	5.9%	4.2%	4.5%	5.1%	4.2%	3.8%	3.6%	3.6%
4-10th	9.5%	10.5%	13.0%	13.0%	10.0%	9.3%	7.3%	7.7%	6.0%
Pass	43.7%	61.2%	75.2%	77.1%	77.9%	79.6%	82.1%	84.2%	84.8%
Fail	56.3%	38.8%	24.8%	22.9%	22.1%	20.4%	17.9%	15.8%	15.2%

TABLE 29 – Test results from Type Structure with Split Unification, with Atomic Bias 1 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.9%	27.1%	51.1%	51.4%	56.2%	59.7%	64.1%	64.9%	68.3%
2nd	6.2%	25.1%	10.8%	12.6%	11.1%	10.3%	9.9%	10.6%	9.5%
3rd	4.9%	7.0%	4.7%	3.2%	4.2%	3.7%	3.2%	3.0%	2.7%
4-10th	15.4%	11.0%	10.8%	11.1%	7.5%	6.9%	5.9%	5.8%	4.5%
Pass	53.5%	70.2%	77.4%	78.3%	78.9%	80.6%	83.1%	84.2%	85.0%
Fail	46.5%	29.9%	22.6%	21.7%	21.1%	19.4%	16.9%	15.8%	15.0%

TABLE 30 – Test results from Type Structure with Split Unification, with Atomic Bias 1 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	24.7%	23.0%	45.5%	48.9%	52.1%	56.0%	61.0%	61.9%	65.2%
2nd	3.9%	18.1%	11.2%	11.3%	11.2%	10.4%	10.1%	10.9%	10.1%
3rd	3.0%	7.1%	5.7%	4.7%	4.7%	4.0%	3.8%	3.5%	3.8%
4-10th	4.9%	10.4%	12.3%	12.1%	9.6%	9.0%	7.0%	7.7%	5.7%
Pass	36.4%	58.7%	74.7%	76.9%	77.6%	79.4%	81.9%	84.0%	84.7%
Fail	63.6%	41.3%	25.3%	23.1%	22.4%	20.6%	18.1%	16.0%	15.3%

TABLE 31 – Test results from Type Structure with Split Unification, with Atomic Bias 2 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.7%	27.0%	50.6%	51.0%	55.8%	59.0%	63.8%	64.4%	68.0%
2nd	6.2%	24.2%	10.9%	12.9%	11.6%	10.6%	9.7%	10.6%	9.3%
3rd	4.7%	6.2%	4.5%	3.7%	3.8%	3.5%	3.4%	2.8%	3.2%
4-10th	13.6%	11.8%	11.2%	10.5%	7.5%	6.9%	5.9%	6.5%	4.6%
Pass	51.2%	69.2%	77.2%	78.1%	78.7%	80.0%	82.7%	84.3%	85.0%
Fail	48.8%	30.8%	22.8%	21.9%	21.4%	20.0%	17.3%	15.7%	15.0%

TABLE 32 – Test results from Type Structure with Split Unification, with Atomic Bias 2 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	27.1%	27.2%	51.6%	51.8%	56.7%	59.9%	64.7%	65.2%	68.2%
2nd	6.6%	26.0%	10.9%	13.1%	11.4%	10.9%	9.3%	10.3%	9.4%
3rd	4.5%	6.3%	5.0%	3.3%	3.9%	3.5%	3.6%	3.0%	3.4%
4-10th	15.6%	12.1%	10.7%	10.6%	7.1%	6.5%	5.5%	5.5%	3.8%
Pass	53.7%	71.7%	78.2%	78.8%	79.2%	80.8%	83.1%	84.1%	84.8%
Fail	46.3%	28.4%	21.8%	21.2%	20.8%	19.2%	16.9%	15.9%	15.2%

TABLE 33 – Test results from Type Structure with Split Unification, with Atomic Bias 2 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.8%	26.2%	50.6%	51.6%	56.3%	59.6%	64.2%	64.7%	68.1%
2nd	4.4%	22.8%	10.6%	12.8%	11.3%	10.3%	9.8%	10.7%	9.5%
3rd	2.8%	6.6%	4.9%	3.4%	3.8%	3.7%	3.0%	2.8%	3.0%
4-10th	6.1%	11.5%	11.2%	10.5%	7.4%	6.6%	5.8%	6.0%	4.4%
Pass	39.0%	67.1%	77.3%	78.4%	78.8%	80.3%	82.7%	84.1%	85.0%
Fail	61.0%	32.9%	22.7%	21.6%	21.2%	19.7%	17.3%	15.9%	15.0%

TABLE 34 – Test results from Type Structure with Split Unification, with Atomic Bias 5 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	27.1%	27.1%	51.7%	51.7%	56.7%	59.8%	64.5%	65.2%	68.3%
2nd	6.4%	26.2%	10.9%	13.1%	11.4%	10.8%	9.5%	10.4%	9.5%
3rd	4.0%	6.4%	5.0%	3.6%	4.0%	3.6%	3.6%	2.8%	3.1%
4-10th	13.2%	12.0%	10.4%	10.5%	7.2%	6.6%	5.5%	5.5%	4.0%
Pass	50.6%	71.7%	78.0%	79.0%	79.3%	80.8%	83.1%	83.9%	84.9%
Fail	49.4%	28.4%	22.0%	21.0%	20.7%	19.2%	16.9%	16.1%	15.1%

TABLE 35 – Test results from Type Structure with Split Unification, with Atomic Bias 5 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	27.0%	26.9%	51.6%	51.9%	57.1%	60.2%	64.9%	65.5%	68.5%
2nd	6.6%	26.6%	11.4%	13.3%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.4%	6.3%	4.9%	3.5%	4.2%	3.7%	3.6%	2.9%	3.3%
4-10th	15.3%	12.2%	10.7%	10.6%	6.7%	6.2%	5.2%	5.2%	3.6%
Pass	53.3%	72.2%	78.6%	79.4%	79.6%	81.2%	83.3%	84.1%	85.0%
Fail	46.7%	27.8%	21.4%	20.6%	20.4%	18.8%	16.7%	15.9%	15.0%

TABLE 36 – Test results from Type Structure with Split Unification, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	17.6%	15.6%	36.3%	40.5%	42.7%	47.7%	56.7%	57.4%	61.3%
2nd	1.9%	10.4%	11.9%	9.9%	10.5%	11.2%	9.7%	11.2%	10.5%
3rd	0.8%	5.4%	5.0%	6.6%	7.4%	4.4%	4.0%	5.1%	4.2%
4-10th	3.3%	13.7%	17.6%	18.2%	16.0%	14.4%	9.8%	9.3%	8.0%
Pass	23.7%	45.1%	70.7%	75.2%	76.6%	77.7%	80.2%	83.1%	84.0%
Fail	76.3%	54.9%	29.3%	24.8%	23.4%	22.3%	19.8%	16.9%	16.1%

TABLE 37 – Test results from Type Structure, with Atomic Bias 1 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	24.7%	22.9%	45.4%	48.3%	51.9%	55.7%	60.8%	61.8%	65.3%
2nd	5.6%	21.9%	12.7%	11.3%	10.8%	10.4%	10.2%	11.0%	10.0%
3rd	3.6%	5.8%	4.1%	4.5%	5.1%	4.2%	3.8%	3.6%	3.6%
4-10th	9.9%	10.5%	13.0%	13.0%	10.0%	9.3%	7.3%	7.7%	6.0%
Pass	43.7%	61.2%	75.2%	77.1%	77.9%	79.6%	82.1%	84.2%	84.8%
Fail	56.3%	38.8%	24.8%	22.9%	22.1%	20.4%	17.9%	15.8%	15.2%

TABLE 38 – Test results from Type Structure, with Atomic Bias 1 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.1%	26.8%	51.0%	51.4%	56.1%	59.6%	64.1%	64.8%	68.3%
2nd	6.5%	25.5%	10.9%	12.5%	11.1%	10.3%	9.8%	10.6%	9.5%
3rd	5.1%	6.9%	4.7%	3.2%	4.2%	3.7%	3.2%	3.0%	2.7%
4-10th	15.8%	11.0%	10.8%	11.2%	7.5%	6.9%	5.9%	5.8%	4.5%
Pass	53.4%	70.1%	77.4%	78.3%	78.9%	80.5%	83.0%	84.2%	85.0%
Fail	46.6%	29.9%	22.6%	21.7%	21.1%	19.5%	17.0%	15.8%	15.0%

TABLE 39 – Test results from Type Structure, with Atomic Bias 1 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	23.9%	22.7%	45.5%	49.0%	52.1%	56.1%	61.2%	62.0%	65.3%
2nd	4.1%	18.6%	11.3%	11.2%	11.1%	10.4%	10.1%	10.9%	10.1%
3rd	3.1%	7.0%	5.7%	4.7%	4.8%	4.0%	3.8%	3.5%	3.8%
4-10th	5.1%	10.4%	12.3%	12.2%	9.6%	9.0%	7.0%	7.7%	5.7%
Pass	36.2%	58.7%	74.8%	77.0%	77.7%	79.5%	82.0%	84.2%	84.8%
Fail	63.8%	41.3%	25.2%	23.0%	22.3%	20.5%	18.0%	15.8%	15.2%

TABLE 40 – Test results from Type Structure, with Atomic Bias 2 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.9%	26.6%	50.5%	51.0%	55.7%	59.1%	63.8%	64.5%	68.0%
2nd	6.5%	24.6%	11.0%	12.8%	11.5%	10.6%	9.7%	10.6%	9.3%
3rd	4.9%	6.2%	4.4%	3.7%	3.9%	3.5%	3.4%	2.8%	3.2%
4-10th	13.9%	11.8%	11.3%	10.6%	7.5%	6.9%	5.9%	6.5%	4.5%
Pass	51.1%	69.2%	77.2%	78.0%	78.6%	80.0%	82.7%	84.3%	85.0%
Fail	48.9%	30.8%	22.8%	22.0%	21.4%	20.0%	17.3%	15.7%	15.0%

TABLE 41 – Test results from Type Structure, with Atomic Bias 2 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.3%	26.9%	51.5%	51.8%	56.6%	60.0%	64.7%	65.3%	68.3%
2nd	6.8%	26.4%	11.0%	13.1%	11.4%	10.9%	9.4%	10.3%	9.4%
3rd	4.7%	6.2%	4.9%	3.3%	4.0%	3.5%	3.6%	3.0%	3.4%
4-10th	15.8%	12.2%	10.8%	10.7%	7.2%	6.6%	5.6%	5.6%	3.9%
Pass	53.6%	71.7%	78.2%	78.8%	79.2%	81.0%	83.2%	84.2%	85.0%
Fail	46.4%	28.3%	21.8%	21.2%	20.8%	19.0%	16.8%	15.8%	15.0%

TABLE 42 – Test results from Type Structure, with Atomic Bias 2 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.0%	25.9%	50.6%	51.7%	56.3%	59.6%	64.2%	64.7%	68.1%
2nd	4.6%	23.2%	10.8%	12.7%	11.3%	10.3%	9.8%	10.7%	9.5%
3rd	3.0%	6.5%	4.8%	3.5%	3.8%	3.7%	3.0%	2.8%	3.0%
4-10th	6.2%	11.5%	11.2%	10.5%	7.4%	6.6%	5.8%	6.0%	4.4%
Pass	38.7%	67.0%	77.3%	78.4%	78.8%	80.3%	82.7%	84.1%	85.0%
Fail	61.3%	33.0%	22.7%	21.6%	21.2%	19.7%	17.3%	15.9%	15.0%

TABLE 43 – Test results from Type Structure, with Atomic Bias 5 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.3%	26.7%	51.7%	51.7%	56.8%	60.0%	64.6%	65.3%	68.4%
2nd	6.6%	26.6%	11.1%	13.1%	11.3%	10.8%	9.5%	10.4%	9.5%
3rd	4.2%	6.3%	4.8%	3.5%	3.9%	3.5%	3.6%	2.8%	3.1%
4-10th	13.5%	12.0%	10.5%	10.7%	7.3%	6.7%	5.5%	5.5%	4.0%
Pass	50.5%	71.7%	78.1%	79.0%	79.3%	81.0%	83.2%	84.1%	85.0%
Fail	49.5%	28.3%	21.9%	21.0%	20.6%	19.0%	16.8%	15.9%	15.0%

TABLE 44 – Test results from Type Structure, with Atomic Bias 5 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.1%	26.6%	51.5%	51.9%	57.0%	60.2%	64.9%	65.5%	68.5%
2nd	6.9%	27.1%	11.5%	13.2%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.5%	6.3%	4.8%	3.4%	4.2%	3.7%	3.6%	2.9%	3.2%
4-10th	15.6%	12.3%	10.8%	10.8%	6.7%	6.2%	5.2%	5.2%	3.7%
Pass	53.1%	72.2%	78.5%	79.3%	79.5%	81.2%	83.3%	84.1%	85.0%
Fail	46.9%	27.8%	21.5%	20.7%	20.5%	18.8%	16.7%	15.9%	15.0%

TABLE 45 – Test results from Type Structure, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.1%	26.5%	51.5%	52.0%	57.2%	60.3%	65.0%	65.5%	68.6%
2nd	7.0%	26.9%	11.2%	13.3%	11.2%	10.9%	9.3%	10.4%	9.3%
3rd	4.4%	6.4%	5.2%	3.3%	4.4%	3.6%	3.8%	2.9%	3.2%
4-10th	15.6%	12.0%	10.3%	10.4%	6.7%	6.4%	5.1%	5.3%	3.8%
Pass	53.1%	71.9%	78.2%	79.0%	79.5%	81.2%	83.2%	84.1%	85.0%
Fail	46.9%	28.1%	21.8%	21.0%	20.5%	18.8%	16.8%	15.9%	15.0%

TABLE 46 – Test results from Type Structure, with Atomic Bias 10 and Size Bias 10, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	17.9%	15.8%	36.4%	40.4%	42.7%	47.7%	56.7%	57.4%	61.3%
2nd	1.7%	10.2%	11.8%	10.0%	10.5%	11.2%	9.7%	11.2%	10.5%
3rd	0.8%	5.5%	5.1%	6.6%	7.4%	4.4%	4.0%	5.1%	4.2%
4-10th	3.3%	13.7%	17.5%	18.2%	16.0%	14.4%	9.8%	9.3%	8.0%
Pass	23.7%	45.1%	70.7%	75.2%	76.6%	77.7%	80.2%	83.1%	84.0%
Fail	76.3%	54.9%	29.3%	24.8%	23.4%	22.3%	19.8%	16.9%	16.1%

TABLE 47 – Test results from Type Structure with Unification, with Atomic Bias 1 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.3%	23.2%	45.4%	48.3%	51.9%	55.7%	60.8%	61.8%	65.3%
2nd	5.3%	21.6%	12.6%	11.4%	10.8%	10.4%	10.2%	11.0%	10.0%
3rd	3.5%	5.9%	4.2%	4.5%	5.1%	4.2%	3.8%	3.6%	3.6%
4-10th	9.6%	10.5%	13.0%	13.0%	10.0%	9.3%	7.3%	7.7%	6.0%
Pass	43.7%	61.2%	75.2%	77.1%	77.9%	79.6%	82.1%	84.2%	84.8%
Fail	56.3%	38.8%	24.8%	22.9%	22.1%	20.4%	17.9%	15.8%	15.2%

TABLE 48 – Test results from Type Structure with Unification, with Atomic Bias 1 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.7%	27.1%	51.0%	51.3%	56.1%	59.6%	64.1%	64.8%	68.3%
2nd	6.3%	25.1%	10.8%	12.6%	11.1%	10.3%	9.8%	10.6%	9.5%
3rd	5.1%	7.0%	4.7%	3.2%	4.2%	3.7%	3.2%	3.0%	2.7%
4-10th	15.5%	10.9%	10.8%	11.1%	7.5%	6.9%	5.9%	5.8%	4.5%
Pass	53.5%	70.1%	77.4%	78.3%	78.9%	80.5%	83.0%	84.2%	85.0%
Fail	46.5%	29.9%	22.6%	21.7%	21.1%	19.5%	17.0%	15.8%	15.0%

TABLE 49 – Test results from Type Structure with Unification, with Atomic Bias 1 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	24.4%	23.0%	45.5%	48.9%	52.1%	56.0%	61.0%	61.9%	65.2%
2nd	3.9%	18.2%	11.3%	11.3%	11.2%	10.4%	10.1%	10.9%	10.1%
3rd	3.1%	7.1%	5.7%	4.7%	4.7%	4.0%	3.8%	3.5%	3.8%
4-10th	4.9%	10.4%	12.3%	12.1%	9.6%	9.0%	7.0%	7.7%	5.7%
Pass	36.4%	58.7%	74.7%	76.9%	77.6%	79.4%	81.9%	84.0%	84.7%
Fail	63.6%	41.3%	25.3%	23.1%	22.4%	20.6%	18.1%	16.0%	15.3%

TABLE 50 – Test results from Type Structure with Unification, with Atomic Bias 2 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.4%	27.0%	50.6%	50.9%	55.7%	59.1%	63.8%	64.5%	68.0%
2nd	6.3%	24.2%	10.9%	12.9%	11.6%	10.6%	9.7%	10.6%	9.3%
3rd	4.8%	6.2%	4.5%	3.7%	3.8%	3.5%	3.4%	2.8%	3.2%
4-10th	13.6%	11.8%	11.2%	10.5%	7.5%	6.9%	5.9%	6.5%	4.5%
Pass	51.2%	69.2%	77.2%	78.0%	78.6%	80.0%	82.7%	84.3%	85.0%
Fail	48.8%	30.8%	22.8%	22.0%	21.4%	20.0%	17.3%	15.7%	15.0%

TABLE 51 – Test results from Type Structure with Unification, with Atomic Bias 2 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.9%	27.2%	51.6%	51.7%	56.7%	59.9%	64.7%	65.3%	68.2%
2nd	6.6%	26.0%	10.9%	13.1%	11.3%	10.9%	9.3%	10.3%	9.4%
3rd	4.6%	6.3%	5.0%	3.3%	3.9%	3.5%	3.6%	3.0%	3.4%
4-10th	15.6%	12.1%	10.7%	10.6%	7.1%	6.5%	5.5%	5.5%	3.8%
Pass	53.6%	71.6%	78.1%	78.7%	79.1%	80.9%	83.1%	84.1%	84.8%
Fail	46.4%	28.4%	21.9%	21.3%	20.9%	19.1%	16.9%	15.9%	15.2%

TABLE 52 – Test results from Type Structure with Unification, with Atomic Bias 2 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	25.5%	26.2%	50.6%	51.6%	56.3%	59.6%	64.2%	64.7%	68.1%
2nd	4.4%	22.9%	10.7%	12.8%	11.3%	10.3%	9.8%	10.7%	9.5%
3rd	2.9%	6.6%	4.9%	3.5%	3.8%	3.7%	3.0%	2.8%	3.0%
4-10th	6.1%	11.5%	11.2%	10.5%	7.4%	6.6%	5.8%	6.0%	4.4%
Pass	38.9%	67.1%	77.3%	78.4%	78.8%	80.3%	82.7%	84.1%	85.0%
Fail	61.1%	32.9%	22.7%	21.6%	21.2%	19.7%	17.3%	15.9%	15.0%

TABLE 53 – Test results from Type Structure with Unification, with Atomic Bias 5 and Size Bias 1, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.8%	27.1%	51.7%	51.7%	56.6%	59.8%	64.5%	65.2%	68.3%
2nd	6.4%	26.2%	10.9%	13.1%	11.3%	10.8%	9.5%	10.4%	9.5%
3rd	4.1%	6.4%	5.0%	3.6%	4.0%	3.6%	3.6%	2.8%	3.1%
4-10th	13.2%	11.9%	10.4%	10.5%	7.2%	6.6%	5.5%	5.5%	4.0%
Pass	50.5%	71.6%	78.0%	78.9%	79.2%	80.9%	83.1%	83.9%	84.9%
Fail	49.5%	28.4%	22.0%	21.1%	20.8%	19.1%	16.9%	16.1%	15.1%

TABLE 54 – Test results from Type Structure with Unification, with Atomic Bias 5 and Size Bias 2, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Prefix	0	1	2	3	4	5	6	7	8
1st	26.7%	26.9%	51.5%	51.9%	57.0%	60.2%	64.9%	65.5%	68.5%
2nd	6.7%	26.6%	11.4%	13.3%	11.5%	11.1%	9.7%	10.4%	9.6%
3rd	4.5%	6.3%	4.9%	3.5%	4.2%	3.7%	3.6%	2.9%	3.3%
4-10th	15.3%	12.2%	10.7%	10.6%	6.7%	6.2%	5.2%	5.2%	3.6%
Pass	53.1%	72.2%	78.5%	79.3%	79.5%	81.2%	83.3%	84.1%	85.0%
Fail	46.9%	27.8%	21.5%	20.7%	20.5%	18.8%	16.7%	15.9%	15.0%

TABLE 55 – Test results from Type Structure with Unification, with Atomic Bias 5 and Size Bias 5, showing for each prefix length how large a proportion of results appeared at each position in the completions.

Algorithm	Atomic Bias	Size Bias	Score
Baseline	-	-	26.03
Baseline with Unification	-	-	26.26
Baseline with Split Unification	-	-	26.33
Type Intersection	-	-	49.68
Type Intersection with Unification	-	-	49.77
Type Intersection with Split Unification	-	-	49.97
Split Type Intersection	-	-	50.40
Split Type Intersection with Unification	-	-	50.54
Split Type Intersection Split Unification	-	-	50.61
Type Structure	1	1	37.36
		2	46.55
		5	50.40
	2	1	45.50
		2	49.87
		5	50.83
	5	1	48.43
		2	50.55
		5	50.87
	10	10	50.84
Type Structure with Unification	1	1	37.44
		2	46.71
		5	50.57
	2	1	45.64
		2	50.05
		5	50.98
	5	1	48.60
		2	50.68
		5	51.05
Type Structure with Split Unification	1	1	37.48
		2	46.76
		5	50.66
	2	1	45.71
		2	50.13
		5	51.06
	5	1	48.68
		2	50.76
		5	51.13

TABLE 56 – Score of all the combinations of algorithms and parameters tests were run for.

Algorithm	Atomic Bias	Size Bias	Score
Baseline	-	-	0.11
Baseline with Unification	-	-	0.69
Baseline with Split Unification	-	-	0.84
Type Intersection	-	-	30.42
Type Intersection with Unification	-	-	30.58
Type Intersection with Split Unification	-	-	31.11
Split Type Intersection	-	-	32.18
Split Type Intersection with Unification	-	-	32.47
Split Type Intersection with Split Unification	-	-	32.66
Type Structure	1	1	19.41
		2	30.39
		5	33.75
	2	1	27.96
		2	33.08
		5	33.96
	5	1	29.38
		2	33.29
		5	33.75
	10	10	33.73
Type Structure with Unification	1	1	19.59
		2	30.76
		5	34.16
	2	1	28.36
		2	33.48
		5	34.37
	5	1	29.78
		2	33.65
		5	34.16
Type Structure with Split Unification	1	1	19.69
		2	30.90
		5	34.36
	2	1	28.54
		2	33.68
		5	34.57
	5	1	29.96
		2	33.85
		5	34.36

TABLE 57 – Score based exclusively on no prefix of all the combinations of algorithms and parameters tests were run for.

Algorithm	Atomic Bias	Size Bias	Time (ms)
Baseline	-	-	686
Baseline with Unification	-	-	687
Baseline with Split Unification	-	-	688
Type Intersection	-	-	709
Type Intersection with Unification	-	-	698
Type Intersection with Split Unification	-	-	700
Split Type Intersection	-	-	716
Split Type Intersection with Unification	-	-	709
Split Type Intersection with Split Unification	-	-	709
Type Structure	1	1	703
		2	703
		5	700
	2	1	701
		2	698
		5	698
	5	1	699
		2	698
		5	699
	10	10	694
Type Structure with Unification	1	1	699
		2	699
		5	700
	2	1	699
		2	700
		5	699
	5	1	698
		2	698
		5	696
Type Structure with Split Unification	1	1	698
		2	697
		5	698
	2	1	698
		2	698
		5	698
	5	1	695
		2	698
		5	697

TABLE 58 – The mean duration of each call for completions, for each algorithm.