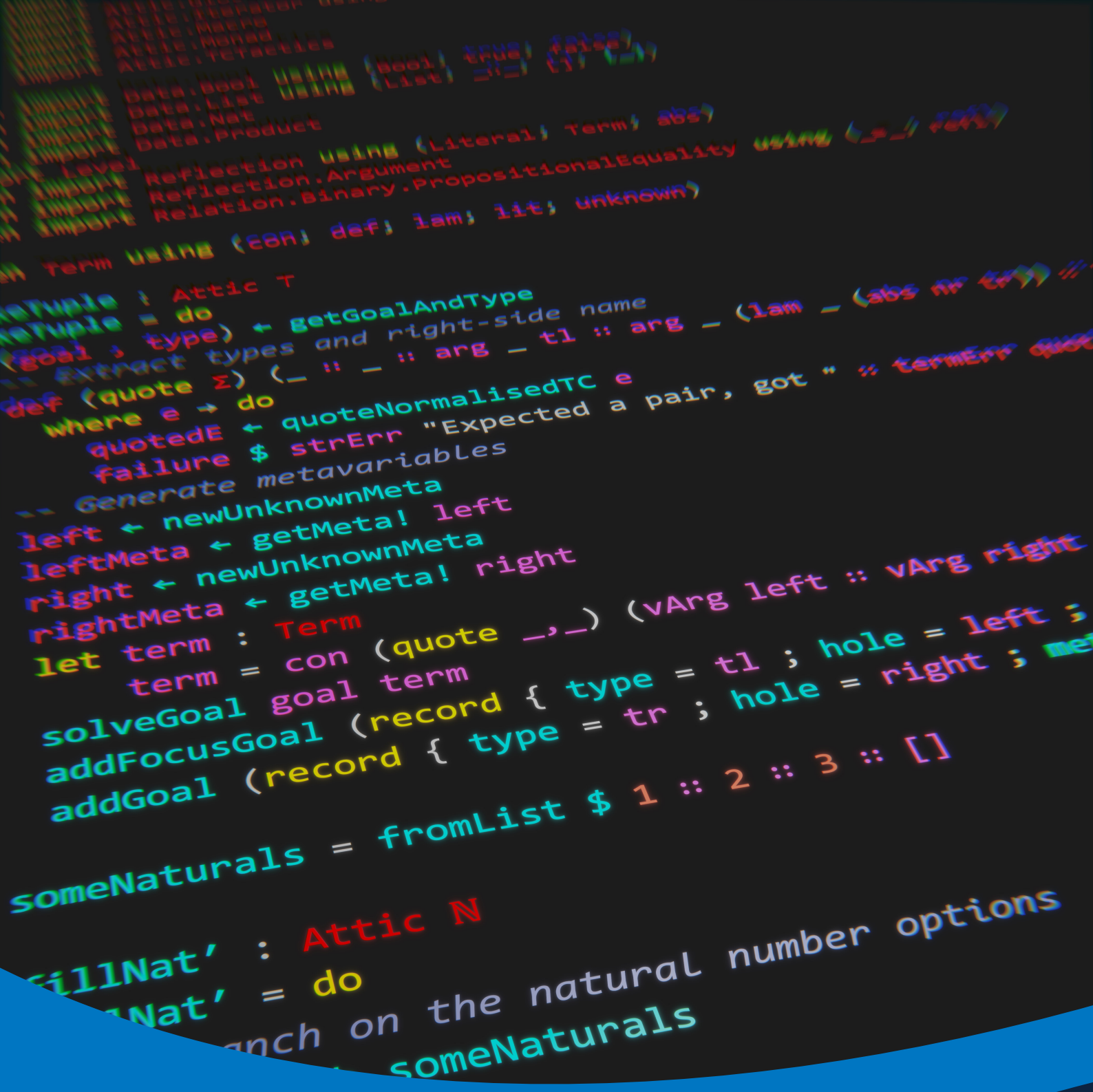


Tactics in Agda using reflection

Paul van der Stel

TU Delft

Tactics in Agda using reflection

Paul van der Stel

2022–09–21

Master's Thesis

Author Pieter Willem Paul van der Stel
Student number 4553799

Defence date 2022–09–21

Specialisation Software Technology
Master programme Computer Science
Faculty Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Abstract

In this thesis, we develop a new library for Agda named Attic, which allows us to create and compose proof tactics that can be used to generate terms through reflection. Such tactics can be converted to Agda macros, allowing them to be used in term positions where they can generate term solutions of the expected type. Tactics can make the development of proofs faster by making proofs easier to read and write.

This project can be seen as a sister project to Ataca, which is an earlier attempt at developing tactics that operate through reflection. Attic explores new mechanisms of operation, such as non-determinism with iterators to allow for multiple solutions, and the use of deferred unification, so that the final proof term is only fully constructed at the end of tactic evaluation.

To allow for the representation of both finite and infinite sequences that can be consumed step-by-step, we have implemented the iterator data type in Agda. Although iterators existed in other systems previously, an Agda implementation had not been made. These iterators underpin the branching mechanism in tactic instructions, and support operations that can be used to generate, transform and filter values.

Finally, we have implemented a number of tactics and operations that are commonly found in other proof assistants. We also compare the resulting library to the Ataca library and examine the differences in runtime for a small test case. While Attic is not yet a complete solution, we present new ideas that may be incorporated in future tactic systems.

Preface

This thesis concerns the development of a library that facilitates the development of tactics for proof authoring in the Agda programming language.

This thesis project has been performed within the Delft University of Technology, under supervision of the TU Delft Programming Languages Group.

The Agda library that was developed as part of this thesis, Attic, allows us to create and compose tactics that can be used to generate proof terms. The tactics work through Agda's reflection machinery. The library contains various tools that are needed to run tactics, such as an interpreter-based evaluation mechanism that supports non-determinism and deferred unification of terms. With these tactics it becomes easier to write formal proofs in Agda, as proof authors no longer need to give full programs, but can use tactics instead. The existing iterator type has also been implemented in Agda.

The thesis committee consists of Andy Zaidman, Jesper Cockx, and Wouter Swierstra.

Samenvatting voor niet-experts

This section is written in Dutch.

Computers zijn niet meer uit ons leven weg te denken. We vinden ze overal om ons heen. De programma's die computers aansturen worden ook steeds complexer. Deze software komt overal voor: de webbrowser op onze mobiele telefoons, de slimme apps op de televisie, en het navigatiesysteem in de auto. Maar software is ook te vinden in de boordcomputer van een vliegtuig, in het systeem dat de hartslag van een intensive care-patient in de gaten houdt, en in de veiligheidssystemen van kernreactoren. Voor dergelijke kritieke software is het belangrijk om te kunnen garanderen dat de software goed werkt, en geen onverwachte fouten bevat.

Om zulke garanties te leveren is een formeel bewijs de gouden standaard. Met wiskundige bewijzen kunnen we aantonen dat een programma voldoet aan bepaalde eisen, en het *certificeren*. Deze bewijzen worden wel erg lang, want hoe complexer een programma is, hoe complexer het bewijs wordt. Het zou dus erg fijn als we dit werk konden automatiseren, zodat een computer dit bewijs voor ons kan leveren. Op deze manier kunnen we software certificeren, maar hoeven we niet ons niet te verdiepen in eindeloze bewijzen.

Computers kunnen ons inderdaad helpen bij het redeneren over zulke eisen. Dit gaat met behulp van speciale programma's: met *bewijsassistenten* (Engels: proof assistants) kunnen we logisch redeneren over wiskundige stellingen. We definiëren eerst onze eis als een formele stelling, en daarna leveren we het bewijs dat deze stelling waar is. Dit bewijs is ook een programma. Dit programma wordt heel nauwkeurig nagekeken door de computer, en deze kan automatisch beredeneren of het bewijs inderdaad een geldig bewijs is voor de stelling. Als het bewijs ongeldig is, geeft de computer een fout. Zo kunnen we wiskundige bewijzen digitaal uitdrukken.

Het ontwikkelen van een bewijs kan nog steeds complex en tijdrovend zijn, vooral voor grote en/of complexe stellingen. Om het schrijven van bewijzen makkelijker te maken hebben veel bewijsassistenten iets slims bedacht: *tactieken*. Deze geven aan hoe we het bewijs te lijf moeten gaan, en iedere tactiek kan gezien worden als een stap om dichter bij een volledig bewijs te komen. Door tactieken te combineren kunnen we dus vele stappen zetten, en uiteindelijk het bewijs afmaken. De bewijsassistent zet deze combinatie van tactieken om in het veel grotere bewijs, en controleert of het bewijs klopt. Tactieken zorgen er dus voor dat bewijzen korter worden, en dus gemakkelijker om te lezen en schrijven.

De programmeertaal Agda kan ook gebruikt worden als bewijsassistent. Hoewel Agda beschikt over moderne functies, heeft het geen ondersteuning voor tactieken. Dat betekent dat bewijzen in Agda volledig moeten worden uitgeschreven.

Het doel van deze thesis is om een bibliotheek van functies te maken die het mogelijk maakt om tactieken te ontwikkelen in Agda. Mijn doel is hierbij om de beste eigenschappen van tactieken in andere systemen te gebruiken, maar de slechte of verwarrende eigenschappen eruit te laten. Zo wordt het mogelijk om ook in Agda tactieken te gebruiken, die bewijzen korter kunnen maken en makkelijker leesbaar maken. Hierbij onderzoeken we nieuwe technieken die niet in bestaande systemen aanwezig zijn, waardoor het opbouwen van het bewijs sneller gaat en het bestaan van meerdere oplossingen mogelijk is.

Acknowledgements

I wish to thank Jesper Cockx for providing me with this topic to work on, and for being my supervisor throughout the thesis project. I am grateful for the suggestions, the explanations of complicated topics, and the help you have given me on both the topic of research and the Agda programming language.

I wish to thank Bohdan Liesnikov for jumping in with feedback and advice, and ultimately becoming a co-supervisor. Thank you for the advice on Agda and explanations of complicated topics.

I wish to thank the late Eelco Visser, whose teachings sparked my interest in programming languages.

I wish to thank Andy Zaidman for stepping in as a member of the thesis committee, and I wish to thank Wouter Swierstra for accepting my invitation to additionally join the thesis committee.

Finally, but most importantly, I wish to thank my dearest family and friends. Without you, without your love, without your support and motivation, this document would not have come to be. Thank you for being there for me, and for the time we have spent together. Thank you so much.

Paul van der Stel, 2022–08–21

Contents

1 Introduction

The programming language and proof assistant Agda does not have tactics. This thesis is about the development of a library with which proof tactics can be written for the Agda programming language. In this introductory section we will give an overview of automated proofs, introduce the Agda programming language, describe some challenges and approaches, and list our contributions.

1.1 Proof assistants

Automated proof checking is a discipline where computers are used to develop and verify formal proofs. With proof checkers, a theorem can be expressed formally such that it can be mechanically checked by a computer. The theorems we define must also be accompanied by a proof term, a computer program which serves as an argument that the theorem holds. Through type checking, computers can decide whether this proof term actually proves the theorem. Using computers to verify proofs means that there is a smaller chance of errors. Checking also becomes faster, which is invaluable for large and complex proofs.

The creation of such proof terms can be very long and complex, especially as theories become larger and more complicated. As a result, proof assistants have been developed which can better help humans with authoring proofs. A person who wishes to prove a theorem will first write the theorem, and the proof assistant will then guide the proof author through the proof interactively, step by step. Proof development typically happens in high-level languages, which is elaborated on and turned into a lower level (less expressive) language. These low-level expressions are then checked by a small trusted kernel, guaranteeing that the proof is actually correct [1].

One feature that several modern proof assistants have is *tactics*. Tactics are small programs that help the proof author move towards a complete proof. Tactics typically perform a small operation, but they can be composed and combined to generate larger tactics, which can solve theorems. They can be found in a variety of proof assistants, being present in systems such as Coq [2], Idris [3], and Lean [4]. In fact, tactics go a long way back, being described in 1984 [5]. With tactics, proof authors no longer need to write out a full proof program. This makes it easier to understand the proof, as unnecessary details are hidden and proofs become shorter.

1.2 Agda

Agda is a dependently typed programming language. It is both a general-purpose language, but can also be used as a proof assistant due to its dependent types [6]. While the programs written in Agda can be regarded as formal proofs, Agda has does not have tactics. Instead, programs must be written out in full notation, although this is made far less cumbersome through implicit parameters, instance arguments, and the many modern and convenient features found in other programming languages that Agda has adopted. Despite these features, writing a proof in Agda is often still not as convenient as writing some tactics which can automatically construct a proof for the theorem.

Instead of tactics, Agda does have a powerful reflection API. Through this reflection API metaprogramming becomes possible; it is possible to write Agda programs that write Agda programs. Such metaprograms are exposed as macros. Macros are unlike the macros one might encounter in other languages, which are usually based on text substitution (either naïve such as in C [7] or hygienic such as in Lisp [8]), or in some way expand or modify the term they are working on. Instead, macros in Agda operate within the type checker that Agda provides. A result of this design choice is that Agda macros operate on a much higher level than macros in other languages; they can programmatically consume or generate all kinds of terms directly, as these terms are simply abstract syntax trees of the Agda source code.

Typically, the macros operate on *holes*, which are part of the program that are yet to be filled in with a term. The compiler is designed to deal with these holes appropriately, and they enable various degrees of interactivity while creating Agda programs. This quality of macros make them an interesting target for the implementation of tactics in Agda. The ability to generate syntax elements and place that syntax inside the hole means that it should be possible to dynamically produce macros, which will then generate the syntax for a proof term that can then be used to fill the hole.

1.3 Challenges

With the implementation of tactics come some challenges that we will attempt to address in this thesis. The reflection API that Agda offers is quite comprehensive, which is both a good thing and a bad thing. It exposes many of the functionalities that Agda uses internally to represent and type-check programs. This enables rich and complex reflection-based programs that can perform many flexible tasks. At the same time, this complexity and flexibility makes the API daunting to users of the languages who are not accustomed to working with this API directly. Although all the functionality the user could need is readily available, they might still be unable to make use of it.

Instead, a library that supports the creation and evaluation of tactics should focus on abstracting away the details and complexities that are not relevant to the creation and evaluation of tactics. While some manual bookkeeping will still be required, a tactic script should not be complex to use. Yet at the same time, the system should also be extensible enough that it remains possible to use more advanced features, both for lower-level tactics contained within the library, as well as tactics created by end users. Our tactic library comes with a number of tactics that abstract away basic operations, so that tactic development remains high-level.

Tactic programs must also be able to deal with backtracking. If a tactic program is not applicable to the current state of the proof, we must be able to 'take a step back' and try another approach. At the same time, we must also not backtrack endlessly, or we might spend too much time evaluating incorrect solutions, so that we never get to the solution(s) we are looking for. We must thus design tactics in such a way that the tactic author remains in control and can choose how failure cases are handled. To this end, we have developed a branching instruction within our tactic language to facilitate non-deterministic evaluation.

Another open challenge is performance of the tactic system. In prior work, the tactic language is often a central feature, and would as a consequence be optimised so that it constructs terms quickly. Although there is some prior work on tactics in Agda [9], the author has stated that this library suffers from some performance issues. It is important that performance is kept in check, so that proof authors do not have to wait unreasonably long for a tactic to succeed. We have made attempts to minimise the amount of work that is done while evaluating tactics, to gain better runtime performance.

1.4 Structure and contributions

We make the following contributions:

- In section 2 we give the theoretical background for automated proofs and proof assistants. We also introduce some aspects of the Agda programming language that will be relevant to this thesis.
- In section 3 we will introduce our first contribution: an iterator type in Agda. With this type, both finite and infinite sequences can be represented. Iterators serve as a type with which branching instructions are implemented.
- In section 4 we will introduce the Attic library. We will show the tactics that can be built with it, and how they are used in practice.
- Section 5 presents a deep dive on the technical implementation of Attic. Herein lies our second contribution: a tactic evaluation mechanism that supports non-determinism through branching, and uses deferred unification of terms.
- In section 6 we will draw a comparison between Attic and a previously developed library that supports the creation of tactics in Agda.
- Section 7 describes future work that would improve Attic. This includes technical challenges, but also describes possible extensions to the library.
- In section 8 we will mention works by others that relate to the concepts we have discussed in this thesis.
- Finally, we conclude this thesis in section 9.

The source code for Attic can be found online, at `https://github.com/pvdstel/Attic`.

2 Background

This section provides background information on proof assistants, their theoretical foundations, and the programming language Agda.

2.1 Proof assistants

Proof checking is the verification of mathematical theories in an automated fashion. This is done by first formalizing the theories, which involves specifying the primitive notions, axioms, definitions and proofs of the theory to be verified [10]. The correctness of such a proof is then verified by a program called the *proof checker*.

The development of such theories and their proofs can be complex and cumbersome. To make this process feasible, proof checkers are usually part of a larger software suite called a *proof assistant*. The core idea is that the user of the proof assistant (also known as a proof author) can write the theory and the corresponding proof in a high-level programming language, from which the proof assistant can generate a proof term that can be verified by the proof checker.

Proof assistants typically offer some sort of interface through which proofs can be developed interactively. Since programming languages are most commonly text-based, such an interface is commonly a text editor with some additional support for the specific features of the proof assistant for which it is being used. Such features usually include the ability to show which theories have or have not yet been proven, as well as the current progress towards a complete proof. Some interfaces also allow the user to refactor existing code, or help with automatically completing phrases in the code.

2.1.1 Proofs as programs

A proof checker will verify a theory within a certain frame of logic. This frame of logic lays out the basic rules that the proof checker will follow while attempting proof verification. A common frame of logic that is used for proof checking is *type theory*, because a type theory is a formal system that has meaning in the fields of mathematics, logic and computer science. This property makes type theory a great candidate for an application that exists on the intersection of these fields. Other frames of logic are also used in the world of proof checkers; the proof assistant Isabelle supports multiple, with the most developed instance being Isabelle/HOL, which uses higher-order logic [11].

A type theory consists of a set of inference rules which can be used to infer judgements. Such judgements express what is true within the type theory. Applying the inference rules to existing judgements may lead to new judgements. A judgement does not necessarily have meaning within the type system; the type system merely dictates the rules with which new judgements can be made. Judgements can be written in several ways, but we will use the example `e : R` here. This judgement can have several interpretations. A mathematician considers that `e` is an element of the set `R`, whereas a logician finds that `e` is a proof of the proposition `R`, and a computer scientist holds that `e` is a term of the type `R`. Since we are interested in proving theories using computers, let us consider the interpretations of the logician and the computer scientist.

Fortunately, we may consider the interpretations of formal proofs and computer programs as equivalent. This property is known as the *Curry–Howard correspondence*, also known as the *proofs-as-programs* or *formulae-as-types interpretation* [12]. In fact, Cartesian closed categories (of which sets are an example) are also equivalent, resulting in a three-way isomorphism which is sometimes called the Curry–Howard–Lambek correspondence [13]. See fig. 1 for an illustration. Through this correspondence, a type can be regarded as a proposition, and a term of that type will then be seen as a proof of that proposition. Thus, we can use well-typed software programs to express proofs for propositions.

This approach is exactly what many proof assistants will do behind the scenes. Proof authors specify the theorem they want to prove, by declaring a value with a type annotation. This type is the theorem that we wish to prove. The expression term that we assign to this value shows that the type is inhabited (a value of the type exists), or alternatively, proves that the theorem which corresponds to the type holds. We will illustrate this using listing 1. The first declaration, `example`, has the integer type. Its associated implementation fills in an integer value, essentially showing that the type is inhabited, and thus proving

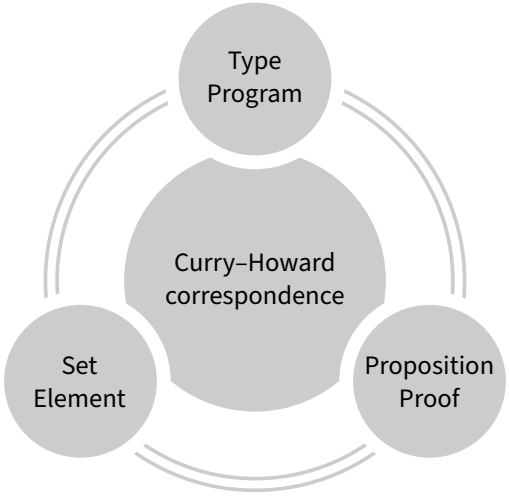


Figure 1: An illustration of the Curry–Howard correspondence. A set, type and proposition are equivalent, as are elements, programs and proofs.

```haskell
example :: Integer
example = 5

isIntTrue :: Integer -> Bool
isIntTrue 0 = False
isIntTrue _ = True
```

Listing 1: Some functions in Haskell, which can be interpreted as theorems and their proofs. The function `isIntTrue` encodes the conversion of integers to booleans in the C and C++ programming languages, where zero is regarded as false, and any nonzero value is regarded as true [14].

that a value of the integer type exists. In the second declaration, `isIntTrue`, we wish to show that if we have a term of the integer type, we can derive a Boolean value from it. The type represents this as a function that maps integers to Booleans. And indeed, the implementation of this function is able to produce a Boolean value. Of course, these examples are fairly trivial, but they illustrate the idea well. We will provide more examples in section 2.3.3 once we have introduced the Agda language.

2.2 Reasoning with tactics

Proving theorems on computers tends to work slightly different compared to the traditional (human-written) method of reasoning. Traditional proofs tend to employ *forwards reasoning*, where we have a set of premises, and then use inference rules to infer new premises, until we finally conclude with a goal. As an example, if we wished to prove $A \wedge B$ we would first prove A and B separately, and then combine them to prove $A \wedge B$. Computers usually make use of *backwards reasoning* instead: we already have a goal (our type/theorem) and then gradually transform and refine that goal until every subgoal has been solved. To illustrate, if we wished to prove $A \wedge B$, we would start with that theorem as a goal. This goal can be transformed into two subgoals A and B which we then need to prove. After both subgoals have been proven, no subgoals are left, and we have proven that the original goal holds. Backwards reasoning is a nice tool, because it allows us to reason about our goal and solve small parts of it, until the entire goal has been solved.

Backwards reasoning can be implemented using a *proof state*. A proof state is the state of the proof assistant, where it is used to keep track of the subgoals as we reason our way through the proof. For example, the proof assistant Coq considers the proof state to consist of one or more unproven goals [15, Proof mode]. Every time we make an inference, we transform the proof state. Some inferences will allow us to solve a subgoal immediately, whereas some others will require some more work and will

spawn one or more new subgoals. This is all stored in the proof state. In addition to the list of unproven goals, Coq also stores some additional data, including the currently focused subgoal (that means, the subgoal which is the current subject of our inferences) [15, Proof mode]. While Coq does not explicitly consider this to be part of the proof state, it is part of Coq's internal state.

The ability to use computer programs to prove that a theorem holds is quite convenient. As long as we can write a program term whose type is that of our theorem, we can prove that our theorem holds. But as the theorems we wish to prove become longer and more complex, so do the programs that serve as our proofs. It is hard for humans to read computer programs that are long and complex, compared to source code that is more terse. In proofs especially, it would be useful if we could abstract away from details that aren't necessarily relevant, and could let these be filled in automatically.

This is where *proof tactics* come in. Tactics are, at their core, small programs that transform the proof state in one way or another. Some tactics may be able to prove a goal, in which case that goal is removed from the list of unproven goals. This can happen when a goal is relatively simple; a nice example of such a goal is one where a variable of its type is already in scope, which means that variable can just be filled in by the tactic (in Coq this tactic is called `assumption`). Tactics can also take an existing goal, and replace it with one or more subgoals. A good example of this scenario is a value whose type is a data type with several constructors. If we wish to match on this value and provide a proof for each possible constructor, we could use the `destruct` tactic. It will replace the current goal with goals for each possible constructor, thus allowing us to run a different tactic for each possible branch that might exist.

Tactics allow us to abstract away complex operations, and allow us to remain focused on the the relevant parts of proving a goal. We can specify the operations that should happen, and the tactics will attempt to transform the goal. Sometimes tactics might not be applicable, in which case they will either do nothing, or they might produce an error. Combining tactics in various ways makes for a good way to solve many goals. It is a quite powerful paradigm that enables proof authors to prove their theorems while at the same time keeping the proof script itself short and readable. This makes it easier for others (including the original author, should they revisit their proof later) to read and understand the proof.

A final point to note is that proof development using tactics can be interactive when using some proof assistants. For example, Coq allows a proof author to step through their proof, inspect the proof state at any point. Here, Coq will display which goals are currently unproven, and display the context of variables that are in scope. Using this information, the author can see how the proof state has changed, and can decide which tactic to apply next. This is particularly helpful when a sequence of tactics is not working as expected.

2.3 Agda

In this subsection we will cover some of the basics of the Agda programming language, which is largely the subject of this thesis. Agda's type system supports dependent typing, making it also suitable as a proof assistant. As such, we will give a small overview of dependent typing.

2.3.1 Dependent types

Many statically typed programming languages have support for type polymorphism. This means that instead of defining a single type, we can define a whole family of types. A common example are collection data structures, such as the ubiquitous list. Instead of defining a type `IntegerList` and a `BooleanList`, we can simply define the type `List a`, where `a` is a type that can be filled in later. We can then declare values of type `List Integer` if we need a list of integers, and the compiler will abstract away the implementation details. Parameterised types have existed for quite a while in programming languages. Various languages such as ML, CLU and Ada have supported some notion of type polymorphism [16]. Since then, many programming languages have adopted type polymorphism, some of these being languages that are in widespread use [17], [18].

Dependent types can take this a step further. 'Ordinary' parameterised types can only take other types as arguments. On the contrary, dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data [19]. In other words, we can define polymorphic types whose member

```
doubleVector : {n : ℕ} → Vector A n → Vector A (n + n)
doubleVector as = as ++ as
```

Listing 2: A function in Agda that 'doubles' a vector.

values are explicitly related to the data that is used to instantiate that type. Dependent types can thus blur the distinction between types and values.

A common example of a use of dependent type is the vector type. This type is essentially a list type, but the length of this list is known and encoded in its type. A simplified vector type definition would look like `Vector A n`, where `A` is the element type of the vector, and `n` is a natural number that expresses how long the vector is. This definition is not just a single type, but it is a family of related types. We can now specify how long a list is at the type level. As an example, the type `Vector Boolean 3` specifies that we have a vector of exactly three Boolean values. As type definitions become more precise, so too does our knowledge of what values may inhabit these types, allowing us to rule out entire classes of bugs at compile time.

But we could go even further than that. Let's take a look at listing 2, where we define a function in Agda named `doubleVector` that 'doubles' a vector. In practice this means that we simply concatenate the vector to itself. However, the function's implementation is not what we wish to focus on here; the type is far more interesting. In the type, we first declare an implicit parameter `n` whose type is a natural number. Because the parameter is implicit, we usually do not need to fill it in manually as the compiler can infer it automatically. Next, we indicate that our function accepts a value of type `Vector A n`, where `A` is a type parameter of the vector's elements, and `n` is a natural number representing the length of the vector. Finally, we state that the return value of the function will be a vector of the same type of elements, but it will be exactly twice as long. We can indicate this to the compiler by using the variable `n` again. The Agda compiler will verify this property, and type checking of the function implementation will succeed if and only if Agda can prove that the return value has this expected length. Buggy implementations that produce vectors of an incorrect length are thus ruled out at compile time. Rich type annotations can be an effective tool for preventing faulty programs.

2.3.2 The Agda language

We have already seen a little bit of Agda code in listing 2, and on the surface it looks somewhat like Haskell code. Indeed, its syntax is quite similar to what we are used to, coming from Haskell. Agda is far more permissive about variable names, however, allowing most characters to be used in names, even Unicode characters [6, Lexical Stucture]. But mere looks can be deceiving. Agda incorporates many of the ideas that we have discussed so far (and more). Its type system fully supports dependent types. This allows Agda to be used as a tool for mechanical proving [19]. Through dependent types, we can pick and choose parts of our software system that we would like to verify. In these places, we can make our types more precise so that the compiler can verify the behaviour we expect from the function.

To ensure that Agda can adequately type-check expressions during compilation, several restrictions are imposed on the definitions in Agda: definitions must be *total*. Arbitrary expressions might not produce a value, or their computations might not terminate. Therefore not all expressions are admitted by the compiler. As a result, we know that Agda programs are safe to evaluate, and by 'safe' we mean that programs are guaranteed to produce a value within finite time. Totality is achieved by a number of checks that the Agda compiler performs during each compilation. It guarantees that expressions are terminating, it verifies that all definitions are complete, and it checks that definitions of data structures are strictly positive.

- Termination checking refers to recursive functions, and Agda will only accept recursive schemas that it can mechanically prove to be terminating. In practice this means that only obviously terminating definitions are accepted. Some definitions may be terminating, but if Agda cannot prove it, they are rejected. It is possible to turn off termination checking for individual definitions using a pragma (`TERMINATING`) which essentially asks Agda to just trust us. There is also an alternative

pragma (`NON_TERMINATING`) that will also turn off the termination checker, but sets a flag which causes Agda to not reduce that function during type checking. This flag can be considered to be slightly safer, as it still admits non-terminating programs but prevents them from being evaluated during type checking.

- Completeness refers to function definitions that employ pattern matching. Agda will require code authors to handle every possible case that might occur. This is important, so that Agda programs never run into a case where the desired behaviour is unspecified. Suppose we have a function that accepts a Boolean value as its argument, but only has a case for the *false* value; the *true* value will trigger the undesired undefined behaviour. This is an example of a partial function. The completeness checker will find partial functions and mark them as errors.
- Strict positivity checking applies to data types that are defined in Agda. To paraphrase from the Agda documentation, this restriction means that every constructor of a data type A may only have parameters that are non-inductive (and thus do not mention A at all) or are inductive. Inductive parameters may not have function types that use A in any of their parameters. If such definitions were allowed, these constructors might result in programs that do not terminate. This check can be disabled with the pragma `NO_POSITIVITY_CHECK`.

Agda has the notion of *holes*. If we are writing a program in Agda, we might not be able to give a finished program yet. So instead, we can place a hole there. These indicate a missing part of the source code, and Agda will try to type-check around it. If the editor that is used to edit the code supports Agda, it will be able to highlight the holes and provide the user with relevant commands that may be executed in the context of the hole. For example, a programmer may inspect the hole, see the type of the term expected there, or see which bindings exist in the context. There are also editor commands to refine the hole, or even to completely solve it in some cases. The automatic solver is adequate for some simple cases, but cannot always handle more complex cases and is not extensible.

The holes and editor commands form the basis for interactive proof development. Each individual unsolved hole is akin to a goal that must be proven in Coq. When we view the context and expected type of the hole, we are essentially looking at the variables that are in scope and the goal that this hole represents, which Coq also allows us to inspect. If we have come up with a term that might fit in there, which could even be a term with another hole, we can type that term in the hole and ask Agda to *give* the term using an editor command. It will type-check the term in the hole against the hole's type and context, and if it matches, the hole will be replaced by the term. Besides typing terms manually, we can also define macros. These are Agda programs that can be run from the context of a hole, and could then generate a term that fits inside the hole. This will be discussed in the next section.

Agda has both explicit and implicit parameters. Explicit parameters are the default and do not need special syntax, but can be denoted by parentheses `( )`. Implicit parameters must always be indicated by curly braces `{ }`. Implicit parameters often do not need to be filled in, as the compiler can infer them. They are also hidden in matches. They can be made explicit by surrounding an argument or match pattern with curly braces.

Function calls in Agda do not necessarily look like function calls. Depending on the function name, it may also be used as a mixfix operator [6, Mixfix Operators]. A nice example is the cons operator for lists, whose full name is `_::_`, but is used like `h :: t` (though `_::_ h t` is also allowed, but is unconventional and unexpected). Underscores in names mean that the name can be split up into multiple parts, with each underscore representing a parameter. For example, the function `if_then_else_` (which unsurprisingly is an if-else expression) is used like `if c then t else f`. In this document we will typically refer to these functions by their full name, including any underscores.

2.3.3 Unprovable theorems

In listing 1 we already gave some examples of the Curry–Howard correspondence, showing how types can encode theorems, and how values that are members of this type (also referred to as inhabitants of a type) serve as proof that the theorem holds. But with Agda we can express more interesting types, which thus allow us to define more interesting theorems.

A selection of these theories are shown in listing 3, and we will go through them here. We first define the

```agda
data ⊥ : Set where
  -- No constructors; this Empty type cannot be instantiated.

data ⊤ : Set where
  -- This Unit type only has the trivial tt constructor.
  tt : ⊤

NonZero : ℕ → Set
NonZero zero = ⊥
NonZero (suc _) = ⊤

1IsNonZero : NonZero 1 -- becomes ⊤
1IsNonZero = {! hole 1  !}

0IsNonZero : NonZero 0 -- becomes ⊥ ; unprovable
0IsNonZero = {! hole 2  !}

¬_ : Set → Set
¬ A = A → ⊥

¬¬-intro : ∀ {A} → A → ¬ ¬ A
¬¬-intro a = λ ¬z → ¬z a

¬¬-elim : ∀ {A} → ¬ ¬ A → A
¬¬-elim = {! hole 3  !}
```

Listing 3: A showcase of some theorems in Agda. The `⊤` (unit) type was adapted Agda's built-in definitions, and the types `⊥`, `NonZero`, and `¬_` are taken from the standard library. The parts of code marked in green are holes that have not yet been solved.

empty type `⊥` and the unit type `⊤`. The empty type cannot be instantiated, and can thus be used to create contradictions. The unit type can always be constructed, and its construction is trivial.

Our next step is the `NonZero` function. This function takes a natural number as its single argument. If the number is zero, it will return `⊥`, and if the number has any other value (is non-zero), it will return the type `⊤`. In practice this means that we can only prove `NonZero x` for any `x` that is actually non-zero, as it is becomes impossible to construct a proof when `x` is zero. This is illustrated by the definitions `1IsNonZero` and `0IsNonZero`: the first one can be proved, but the latter one cannot. During type checking Agda automatically reduces the type `NonZero 1` to the `⊤` type, and `NonZero 0` gets reduced to `⊥`. Agda will also infer that *hole 1* can be filled with the term `tt`, and *hole 2* cannot be filled because no term inhabits the `⊥` type.

The intuitionistic logic system that Agda is built on can also express negation in this way. We have defined the function `¬_`, which accepts a type `A` and then returns a function type. This function type expresses that if we have a term of type `A`, we get a contradiction as we will need to instantiate the empty type, which is impossible. We will call this function the *negation type*.

We will use this type to show how Agda's logic differs from classical logic: double negation cannot be eliminated. In classical logic, we can say that 'not true' is 'false', and that 'not not true' is 'true'. We have thus eliminated the double negation. This property does not hold for propositions in Agda: while we can introduce double negatives, we cannot eliminate them. The double negative introduction is expressed by `¬¬-intro`. It expresses that if we have a value of type `A`, the existence of a theorem showing that an instance of `A` exists leads to a contradiction would itself be a contradiction. But proving the double negation elimination is impossible: the existence of a theorem showing that `A` leads to a contradiction itself being a contradiction does not count as proof of `A`. The *law of the excluded middle* ($p \vee \neg p$) does not hold in Agda (and intuitionistic logic in general) [20, Negation].

2.3.4 Tactics as reflection programs

Agda has no native support for tactics. All proof terms must thus be fully written out for the compiler to accept. This is typically not a deal breaker, since Agda is a fully featured functional programming language with an extensive official standard library. Implicit arguments can hide much of the complexity that is typically expected from proof programs, and Agda can often infer their values automatically. Regardless, writing full program terms is not as convenient as combining a number of tactics that automatically expand to the full terms.

In place of tactics, Agda has a powerful reflection API which exposes compiler services. In the context of computer programming, 'reflection' often refers to introspection; the ability of a program to examine its own operations and structures [21], as well as the ability to modify these operations and structures. The reflection API would be a good candidate to implement tactics, as we could use it to generate terms that serve as proofs. Agda's reflection API is powered by the compiler, and can thus not be used at run-time. This is not a problem for the purpose of implementing tactics, since all proof terms must be fully expanded at compile-time already. As such, producing a executable binary program is not usually required when authoring proofs in Agda, since successfully type checking the code is enough to verify that the theories expressed by the code are proven.

The compiler-based interface presents all the functionalities needed to inspect and generate syntax terms and types. It also exposes a number of structures that are internal to the compiler, and although they are largely abstracted away from the surface language as implementation details, these can be accessed through the reflection services. The terms generated using this API are always syntactically well-formed, since the reflection programs do not operate on a textual representation of the code. Instead, all operations operate on data types provided by the compiler, which are used to build syntax trees.

To run these reflection programs, they must be written as so-called *macros*[1]. These macros are unlike those found in most other languages, which operate by text replacement (such as in the C programming language [7]) or even as hygienic macros that do not capture variables [8]. Instead they are regular Agda functions that can perform any computation that can be computed in Agda (that is, they should obey the restrictions that Agda imposes). By marking this function as a macro (done by by defining them inside a `macro` block in the program source code), Agda can then use these functions to manipulate other elements of the source code.

The type of macro functions must be $t_1 \to t_2 \to \cdots \to t_n \to \texttt{Term} \to \texttt{TC}\top$. This means that the macro can take an arbitrary number of arguments, followed by a value of type `Term` that represents the hole, and produces a value of type `TC ⊤` as its output. This output value is the sequence of instructions that the compiler will execute in the context of the type checker when the program is run.

Holes in the Agda source code correspond to *metavariables* that Agda keeps track of internally. Such metavariables are used to keep track of terms that aren't fully known yet. In Agda parlance, these are also known as *metas* (singular *meta*). We can solve a metavariable using the `unify` function that the Agda reflection API exposes. The function takes two arguments, possibly containing metavariables, and Agda will try to resolve them. If this is not possible, a type error is raised. We will be creating and using metavariables to construct terms that are not fully known yet. We should be careful however, as the existence of metavariables that are unsolved will cause Agda to raise warnings that prevent compilation from fully succeeding.

2.3.5 Universe polymorphism

In Agda types can be used as terms. This means that type expressions must themselves have a type. These types are called *sorts* (also known as universes). A sort is a type whose members are types. The basic sort in Agda is named `Set`, and it contains all the small types. The set of small types is essentially the set of all types, except for `Set` itself. This avoids the typing rule `Set : Set` which would lead to a logical contradiction similar to Russell's paradox [6], [22].

This still leaves the question of what the type of `Set` should be. To avoid the contradiction, we can define a new sort which contains the basic sorts, which we shall call $\texttt{Set}_1$. And then the type of this sort

[1] There exist other ways to run metaprograms, but these are used for reflection programs that do not generate terms. Instead, these methods are used to generate entire definitions.

will in turn be `Set₂`. We thus construct an infinite hierarchy of sorts, such that each sort is a member of the next higher sort. To make working with these different universes easier, Agda has introduced the primitive `Level` type, which represents a universe. By defining data structures to be polymorphic on this type, it is possible to let these data structures work with all possible levels (universes) of data. The most notable example is `Set` itself, which may take an argument of type `Level`. Polymorphic sorts look like `Set ℓ` in Agda, where `ℓ` is the universe level argument.

The data structures and associated operations we define later in this document are almost all universe polymorphic (there are some exceptions where polymorphism cannot be applied or is not needed). Due to the design of certain structures, where types exist as fields within records, care must (and has) be taken to ensure that levels are correctly specified.

3 Iterators

This section describes an implementation of iterators in Agda, allowing for the generation of values that continues either finitely or infinitely, while still allowing for interesting operations that are not possible on structures from the standard library. Iterators are also used as the underlying data structure for the library design that we describe in section 4 and section 5, allowing for flexibility in backtracking. We will first give a case for iterators, then proceed with the overall design, follow that up with some useful operations that have been implemented for iterators, and finish this section with a selection of unsuitable iterator designs.

3.1 The case for iterators

When finding a solution to a problem, a common scenario is having to choose from a range of possible values. This range can either be finite ('try these constructors') or infinite ('pick a natural number'). It would thus be great if we could use a data structure that allows us to express both of these cases. Such a data structure should allow us to transform, filter, and combine them, so that we can generate and select the data we want.

In some languages, such as Haskell, this is fairly straightforward. We can simply define a list expression by typing `[1, 2, 3]` to create a list that contains exactly, in order, the integer numbers 1, 2, and 3. To create an infinite list, we can simply write `[0..]`, which will give us an endless list that contains all natural numbers. This works, because Haskell maintains lists as a head and a tail, and due to its nature of lazy evaluation, the infinite tail is only evaluated one number at a time, when needed. Haskell also allows us to perform all kinds of interesting operations on these lists, allowing us to consume the values contained therein, or map to new (possibly infinite) lists. This is a simple and easy-to-understand way of dealing with such streams of values.

Agda does not have syntax like `[n..]` to create infinite lists by default, so let us see how the Glasgow Haskell Compiler implements it. For simplicity we will only describe the case for integers here. The expression is transformed into the expression `enumFrom n`. The implementation of `enumFrom` and a helper function is given in listing 4. The helper function, which takes a starting number and a delta to add in each successive list item, constructs the actual list. The head of the list is the starting number, and the tail is a recursive call with the delta added to the starting number.

```
enumFrom x = enumDeltaInteger x 1

enumDeltaInteger :: Integer -> Integer -> [Integer]
enumDeltaInteger x d = (x : enumDeltaInteger (x + d) d)
```

Listing 4: A simplified implementation of `[n..]`, adapted from the GHC source code [23, Enum.hs].

```
nats = 0 : map (+1) nats
```

Listing 5: An alternative definition of all naturals in Haskell.

This implementation is very similar to the alternative definition of natural numbers given in listing 5. Here, we define a list where the first item is 0, and the accompanying tail is a recursive expression which maps that same list using an incrementing function. Thus, every next value is one higher than the number preceding it.

Unfortunately this approach cannot be used in Agda. Although Agda's built-in `List` data type is perfectly adequate for a finite number of items, it is not capable of storing an infinite amount of them. Both Haskell and Agda employ lazy evaluation, so both are able to store infinite lists, because the tails are not evaluated; the tails are *thunks*. However, having an infinite list involves a recursive definition. Unless Agda can prove that the recursion is structural, termination checking prevents the code from being compiled [6, Termination Checking]. The recursive definitions we have seen to build the infinite lists do not

```
function* natsFactory() {
    let i = 0
    while (true) {
        yield i
        i = i + 1
    }
}

const nats = natsFactory();
nats.next() // 0
nats.next() // 1
nats.next() // 2
// ...
```

Listing 6: An example generator function in JavaScript enumerating all natural numbers in order. The function `natsFactory` produces a new generator object when called. This generator object has a `next` function, which will evaluate the next value in the generator and return it.

use structural recursion. Instead, these definitions make use of generative recursion, and are thus not allowed in Agda, as the compiler cannot prove that these expressions terminate.

Of course, the built-in `List` is not the only data type we have at our disposal. The standard library has many types from which we may choose a suitable one. And indeed, the standard library has a module named *Codata* which houses a collection of coinductive data types. Coinductive data types can be infinite. Of particular interest are the data types `Colist` and `Stream`, as these can represent the ranges of values we would like to examine and search through.

- `Colist` is a data type that is similar to the built-in `List` type, and also features many of the same operations. It can be either finite or infinite, because in this version of the list data structure, the tail is lazily evaluated. However, it does not have all operations that regular lists have. Most notably the `filter` function is missing. This omission is reasonable, because depending on the filtering predicate, there may not be a next item that satisfies it, making an implementation in Agda impossible.
- `Stream` is a data type that represents an infinite stream of items. This description already reveals a problem: it cannot represent a finite number of items. Additionally, it suffers from the same problems that `Colist` also has: it does not support all operations that the regular list does.

Similar structures also exist in other languages. For example, the `Array` type in JavaScript cannot contain an infinite number of items. JavaScript is an eagerly evaluated language, so having an infinite number of items would crash the runtime. However, modern versions of JavaScript have the concept of *generator functions*. Generator functions are indicated by `function*` (note the asterisk) and produce *generator objects* [24]. See listing 6 for an example.

While the generator function in listing 6 produces an infinite generator, it can also be finite. The concept of generators extends to other languages like Python where they are called generators as well [25], but also C# and Rust where these objects are called *iterators* [26], [27]. Literature on iterators predates all of these languages, and describes them in the Alphard and Icon languages [28], [29]. They also exist in the CLU language, but that implementation was based on Alphard [30]. In the rest of this document, we will refer to these objects as iterators, unless we mention them in the context of a language which calls them generators. In literature where 'objects that can produce several values' are first described, they are mostly referred to as 'generators', and 'iterators' iterate over generators. However, due to naming conventions in more modern languages that we are more familiar with, the name iterator stuck.

Each of these iterators, regardless of the ecosystem they are implemented in, can produce either a finite or infinite number of output values. The example shown earlier in listing 6 will produce all natural numbers, and thus is infinite. But we can also define iterators that output a finite number of values. Some languages also support interesting operations on their iterators. For example, C# supports mapping, fil-

```
private
  variable
    ℓa ℓs : Level

data Step (A : Set ℓa) {S : Set ℓs} : Set (ℓa ⊔ ℓs) where
  Done : Step A
  Skip : S → Step A
  Yield : S → A → Step A

record Iterator (A : Set ℓa) : Set (ℓa ⊔ (suc ℓs)) where
  constructor iterator
  field
    {S₀} : Set ℓs
    bound : Bound
    state : S₀
    next : S₀ → Step A {S₀}
```

Listing 7: The definition of `Step` and `Iterator`.

tering, grouping, and many other operations on iterators through functions in the .NET standard library [31].

3.2 Designing iterators in Agda

The design of iterators in Agda has gone through several iterations. This section describes the one that was considered good enough for use in the library as described in section 5.

If we examine the earlier example with JavaScript's generator functions (listing 6) we could wonder how a JavaScript runtime would implement this iterator behind-the-scenes. Fortunately, the abstract procedure is described in the ECMAScript specification [24]. Internally, the runtime builds a state machine that stores the context in which the function is executing. Each time the `next` function is called, this state machine resumes unwinding the iterator where it was last suspended, based on the state that the runtime maintains internally. If the generator function returns (which indicates that it has finished), the runtime moves to a state where no further evaluation happens. What's more, this `next` function is always the same and immutable; the behaviour of the function depends solely on the internal state. As a positive consequence, the function can be passed around without having to worry about maintaining state.

This implementation is strikingly similar to the Alphard language, which defines generators as *forms* (which are akin to a specification and implementation) that can generate values. In fact, it has a function `&init` which is used to construct the initial state of the generator. It also has a `&next` function, which performs the next step of the computation, based on the previous value. The language has built-in support for iterators, which can step through the generators and consume their values. These iteration constructs are transformed into while-loops during compilation, where the compiler uses inaccessible hidden variables to manage the control flow.

This is a design which can serve well as an inspiration for Agda iterators. A similar approach has already been built for Haskell [32]. This Haskell version is originally based on iterators as they are found in the Rust programming language. It was subsequently optimised to perform faster in the Glasgow Haskell Compiler using the results from [33]. This optimised version of the iterator works very well as a starting point for building iterators in Agda; not because it performs very well in terms of runtime, but because it has a property that the other implementations do not have. This property is that trying to retrieve the next value will always output a result, even if that value does not exist. This is accomplished using a data type which can express a lack of a value. In other words, this iterator is guaranteed to be productive, which is a requirement in Agda. This property means that we will be able to implement operations that are common for data structures that represent sequences. In later sections we will explore this further.

3.2.1 First steps

Let us start defining our iterators by first defining a supplementary type: `Step` (shown in listing 7). This type will represent the output that the iterator will be producing. `Step` has two type parameters: one that that determines the output values that the step can carry, and the other is the type of the output state that the step can hold. It has three constructors, all with a different purpose.

- `Done`. This output value indicates that the iterator has finished. A consumer of the iterator should not attempt to obtain any more values.
- `Skip`. This output value indicates that there is no value here in the iterator, but that more values can be obtained from the iterator. A consumer of the iterator should attempt to continue iterating. Its single parameter is the output state of the iterator, which must be used to continue iterating.
- `Yield`. This output value signals that the iterator has yielded an output value. The constructor carries both the next state of the iterator which must be used in the next iteration, as well as the output value that can be used by a consumer of the iterator.

But why have the `Skip` value? What is the point of indicating that a value does not exist? Surely it would be possible to continue iterating until a suitable value is found? Unfortunately (or fortunately, depending on the point of view) this is not as easy in Agda. Suppose we have an iterator that enumerates all naturals in order, and we filter it with a predicate that is satisfied if and only if the number is smaller than 10. Now, if we decided to retrieve the first 11 results of this iterator, we would loop endlessly as an 11th number could never be found. Agda does not allow this under normal circumstances, and this is what the `Skip` result is useful for: it can tell us that there is no value when the filter predicate is not satisfied, but that a satisfactory value may still exist in future iterations.

3.2.2 The `Iterator` record

Now let us take a look at the definition of the `Iterator` record (listing 7). It has two implicit parameters (the universe level parameters, declared in the `variable` block), one explicit parameter (the output type), and four fields that we will describe now.

- S_0. The type of this iterator's state. This field is implicit, and is thus not specified in the automatically generated record constructor. Instead, the value is automatically inferred from the initial state value's type. The state is encapsulated inside the record, and not very important to the outside world.
- `bound`. The upper bound on the number of items this iterator can yield. This value can be used as a hint to iterator consumers, signalling how far an iterator must be unrolled to compute all possible values.

  The bound is represented by the `Bound` data structure in listing 8. It has two constructors; one to indicate a concrete finite bound, and one to indicate an infinite bound. The constructor `unbounded` indicates that the iterator potentially has an unbounded number of values it can produce. An example of such an iterator would be one that yields all natural numbers. The second constructor, `bounded`, signals the maximum number of attempts a consumer should make at obtaining a value from the iterator. Standard arithmetic functions (addition, monus, multiplication, maximum, minimum) are provided for this type.

```
open import Data.Nat using (ℕ)

data Bound : Set where
  unbounded : Bound
  bounded : ℕ → Bound
```

Listing 8: The definition of the `Bound` type that is used for iterators.

It is expected that functions that create new `Iterator` instances will set this value to an appropriate value. More information and concrete examples of this behaviour will be given in the section 3.3. Some functions that produce iterators may use relatively slow computations to calculate what the bound of the iterator will eventually be. However, in most cases this is perfectly fine due to Agda's lazy evaluation. As a result, many of these values will remain as thunks.

- `state`. The initial state of the iterator. It is possible to replace the state of an existing iterator to essentially drop the first items up until this new state.
- `next`. The function that accepts an iterator state, and uses that state to generate the next step of the iterator. This function decides whether the iterator will be done, skips the current iteration, or produces a new value.

Every iterator has a field containing the type of its state. This design choice was made because an iterator can now describe itself, which allows for better encapsulation. An outside consumer of the iterator does not need to be concerned about the type of the state, and must treat it as a black box. This design does come with some consequences that must be kept in mind. Because the state type S_0 is contained within the iterator, the iterator's level must be at least the successor of the level of the value of S_0. Thus, the level of an iterator is either the level of the output value, or the successor of the level of the value in S_0, whichever is higher. Wrapping an iterator record inside of another data structure means that this wrapping structure must accommodate the level change. Additionally, when an `Iterator` type is described, use sites must also ensure that the level of the state type is declared. This makes using iterators slightly more cumbersome than they ideally would be, but this is still better than an alternative design proposed in section 3.5.

3.2.3 Using iterators

Together, the fields `state` and `next` can emulate the JavaScript generator functions, or more broadly, the iterator concept as found in many other languages [26]–[29]. Obtaining values from an iterator requires some more bookkeeping by the consumer, however. Values in Agda are immutable, so it is not possible to modify the state of an iterator through the mutation of hidden variables.

Instead, consumers of an iterator can apply the `next` function to a state value with the corresponding type. In the first iteration, this state value is the `state` field inside the iterator. The output value of the `next` function is a `Step`. This step signals to the consumer which actions can be taken next. If the iterator is done, the consumer should not attempt to retrieve more values from the iterator. If the iterator signals a skip, it includes the next state to use as input to the `next` function. And finally, if the iterator yields an output value, it includes both the next state, as well as the output value. Thus, *unrolling* the iterator requires the use site to perform some work that would usually be done transparently behind the scenes in other languages.

In practice, this is not too much of an annoyance when it comes to unrolling an iterator. Having the distinct constructors from the `Step` type requires a consumer to handle all cases that may occur when dealing with iterators. Depending on that specific implementation, a skip might invoke specific behaviour to handle a missing value. So although passing on the state value is still mandatory in the current implementation, the final implementation remains legible and easy to understand.

This implementation of iterators fulfils both of the requirements that the data structures we mentioned before cannot fulfil. Because iterators can signal that they are done, and their values are evaluated on-demand, they can represent sequences that are finite in length, but infinite sequences can also be created. The onus is on the consumer of the iterator to ensure that the computation is finite, to comply with Agda's termination rules. Iterators also support most operations that are desired for data structures that represent sequences; these are detailed in section 3.3.

Let us finish this section with an example of an iterator. So far, we have referred to an iterator that generates all natural numbers numerous times, and an example of an implementation in JavaScript was also shown. It is high time we show the implementation in Agda in listing 9. This iterator is unbounded, and produces values of type $\mathbb{N}$, the type of natural numbers in Agda. The state of the iterator is that also the type of natural numbers. Our initial state value is the number `0`. The iterator always yields a value; the

```
nats : Iterator ℕ
nats = iterator unbounded 0 next
  where
    next : ℕ → Step ℕ
    next i = Yield (suc i) i
```

Listing 9: An Agda iterator that enumerates all natural numbers.

```
empty : Iterator A
single : A → Iterator A
fromList : List A → Iterator A
toList : Iterator A → ℕ → (List A)
map : (A → B) → Iterator A → Iterator B
filter : (A → Bool) → Iterator A → Iterator A

_∷_ : A → Iterator A → Iterator A
_++_ : Iterator A → Iterator A → Iterator A
concat : Iterator (Iterator A) → Iterator A
interleave : Iterator A → Iterator A → Iterator A
```

Listing 10: Simplified types of basic operators and combining functions for iterators.

`next` function returns the input state number as its output value, and the accompanying next state is the successor of this input number.

3.3 Operations

As mentioned before, one of the core reasons that iterators were implemented in Agda is because they support some operations that the existing data structures do not (and in many cases, *cannot* support). In this section we will give an overview of some of these operations, and how they are implemented. Some of these operations are relatively straightforward, while others are more complex. We will start with the simpler operations, and then build our way to the more interesting ones. A simplified version of their types are given in listing 10. Many of these functions are also described in [33], although we were not aware of this work at the time of implementation.

First however, it would be useful to describe some of the utility functions that exist for iterators. These are used throughout some of the other operations. The first utility function is the `empty` function. It creates an iterator whose next function will unconditionally signal that the iterator is done, thus yielding no items. The type of its state is a level-polymorphic unit type whose level is automatically inferred by the compiler, although variations of this function exist that explicitly fill in this level if it cannot be inferred. A second function is the `single` function, which accepts a value of type `A`, and then returns an iterator which produces that exact value once, and will then indicate that it is done. This iterator stores a value of type `Maybe A` as its state. If this optional value is set, `next` will yield that value, along with an empty optional state value. If it does not contain a value, the iterator will signal that it is done. The correct bounds are set for both `empty` and `single`, to zero and one respectively.

To make iterators easier to use, utility functions that can convert from and to standard lists have also been implemented. The most notable function here is the `fromList` function, which creates a new iterator that yields exactly the items in the list, without skips, and then signals that it is done. The initial state of the iterator is the given list. On every call to `next`, it takes the head of the list and yields it, setting the next state to the tail of the list. Thus, as the iterator is unrolled, each item of the list is returned. Similarly, there is a `toList` function which behaves exactly as its name implies: it turns an iterator into a list of values. To prevent nontermination, a depth parameter is required, specifying how far the iterator can be unrolled. Each yielded value is collected into the resulting list.

Let us now look at the `map` and `filter` functions, starting with `map`. This function accepts two para-

meters; a function `f` of type `A → B`, as well as a value of type `Iterator A` which we'll call iterator A. It will return a value of type `Iterator B`, which we'll call iterator B. This iterator is almost the same as the input iterator: the bound and state (and thus state type) are kept the same. The only difference is in the `next` function. On every iteration, iterator B simply calls the `next` function from iterator A, and discriminates on the result as follows:

- If iterator A produces `Done`, then iterator B returns a `Done` as well.
- If iterator A produces `Skip`, then iterator B returns a `Skip` with the unmodified state value.
- If iterator A produces `Yield`, then iterator B will also yield. The state is unmodified, but the value contained within is mapped using the function `f`.

Thus, `map` produces a new iterator whose behaviour is only different when the original iterator yields, by mapping the yielded value internally. The `filter` function works roughly the same, in the sense that it only changes the behaviour of the resulting iterator if the original iterator yields. Instead of a mapping function, `filter` accepts a predicate function of type `A → Boolean` that decides which elements to keep and which to toss. Thus, each time iterator A produces a `Yield`, the predicate function is applied to the value. If the predicate holds, a `Yield` is produced. If the predicate fails, a `Skip` will be generated instead, to indicate the lack of a value.

3.3.1 Combining operations

In Agda, items can be prepended to a list using the `_::_` constructor. A similar operation is implemented for iterators. It accepts two arguments, of type `A` and `Iterator A`. Internally, it will construct a new iterator that will first yield the single value that is to be prepended, and will then defer to the iterator to produce new steps. Although this behaviour is unlike the standard list, where prepending is the primary way of constructing the list, the end result is the same when a consumer unrolls the iterator.

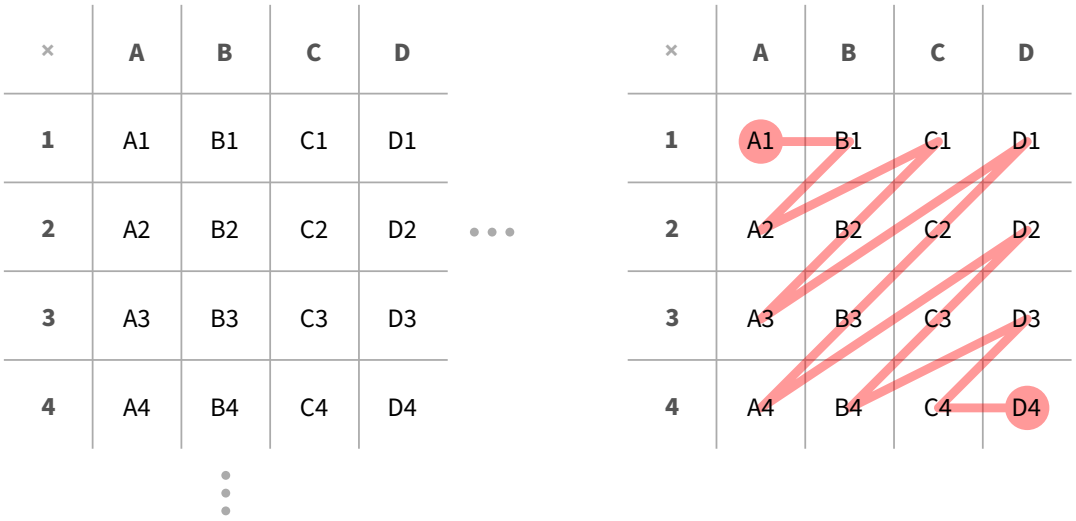
This concept can be generalised to the append function, which typically appends two lists and is denoted by `_++_`. For the iterator variant, it takes two iterators of type `Iterator A`, and creates a new `Iterator A` in which both iterators are appended. Internally, this new iterator uses a sum type for its state, which can contain either the state of the first iterator, or that of the second iterator. It will first defer its behaviour to the first iterator until it is done, and it will then switch to unrolling the second iterator. Naturally, if the first iterator is infinite, the second iterator will never be used.

In turn, the append function can be more generalised so that we get the `concat` function. This function concatenates a list of lists (all of the same type) into a single list. It would be convenient if we could do the same thing with iterators: combining an iterator of iterators into a single iterator. For this implementation, more bookkeeping is required within the state structure. We need to distinguish two iterators here: the *outer iterator*, which is the iterator of iterators; and the *inner iterator*, which is the iterator that was last produced by the outer iterator. The resulting concatenated iterator maintains the state of both iterators, so that it can unroll the inner iterator and return its results, as well as generate a new inner iterator if the current one is finished. To prevent level escalations in Agda, the state record of `concat` does not store the inner iterator directly. Instead, only the state and next function are stored.

As mentioned earlier, the append function will never get to the second iterator if the first one never finishes enumerating items. There exist valid use cases where we would like to examine values from two different iterators, however. To accommodate this use case, an `interleave` function was developed. It accepts two parameters, both of type `Iterator A`, and creates a new iterator that first steps the first iterator, and then steps through the second iterator, in alternating fashion. If one iterator is done, it switches to the other iterator exclusively. If both are finished, it outputs a `Done` indicator. The interleaving operation can be used to combine two iterators in a fairer way.

3.3.2 Cartesian product

As a theoretical exercise, we have implemented a Cartesian operator that can compute the Cartesian product of `n` iterators. This operation produces a new iterator that iterates each tuple in the Cartesian product. In a Cartesian product between `n` sets, all possible `n`-tuples are enumerated (see fig. 2a for a two-dimensional example). The challenge here is to ensure that this works with the infinite sequences that can be represented by iterators.



(a) A Cartesian product of the alphabet and the natural numbers.

(b) Using diagonalisation to traverse the different elements of the Cartesian product (finite example).

Figure 2: An example of a Cartesian product, along with an example ordering in which all elements are traversed in a fair way.

When implementing a Cartesian product of potentially infinite sequences, it is important that the elements are enumerated fairly. Suppose that we desired a Cartesian product of the alphabet and the natural numbers (fig. 2a), it *might* be good enough to get the first letter A, and then simply produce the values A1, A2, A3, etc., depending on the use case. But if we have a more complicated example, such as $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$, and filter this iterator such that only the values (a, b, c) are kept if $a^2 + b^2 = c^2$ (i.e. they are a Pythagorean triple), we might never find a result because one value grows faster than the other.

To remedy this problem, and ensure that elements are enumerated fairly, a suitable traversal strategy is required. For our approach, we have chosen to employ a diagonalisation strategy. An example is given in fig. 2b. Here, we start with the element A1, then continue with B1, then B2, then C1, and so on. Both iterators are stepped through at roughly the same speed, so there is no 'starvation'. This traversal strategy is similar to an enumeration of rational numbers that is used to show that the rational numbers are countable [34]. We will use a generalised version of this approach to enumerate all possible combination of iterator elements.

Implementing this diagonalisation process requires some bookkeeping. Let us again use the earlier example of the alphabet and the natural numbers. Let us call the alphabet sequence `A`, and the sequence of natural numbers `N`. We could build the diagonalised Cartesian product for these two sequences iteratively. The steps for this process are as follows:

- We step both iterators, and append the new value to the list of values we have already obtained from the iterators. These lists are called `As` and `Ns`.
- We compute the reverse of `Ns`, which we shall call `Ns'`.
- Zip `As` with `Ns'` resulting in pairs of values.

Every time we perform this operation, we get a list of items in the Cartesian product we have not yet seen before. Their order corresponds to the diagonalisation strategy described before. See fig. 3 for an illustration. Note that each cycle in fig. 3 corresponds to a diagonal line in fig. 2b (the ones at a 45° angle, so e.g. from D1 to A4 corresponds to cycle 3).

Naturally, this example is two-dimensional, and it does not take into account the implementation details that come with iterators. The example here can be scaled up to support these use cases. For this purpose, the cycles in fig. 3 are denoted with numbers starting at zero. In the remainder of this section, we will use the convention that list indexing is also zero-based, i.e. the first element of a list exists at index 0.

Using this information, there is an observation we can make. In each cycle, the sum of the element's

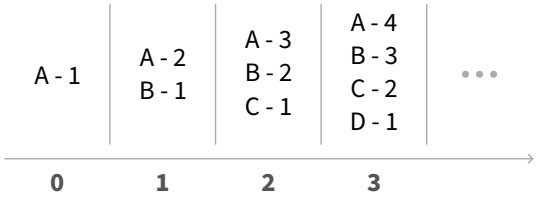


Figure 3: An illustration of an algorithm for diagonalisation in a two-dimensional Cartesian product. The numbers along the bottom indicate the cycle number, starting at zero.

indices is equal to the cycle number. The letter A and the number 1 both have index 0, so the index sum becomes 0, which is indeed the cycle number. The same goes for the elements A2 and B1, whose index sums are both 1. And for A3, B2 and C1 it holds that their index sums are equal to 2. This pattern is not limited to the two dimensions, and scales up well. Mathematically speaking, such a sequence of indices can be seen as a *composition* of the cycle number [35]. Specifically, it is a k-composition of `n`, where `k` is the dimension of the Cartesian product and `n` the number of the cycle.

We can also use this pattern to find the elements in the Cartesian product. For each cycle, we can generate all possible k-compositions for that cycle number, and use the components of the composition as indices to the values of the iterators. Internally, the Cartesian operator uses exactly this approach when generating the combined iterator. Just like the example, the iterator works in cycles. It keeps track of the current cycle using a simple counter variable, which is initialised to zero. We also store a list of compositions, whose associated elements should be returned next, in a list called the *work list*. The Cartesian iterator is evaluated lazily; the next computation is evaluated only if the iterator's `next` function is called. We will now describe how the `next` function operates. A visualisation is given in fig. 4. There are two cases here:

- The worklist is empty. This scenario is the start of a cycle. We must first step all the iterators, and then compute which elements will be returned in this cycle. So we indeed step through all iterators, and store the values they produce in a vector. Each input iterator has its own vector of optional values, which we use to store all elements it has produced. If the iterator yields a value, we use the `just` constructor to store it, and if it skips we use `nothing`. If the iterator is done, no value is appended. This is necessary, because eventually each element has to be combined with all elements from every other iterator. We then check whether new items can actually be returned; if every input iterator is done, eventually there are no more elements left to be produced. This produces another two cases:
  - No further items can be produced. The Cartesian iterator will return `Done`.
  - More items can be produced. A new work list is generated, containing all the k-compositions of the current cycle number. The cycle number is incremented by one. The iterator will then produce a `Skip` instruction, carrying this new state.
- The worklist has items. We consider this scenario to be the middle of a cycle. The work list is split into a head and tail. The head, which is a k-composition, will be used to to index the `k` vectors of items we have collected so far. This is done with a k-dimensional lookup function. We then decide whether every iterator yielded a value at the respective index (meaning that the optional value we looked up from the vector indeed carries a value). There are then two situations:
  - Each iterator yielded a value. We can then return the tuple of all elements, using the `Yield` constructor.
  - At least one iterator did not yield a value. The tuple cannot be created, because a value is missing. The iterator will return the `Skip` result.

  Regardless of the case, the tail of the work list is used in the next state. No other state values are updated. So we go through each of the compositions, until all compositions in this cycle have been explored.

There is one additional restriction that comes into play. One iterator may finish before another. This means that some possible k-compositions of the cycle number are invalid, because some of the indices

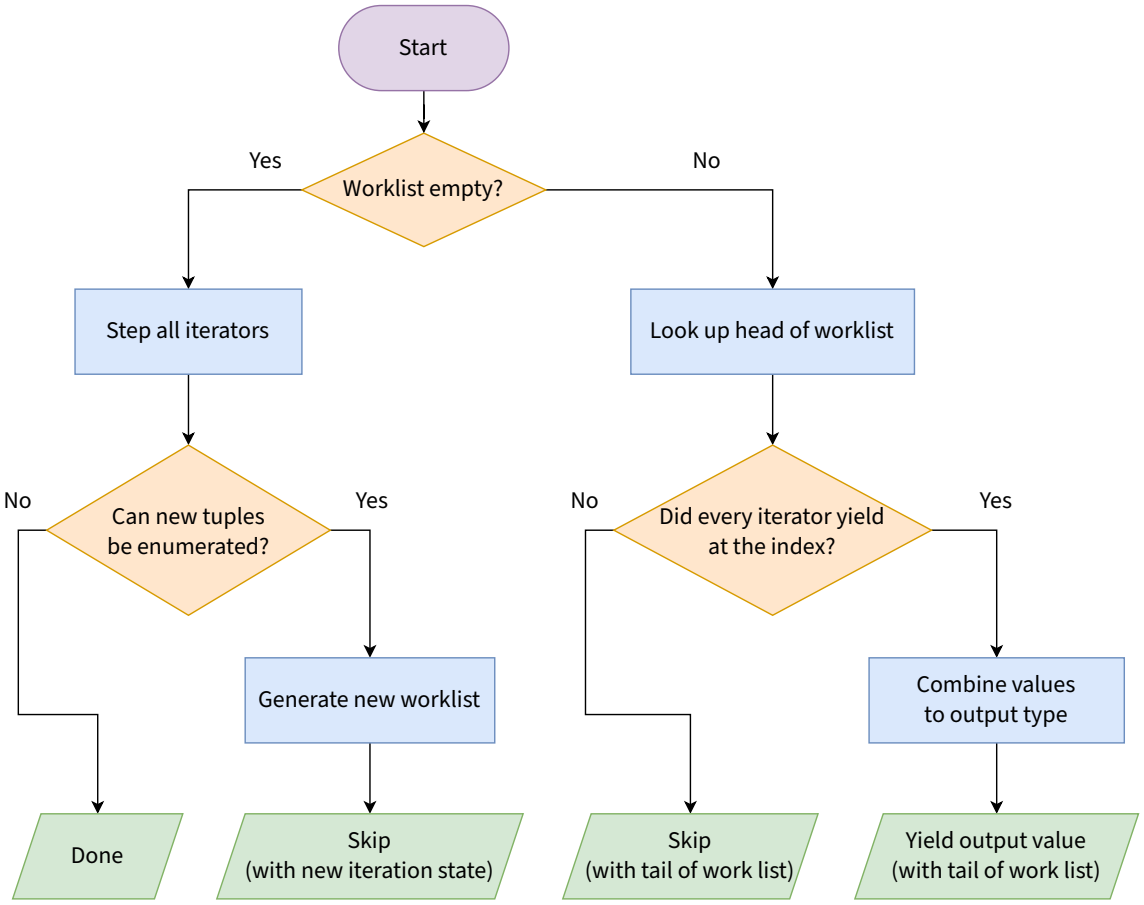


Figure 4: An illustration of the decisions that the Cartesian operation iterator makes.

their components represent may not actually exist in the corresponding element vectors. To prevent this from happening, the algorithm generating the work lists takes the length of these vectors into account when generating the compositions. We enforce this by declaring our operations with strong type signatures in Agda, using the `Fin` type to impose an upper limit on the indices. It thus becomes impossible to generate indices that are out-of-bounds with regards to the vectors they are used for.

The operation is well-typed: it supports combining an arbitrary number of iterators, of which each can produce values of an arbitrary type. The only restriction on these types is that they must have the same universe level, but this limitation only exists because there has been no need to support this. Each value in an output tuple must be combined into a single output value using a function, which the user of the Cartesian operator also has to provide. The bound of the Cartesian product is estimated based on the bounds of the input iterators, and inserted skips are taken into account during this computation. There are some helper functions, which make using the operator easier; these are two- and three-dimensional versions of the Cartesian operator.

3.4 Instances

In some other functional languages, there exist *type classes*. A type class is an interface that defines some kind of behaviour. For example, the Haskell language defines the `Eq` type class, which defines equality and inequality. It means that terms of types that are an instance of `Eq` can directly be checked for equality using the function `(==)` (or `(/=)` if inequality is desired). In Agda this concept of type classes is called *instance arguments* [6, Instance Arguments] and are indicated in source code with the keyword `instance`.

There exist many interfaces for which instances can be defined. Agda's standard library defines a number of these interfaces as well. In particular, we are interested in the following interfaces:

- **Functor.** If a type is an instance of the functor interface, it indicates that the type can be mapped over. It declares a single function named `fmap` in Haskell, and in Agda this function is named `_<*>_`. This function has two parameters: the first is a mapping function of type `A → B`, and the second is the data structure that is to be mapped.

  A common example of a type that is an instance of the functor interface is the `List` type. If a mapping function `A → B` exists, an object of type `List A` can be easily converted to one of type `List B`. It may be no surprise that the `List` implementation of the functor interface is simply the `map` function.

  Likewise, the functor implementation for `Iterator` is the `map` function as defined for iterators. It allows for a mapping from values of type `Iterator A` to values of type `Iterator B`, given that a mapping `A → B` exists. This functor is an instance of the functor as defined by the Agda standard library, so it can be used just like any other instance.

- **Monad.** Monads provide a standardised way to chain various operations into a longer process. This chain of operations represents a *deferred computation* that is only executed when its output value is evaluated. We will not attempt to give a full explanation or examples of monads here, but there exist several in existing literature [36]. Regardless, there are typically at least two functions that must be implemented for a type to be considered a monad:

```
return  ::  A → M A
(>>=)   ::  M A → (A → M B) → M B
```

  The type `M` is the monadic data type, and `A` and `B` are placeholder types.

  The `return` function is able to wrap any value into the monadic data structure. For `Iterator`, this is the `single` function, which accepts a single value and returns a singleton iterators, which yields only that exact value, and then announces that it has reached the end.

  The other function, `(>>=)` (pronounced 'bind'), is used to chain the next step of a computation. For iterators, this implementation was inspired by the `List` version of the bind operation. In lists, each individual item of the list is mapped to a new list, and this list of lists is then flattened (using the `concat` function) into a single list. The same happens with iterators: each value in the original iterator (of type `Iterator A`) is mapped to an iterator, so we end up with a nested iterator (of type `Iterator (Iterator B)`), which is then used as the input to the iterator `concat` function, so that a combined iterator producing all the values is returned.

  The bind function that we define for iterators is not compatible with the instance type that Agda's standard library declares. The standard library is too restrictive. Thus, `Iterator` was not made an instance of the standard library's monad structure. Fortunately, it does not have to be made an instance, and Agda will allow us to use the syntactic sugar for binding (do-notation), as long as we use the names `_>>=_` or `_>>_` and bring them in scope. This can be accomplished locally through the `let` or `where` syntax.

3.5 Unsuitable designs

While designing iterators, several implementations of iterators were developed that ultimately did not turn out to be suitable for use. Some were not usable for use in other parts of the system, whereas others were completely impractical, regardless of use. This section will describe two of these implementations for the sake of completeness.

The initial version of iterators in Agda was intended to be to the version developed by [33]. This definition uses an encapsulated state type parameter to hide the state type from the outside world. At the time when we first began working on iterators, we were not yet fully aware of how such encapsulation might work in Agda. The resulting implementation thus has type parameters for both the output type and the state type, shown in listing 11. The state parameter `S` was made implicit. This allows Agda to fill in the parameter most of the time, and only requires a programmer to manually fill in the parameter if its value cannot be determined automatically.

```
record Iterator {S : Set ℓs} (A : Set ℓa) : Set (ℓs ⊔ ℓa) where
  constructor iterator
  field
    bound : Bound
    state : S
    next : S → Step {_} {_} A {S}
```

Listing 11: A version of the `Iterator` record, using a type parameter for its state type.

This approach, with the added state type parameter (compared to storing the type as a record field) has some advantages, but also disadvantages. The first advantage is that having the state type as a type parameter means that the overall type of the iterator record has a level that is the least upper bound of the levels of the state and the output type. Compare this to the final implementation, where this level is the least upper bound of the level of the output type, and the *successor* of the level of the state type. Dealing with the state type in this way is also very intuitive.

Having an additional state parameter in the type is also the main disadvantage of this implementation. It requires functions that deal with the iterators to take this implicit parameter into account, or Agda will warn that a metavariable (the one representing this implicit parameter) is unsolved. Worse, it also requires types that contain the iterator to declare which type will be used for the state, which either introduces various restrictions if a constant type is used, or requires the wrapping type to also have a second type parameter for the state, which may not always be possible. Overall, the implementation works well, but it is very cumbersome to use. Several variants of this design were built, and all suffered from this issue.

The `Step` still has the additional type parameter to specify the type of the state value. There is no real need to embed the type parameter here. In the current design, the parameter is still implicit. It was however moved after the output type parameter, which is an explicit parameter. So if the implicit parameter has to be filled in explicitly, it is easier to do so, as its placement means that the implicit level parameters need not be filled in.

An entirely unsuccessful variant of the iterator was one where all the state data was contained in the output value as well. This was accomplished using the built-in sigma type[2], used in a non-dependent way. The first element would be the output value, whereas the second element would contain the state data. Each operation on an iterator would wrap the state type in yet another layer, in the resulting iterator. This design is similar to a (non-existent) variation of the `Colist` data structure in the Agda standard library, modified such that it has an additional constructor that represents skips, making it a combination of the `Colist` and `Delay` types from the standard library.

This approach proved to become unwieldly very quickly, as each `next` function must be able to deal with the growing complexity. A proof-of-concept was developed, but due to these issues and the development pains that followed from them, it was not developed any further. We do not recommend that anyone tries this approach, as we cannot conceive of a use case where an approach like this one may be relevant.

[2]The sigma type (also written as Σ-type) is a dependent pair of two values, where the type of the second value can depend on the first value. Its full type can be found in the Agda documentation, or built-in files [6, Built-ins, The Σ-type]

4 Attic usage

Before diving into the technical aspects that make up the Attic library, we will first explore it by giving examples of tactic usage. We will start by providing some examples of how Attic tactics can be used, and how they can be used to fill term positions. We will then give some high-level tactic programs that can fill holes. From there we will move to the implementation of lower-level tactics. The core instructions and the evaluation mechanism is detailed in section 5.

4.1 Defining and using tactic macros

Just like other proof assistants (this includes Agda), Attic uses backwards reasoning. It maintains a proof state consisting of unsolved goals, the focused goal, and solutions we have found so far. Attic tactics are programs that manipulate this proof state. These tactics are themselves written in Agda, and are executed by the compiler. They are constructed of instructions that can be interpreted by the Attic library. These programs can be combined and composed in a standardised way, and can then be turned into macros so that they can be used in term positions within Agda. The details of these structures and their implementation can be found in section 5.

Let us first take a look at the snippet of Coq code below. We first state a theorem, which is a logic expression for which we will need to supply a proof. In mathematical notation, this theorem says $\forall p(p \to p)$, which is a self-implication and thus trivially true ('if we have P, we can derive P'). In the proof, we introduce all variables that exist in the theorem (that is, `p`) using the `intros` tactic. We then prove that we can indeed derive `p` by using the `assumption` tactic, which tries to use the variables which are already in scope to prove the theorem. Naturally, this is a trivial proof, but it allows us to illustrate Attic well.

```
Theorem self_implication: forall (p: Prop), p -> p.
Proof.
   intros.
   assumption.
Qed.
```

In Agda we can write an equivalent proof. Observe the code snippet below. The type `A` is not explicitly defined here, but is declared as a generalizable variable [6, Generalization of Declared Variables]. We declare `self-implication`, with a type equivalent to $\forall p(p \to p)$. The proof is quite simple; we have introduced the variable `p` on the left hand side of the function definition, and use this variable as the proof that we can derive `p` from `p`.

```
self-implication : A → A
self-implication p = p
```

Let us see how we can express this solution with a tactic. Attic comes with some common tactics built-in; here we will use the `intro` and `assumption` tactics. Attic has no built-in `intros` tactic, but does have tactics that repeatedly apply another tactic. To achieve this repetition, we can define `intros` as `repeatTry 10 intro`, which attempts to apply the `intro` tactic at most 10 times. This number can be made as arbitrarily high as desired. Once that is done, we just need to combine these two tactics to achieve a program that will find the same output as the Coq tactic does. See the code below. Here we use Agda's do-notation, which will be desugared by the compiler to use the `_>>_` function behind the scenes [6, Syntactic Sugar]. In Attic, `_>>_` implements a sequencing operation.

```
selfImplication' = do
  intro
  assumption

macro selfImplication = asMacro selfImplication'
```

After defining the `selfImplication'` tactic program, we need to turn this program into a macro. This can be achieved using the `asMacro` function provided by Attic, which accepts a tactic program, and

```
si-hole : A → A
si-hole = {! selfImplication !}

si-term : A → A
si-term = λ z → z

si-macro : A → A
si-macro = selfImplication
```

Listing 12: Different ways to use macros in Agda. The highlighted code denotes a hole.

returns a function that can be executed by the Agda compiler. This means that this function accepts a single `Term` argument that represents the hole, and returns a `TC` value that represents the computation to be executed. Other types cannot be macros.

When invoked, the macro will gather relevant information to create an initial proof state. This proof state is a value that Attic maintains internally. It is similar to the proof state that can be found in Coq, which stores the list of (sub)goals, and the currently focused goal. Attic also stores this information, and additionally uses its proof state to keep track of the solutions that were generated by the tactics. While tactic programs are able to access and update this proof state, it is not directly visible to the proof author (unless it is logged to Agda's debug console using debugging commands). We can not create a partial proof script with Attic, apply that, see which goals are unsolved, and continue writing our proof script. This means that Attic is not a very interactive proving mechanism, at least not as interactive as Coq and Agda. But we can still use Attic a little interactively, and we will now describe how.

So let us see how Attic tactics can actually be used when they are turned into macros. The different steps and possibilities are shown in listing 12. We start out with the example `si-hole`, of which we have declared the type, but we still need to give an actual function definition. So we have left the actual implementation as a hole. We have however entered the name of our tactic macro into the hole, and if we use an editor that supports interactive Agda code editing, we can now start using this macro. We have several options at our disposal that we can use to fill the hole. These options are provided by the Agda compiler. We'll be describing these options here.

The first option is to ask Agda to *elaborate and give* the value we have placed in a hole. This is precisely what we have done to obtain the value in the second definition `si-term`. When we use this command, Agda will process the string we have placed inside the hole, so the hole we left in listing 12 will be resolved to the macro we defined earlier. Agda will then elaborate the macro, which means that it will execute the macro function. The hole will be used as the argument to the function. Any of the reflection calls that change the type checker's state are evaluated by the compiler, and if the function does not error, the hole is replaced by the value that the function assigned to the hole (if any). For `si-term` the tactic has generated the identity function as a solution, and Agda accepted this solution and has replaced the hole with the lambda function. This is an example of an edit-time[3] tactic: the tactic is fully evaluated at edit time, and this results in the proof term being present in the source code.

The second option is to simply use the *give* command, which will replace the hole with the value inside of the hole. While Agda will type-check the value before filling it into the hole, it will not reduce the expression but instead leave it as-is. This is what has been done with the `si-macro` example. Agda can recognize when macros are used in term positions, and will elaborate them at compile time, making this a compile-time tactic. Thus, if the tactic implementation changes, the term that is generated to prove `si-macro` may also change, without any further changes needed from the proof author. Elaboration of the macro does happen every time however, and the resulting expression is fully type-checked.

Both ways of running the tactics have their advantages and disadvantages. If the edit-time way is used and the theorem changes, the author may need to update the proof term that now exists in the source code. Changing a tactic might be easier and faster. On the other hand, if a tactic searches for a value

[3]The term "edit-time" is a wordplay on the words "run-time" and "compile-time", and refers to the time when a program is still being written in an editor [3], [37]. Edit-time tactics can play a role in interactive code editing.

to fill in this, that process might need some time. Storing the solution in the source code might then be faster to compile. The best approach should be considered on a case-by-case basis.[4]

4.2 Tactic implementations

Here we will give an high-level overview of the implementation of common tactics. We will not go into the core instructions and evaluation mechanisms here; these implementation details are given in section 5. Instead we start with the implementation of a trivial tactic, and slowly build our way up to a more complex example, and finally give the implementation of the `intro` tactic. Each implementation has a corresponding description that explains it.

4.2.1 Filling in terms

We will start by giving an example tactic that does exactly one thing: it fills in the variable that is lexically the closest in the current scope. The code for this tactic is given in listing 13. Internally, Agda uses De Bruijn indices to represent variables [6, Reflection]. De Bruijn indices are "a notational system where occurrences of variables are indicated by integers giving the 'distance' to the binding λ instead of a name attached to that λ" [38]. So instead of referring to variables by names, we refer to them by a number. In the original paper describing these indices, only positive natural numbers were used, but in Agda these indices start at zero. Thus, the variable `0` is the variable closest to us, and as the number increases, we are referring to variables farther away. Agda's reflection API also uses this representation for variables, so Attic also uses it by extension, as it uses the reflection API under the hood.

```
fillVar0 : Attic ⊤
fillVar0 = do
  goal ← getGoal
  let solution = var 0 []
  solveGoal goal solution

macro fillVar0' = asMacro fillVar0

fillVar0Demo : Bool → Bool
fillVar0Demo b = fillVar0'
```

Listing 13: An example tactic that fills in the closest variable in the context.

The example in listing 13 consists of three parts: a tactic implementation, a macro definition, and the demo that we use the macro for. The `fillVar0` tactic has the type `Attic ⊤`, which indicates that this value is in fact an Attic tactic, and when evaluated produces a value of the unit type, so it has no usable output value. When the tactic is evaluated, it first obtains the currently focused goal. It then creates a piece of Agda syntax using the `var` constructor. This constructor represents a variable indicated with a De Bruijn index, and some optional arguments which we do not use here. It then marks the goal as solved, using the solution we created here. This tactic is not very smart; it will try this solution regardless of whether the variable exists, or if its type is assignable to the goal type.

Next, we define a macro based on this tactic. We do this using the `macro` keyword and Attic's `asMacro` function. Defining a macro is important, because the compiler will then let us use the program in a term position, or let us run it from a hole. The `asMacro` function wraps the tactic program so that it can be executed by Agda's reflection machinery.

Finally, we have defined a function where we want to use the tactic macro we have defined. This is not a complicated function, and it only exists for demo purposes. The function accepts a Boolean value, and returns a Boolean value. The parameter is bound to the name `b`. Within the function body, `b` can be

[4] It is also possible to mix both approaches. By default Attic produces an error if a tactic leaves unsolved goals. We can however use an *admit* tactic which will mark every unsolved goal as solved, using each goal's metavariable as its solution. Agda will turn metavariables into holes during the unquoting process. Using the *elaborate and give* command to turn the tactic solution into a term with holes, we can then continue to solve these holes with other tactics, or we could even fill them in manually. This approach, where tactics are used to refine a hole in a big step, has been informally referred to in our discussions as 'refine on steroids'.

```
fillNat : ℕ → Attic ⊤
fillNat n = do
  (goal , type) ← getGoalAndType
  (def (quote ℕ) []) ← reduce type
    where
      t →
        failure $ errStr "The expected type is not a natural."
  nmbr ← quoteTC n
  solveGoal goal nmbr

macro fillNat' = asMacro $ fillNat 23
macro fillNat" = λ x → asMacro $ fillNat x

fillNatDemo : ℕ → ℕ
fillNatDemo n = fillNat'
fillNatDemo" : ℕ → ℕ
fillNatDemo" n = fillNat" 23
```

Listing 14: An example tactic that fills in a natural number.

referred to as variable `0` when using De Bruijn notation. Instead of using a syntax term for the function's body, we use the macro we have defined earlier, which fills in the value for us: the function ultimately returns `b` again.

4.2.2 Tactics with parameters

Let us investigate another example, but this time we'll fill in a natural number. This tactic will be a little bit smarter, and it also has support for parameters, so that it can fill in any natural number. The tactic is given in listing 14. We again have three parts to this example: the tactic implementation, the definition of two macros, and two demos, both using one of the macros.

The `fillNat` function takes a natural number, and returns a tactic that will fill the currently focused goal with that number. The basic concept is still the same; we retrieve a goal and we eventually solve that goal. But this time we also retrieve the goal's type. This is the type for which we are currently trying to construct a term. It is stored in the `Goal` structure, but this is a nice shorthand to retrieve both values at the same time. The type is reduced, to ensure that it is in a weak head normal form [6, Reflection], which allows us to match on it even if the type was not yet in the form we expected. We then have an assertion that the expected type is $\mathbb{N}$, the natural numbers type. If it is, all is good and we can continue. If it is not, we produce a failure.

Next, we create a numeric syntax term (the `nmbr` variable) and use that term to solve our goal. Note the usage of the `quoteTC` function. This is a tactic that wraps the Agda API function of the same name. Instead of manually constructing the syntax, we simply ask Agda to generate the syntax term that belongs to the numeric value we want to fill in. In this case we mainly use it for demonstration purposes; for small values it is more sensible to manually construct the syntax. For larger values, it is much more convenient to use the automatic quoting.

We then have to turn the tactic program into a macro, so that the program can be executed at compilation time. We do this with the `asMacro` function again. For demonstration purposes, we have created two variants of the tactic here. In the first macro, we have hard-coded the value 23, so the macro can be run without any arguments. In the second macro, the value we wish to fill in has been made a parameter for the macro, so it has to be provided at the use site. Despite these differences, both demo functions are the same and return 23 in this example.

4.2.3 A complete tactic: *intro*

Finally we will demonstrate one larger tactic program: the `intro` tactic. Although this program cannot fully fill in a term by itself, it can refine the goal that must be filled in. `intro` is used when variables need

```
combine : Bool → ℕ → Bool × ℕ
combine b n = b , n

combine' : Bool → ℕ → Bool × ℕ
combine' = λ b → λ n → b , n
```

Listing 15: Two different but equivalent styles of defining functions.

```
intro : Attic ⊤
intro = do
  (goal , type) ← getGoalAndType
  pi a@(arg argInfo _) b@(abs paramName returnType) ← reduce type
    where t → do
      failure $ strErr "Expected a function type, got " ∷ termErr t ∷ []
  let paramBinding : Binding
      paramBinding = paramName , a
  body ← newUnknownMetaContext $ paramBinding ∷ Goal.context goal
  meta ← getMeta! body
  let v    = visibility argInfo
      lmbd = lam v (abs paramName body)
      bodyGoal : Goal
      bodyGoal = (record { type = returnType
                         ; hole = body
                         ; meta = meta
                         ; context = paramBinding ∷ Goal.context goal })
  solveGoal goal lmbd
  addFocusGoal bodyGoal

macro example = asMacro (intro >> fillNat 23)

introDemo : ℕ → ℕ
introDemo = example
```

Listing 16: An example tactic that generates a lambda, using a natural number for the lambda's body.

to be bound. The `intro` tactic does exactly this, and generates lambda terms. This is illustrated with listing 15. The first function definition is an ordinary function. The second function definition is what we would get by applying the `intro` tactic twice: two lambdas that bind both function parameters, with the inner lambda returning the result of the computation.

The `intro` tactic implementation is shown in listing 16. The overall structure is similar to the earlier examples: we have the Attic program, the macro definition, and finally the demo where the macro is used. We will be going through this example step by step, starting with the tactic program:

- We again collect both the goal and the expected type. The type is reduced, so that we can match on it.
- If the type is a function type (represented by `pi` in Agda), we bind the information we need to named variables and continue on. If the type is not a function type, we return a failure, as there is nothing that we can introduce.
- Next, we generate a new binding. This binding will be added to the telescope of the lambda that we will eventually be generating.
- We generate a new `meta` term using the compiler, and extract the actual `Meta` value. This term will be used to create a new hole. The metavariable will be used to create the goal representing this new hole.

- Then we generate the actual lambda term that will serve as the solution. To do so, we also need some other values that the compiler requires. The body of the lambda is the `meta` term we just created, so the body of the lambda is left unspecified. The compiler will treat it as a hole.
- We then generate the goal that represents the body metavariable. The `intro` tactic cannot generate a complete term on its own, it merely refines an existing hole. So the body still has to be filled in, a fact that will be represented by this goal.
- The original goal is then marked as solved, using the lambda term as the solution.
- Finally, the lambda body goal (which represents the unknown term) is added to the list of unsolved goals. It is also set as the focused goal. A tactic that is executed after `intro` can thus automatically work on this new goal.

Now that we have the tactic program, we convert it to a macro using the `asMacro` function again. But as stated before, the `intro` tactic cannot generate a complete term on its own; it will leave a hole in the lambda's body. Attic will produce an error if unsolved goals remain at the end of a tactic execution, so it is important that all goals are solved. To remedy this, we have re-used the `fillNat` tactic from listing 14. This tactic solves the hole. The `fillNat` tactic is chained to the `intro` tactic. As a result, a lambda function is generated first, and then the hole inside of the lambda will be filled with the number we passed to the `fillNat` tactic.

Finally, we have the demo function on which the tactic macro is used. This is an ordinary function, but this time no parameters are matched on the left hand side of the function definition. Instead, these are bound by the tactic. The generated function returns the constant value 23.

5 Attic

The library that was developed as part of this thesis is the Attic library. The name is a play on the words 'tactic' and 'Ataca', the name of a similar project which will be discussed later[5]. Although Attic does contain some of the standard tactics, its intended purpose is to mostly be a foundation upon which more specialised tactics can be developed. It also houses the iterators as detailed in section 3.

The following section, section 5.1, focuses on the overall architecture of the Attic evaluation pipeline. This is considered an introductory section, and does not focus on implementation details and choices; these are described in subsequent sections. First, section 5.2 discusses a number of fundamental data structures. Section 5.3 lays out the instructions that are present in the Attic library, and the specified behaviour. Then, section 5.5 describes how these instructions are evaluated to produce a final proof state which can be used to fill in holes.

5.1 Overall architecture

The Attic library comes with a number of well-known tactics built-in, which can be used right away. But the goal of Attic is not just to provide a number of tactics that are commonly found in other proof assistant systems, but also to let end users define their own tactics. This can be achieved by combining existing tactics, or by writing entirely new tactics that manipulate the proof state in a novel way. Regardless, Attic needs to be flexible enough that all these use cases are accommodated. Thus, providing a number of fixed tactic programs that are able to execute certain actions will not be good enough.

Attic has been designed in such a way that tactic programs are first constructed using a fixed number of *core* instructions. These core instructions can be combined and chained to create new tactics, as they together produce the desired behaviour from the resulting program. They can also be combined with existing tactics, allowing for composition. In this way, tactics can even be dynamically generated. Combining and chaining tactics is an easy process, because the monadic *bind* operation is used for this purpose, for which Agda has support in the form of syntactic sugar: do-notation [6, Syntactic Sugar].

Once the tactic programs are fully constructed, they must be evaluated. A visual overview of this evaluation is given in fig. 5; the following text describes it in more detail. The evaluation of a tactic program amounts to running each of the instructions that it comprises, and keeping track of the programs's proof state as it is modified by the tactic instructions. This interpreter is also capable of evaluating computations that have side effects. These side effects are interactions with the Agda compiler, to gather information or create new metavariables. During the composition of tactic instructions, these computations are delayed. When interpreting the tactic program, these delayed computations are executed, and their results are used in the rest of the tactic's evaluation.

After the tactic is fully evaluated, the resulting proof state is checked for validity, and the solutions found (if any) are then submitted to the Agda compiler, where they are either accepted or rejected. This happens through the `unify` function that Agda provides. No solutions are final during the evaluation phase, but may turn out to be valid in the unification phase. We call this *deferred unification*: we only unify terms when we have generated the full solution. In the end, the Agda compiler acts as the final gatekeeper to ensure that the solutions proposed by the Attic tactics are valid. Depending on the outcome of this operation, the tactic runner may be able to backtrack and find another solution to try.

This loop of backtracking and trying other candidate solutions is guarded by a *fuel* parameter that limits the number of iterations that can be performed. It starts out with the default value, which is set to 10,000 (ten thousand), and decrements upon every evaluation. The next iteration then starts with this decremented value. The starting fuel number can be customised by the user if necessary. Eventually, tactic evaluation will result in one of the following states:

- *Finished*. This value indicates that the tactic was successfully evaluated. At least one tactic branch proposed a solution that was accepted by the Agda compiler.

[5] Wordplay appears to be all too common in the circles of proof assistants. The name of the Agda programming language was derived from a Swedish song 'Hönan Agda' [39] (literally 'The hen Agda'). This is a reference to the Coq proof assistant, whose name means 'rooster' in French, but is itself also a reference to CoC (the Calculus of Constructions) and Thierry Coquand, one of the initial authors of Coq [40].

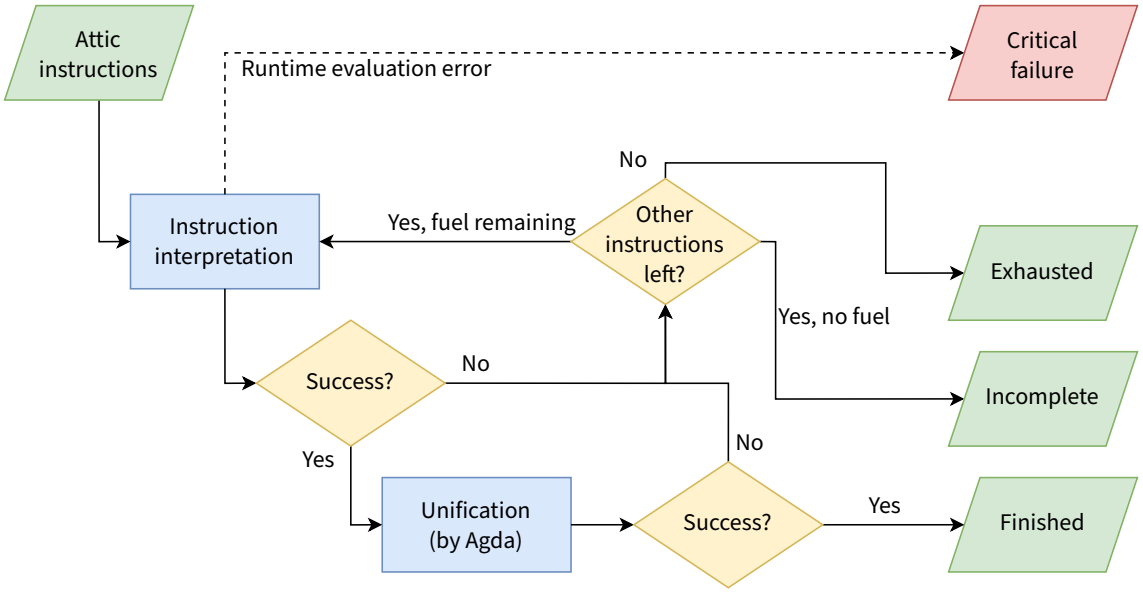


Figure 5: An overview of the evaluation of tactics.

- *Exhausted*. This value indicates that no valid solution was found, and that all possible solutions that could be generated by the tactic have been evaluated. This suggests that the tactic developer should change the tactic so that it proposes different solutions that might work.
- *Incomplete*. This value indicates that no valid solution was found, and that the maximum search depth was reached. There may have been valid solutions that were never evaluated. This status suggests that the tactic developer should optimize the tactic to produce fewer results, or if that is not possible, increase the maximum search depth.
- *Fatal error*. If at any point during tactic evaluation a runtime error occurs, evaluation is terminated immediately. This error will be presented to the developer, along with relevant error information. Such errors include unsolved holes remaining after evaluation, no hole being focused, or an invariant not holding. The runtime does not recover from fatal errors, as these likely indicate an unexpected failure that ought to be fixed.

5.2 Core structures

During execution Attic maintains an internal state which is composed of several data structures. This internal state is integral to the evaluation of tactics, as it represents the current proof state, and also holds the solutions that were selected for the goals.

The first structure is an analogue to data that can be obtained from the Agda compiler. In Agda, terms that are yet to be filled in are represented by metavariables. These are opaque values to consumers of the Agda API, but they do have meaning for the internals of the Agda compiler. Although metavariables are typically referred to as *meta* or *metas* in Agda parlance, we will refer to them as metavariables to avoid confusion with the built-in `Meta` type and the `meta` constructor of type `Term`.

Since Attic tactics operate on holes in term expressions, there is always an initial hole for which an expression must be generated. This hole is represented by the `meta` constructor, which contains both the `Meta` value, as well as the list of variables in scope that are available to the metavariable. This list is called a telescope. It is important that this telescope is well-maintained and correct, as the compiler asserts this correctness during some interactions, and candidate solutions may depend on the variables listed in this telescope, for which a valid telescope is required. When obtaining or creating holes, Attic internally creates an instance of the custom `MetaTerm` record type, which stores both the metavariable and its associated telescope. An equality function (`_meta=_`) that decides whether two instances of `MetaType` are equal is implemented, which is based on the Agda primitive `primMetaEquality`, which

implements equality for the opaque values of type `Meta`. We use these function (as well as its negative variant `_meta≠_`) to find and filter goals, as metavariables are used to uniquely identify goals. This is further detailed below.

Goal

Attic has defined additional record types to store the proof state. The first notable type is the `Goal` type, of which each instance corresponds with a hole in Agda. It stores various fields which are relevant to when examining and manipulating goals:

- *Type.* The type of a hole is stored inside the goal. Although it is possible for a type in Agda to be partially defined (types in declarations can also have holes, and thus contain metavariables), Attic is designed with the expectation that holes are well-typed. There are no technical limitations within Attic itself that prevent partial types from being accepted, although some tactics might assert various properties about the type, and produce a fatal error if such an assertion does not hold.
- *Hole.* The Agda term representing the hole is stored. In practice this will always be the constructor `meta`. While this information itself is not particularly useful during tactic evaluation, it is crucial in the final phase of unification, as the Agda compiler expects a value of type `Term` here.
- *Metavariable*. The metavariable representing the hole is stored in the `Goal` structure, using the `MetaTerm` type described earlier. It is used to uniquely identify a goal. Initially derived from the term that will end up in the hole field. This value could theoretically be used to reconstruct both the hole and context fields of a `Goal` record, but for convenience reasons, these two fields are still kept as separate fields.
- *Context.* When dealing with goals, its 'context' refers to the variables that are in scope for that goal, also known as its environments. Keeping track of this telescope is important when finding solutions, and for certain interactions with the Agda compiler.

The context field especially is an improvement compared to existing systems [9]. Previous systems did not explicitly keep track of this value (or had a way to derive it). Instead, they relied on frequent use of the `getContext` function that the Agda reflection API provides. Using this function requires less internal bookkeeping, but makes it harder to store a goal for later use.

Solution

In order to represent solutions to goals, the record type `Solution` is introduced. It keeps track of the following two fields:

- *Goal.* The goal for which it represents a solution. The metavariable that is stored inside this goal is also used as a uniqueness value to identify the solution.
- *Solution.* The solution for the goal. This is a value of type `Term`. Later, the Agda compiler is able to unify the hole of the goal with this term.

Proof state

Finally, these types are all combined in a type that represents the entire proof state. The `ProofState` is composed of the following fields:

- *Goals.* The list of unsolved goals. When a goal is solved, it is removed from this list.
- *Solutions.* The list of solutions. These solutions include the goal value (see above), so no information is lost when adding a solution to the list.
- *Focused goal.* An optional value representing which goal is currently in focus. For some tactics it is required that only a single goal in the list of goals is focused. If a goal is specified by this field, then this is the goal that will be used. If no value is given, the first goal in the list of unsolved goals is used, if one exists.

```
doublePair : Σ[ (x , y) ∈ (ℕ × ℕ) ] y ≡ x * 2
doublePair = (2 , 4) , refl
```

Listing 17: An example of a constrained type, using a dependent pair.

```
private
  variable
    ℓ : Level

{-# NO_POSITIVITY_CHECK #-}
data Attic (A : Set ℓ) {ℓs : Level} : Set (ℓ ⊔ suc ℓs) where
  value : A → Attic A
  failure : List ErrorPart → Attic A
  error : List ErrorPart → Attic A
  tryCatch : (try : Attic A {ℓs}) → (catch : Attic A {ℓs}) → Attic A
  primitiveTC : TC (Attic A {ℓs}) → Attic A
  updateState : (ProofState → Attic A {ℓs} × ProofState) → Attic A
  branch' : (iter : Iterator {ℓs = ℓs} (Attic A {ℓs})) → Attic A
```

Listing 18: The definition of the `Attic` type, which represents the tactic program instructions.

The concept of a focused goal is not a new one; other proof assistants are already keeping track of the goal that is currently being solved. It is adopted here as well, so that the built-in tactics can always get the right goal to operate on. It also enables tactic authors to explicitly set the goal they would like to solve, and potentially solve goals out-of-order.

As its name implies, the field containing solutions is used to store the solution to each goal. These solutions should not be considered as definite solutions yet; rather they are candidate solutions for which unification will be attempted in the unification stage. If Agda decides that the solution is invalid, Attic will discard the solution and propose another solution, if one exists. As such, Attic employs *deferred unification*, and only tries to unify when the tactic program finds a solution.

For an example of such an invalid solution, consider the following. Agda's type system allows us to create types such that two holes emerge in such a way that the type of the first hole depends on the second one. Suppose we have a dependent pair of two naturals where the second natural must be twice as large as the first. This type, along with one of its infinitely many solutions, could be expressed as in listing 17.

In the proof expression, the constructor `refl` is the only constructor of the structural equivalence type. If the tactic that was used to fill the second hole is not aware of this constraint, it may have suggested a value that ultimately turns out to be invalid (such as `(0, 1)` where 1 is not equal to 2×0). This discrepancy will be caught by the Agda compiler as it tries to unify the values with the type it is solving for.

5.3 Instructions

Fundamentally, Attic tactics are guided search programs. The problem space they work on is the range of values that inhibit the type of the hole that is to be filled. The term that fits the hole is the solution to the problem. Within the Attic library, the instructions that represent these programs are defined by the data structure shown in listing 18. Each constructor in this definition will be explained.

Note that `Attic A` is polymorphic on type parameter `A`, which is constrained to type `Set ℓ`. This makes it suitable for holding all types of values. The `branch'` constructor stores an iterator, and to accommodate this data structure, an additional level parameter `ℓs` is specified on `Attic A`. The overall type of `Attic` is a `Set` whose level is a combination of these variables. The parameter `ℓs` itself is propagated through the various constructors. Although we have removed implicit parameters from most code snippets, we decided that omitting the parameter here would result in a more confusing definition.

The meaning of the several constructors on `Attic` is defined as follows:

- `value`. This constructor carries a single value. This value can be the result of another computation. Usually this constructor is used as an intermediate instruction. In tactic programs, it also ends up as the leaf of the instruction tree, since each tactic instruction must eventually carry a value of type `A`. Using a tactic that carries the unit type's only possible value is a common option.
- `failure`. This instruction is used to signal that the tactic program is unable to produce a valid solution. For example, a tactic that can only fill in holes whose type is numeric may produce this instruction when it is used to fill in a hole where a Boolean value is expected. It is important to note that this is considered a *recoverable* failure; the interpreter may backtrack (if possible) and find other candidate solutions. The sole argument to this constructor carries information about the failure that occurred.
- `error`. Used to signal that a fatal error has occurred. This instruction is generated when an erroneous situation is encountered which would result in undefined behaviour. It is also produced when an invariant does not hold. For example, a higher-level instruction that adds a goal to the list of unsolved goals may verify that no goal with that metavariable is stored. If there is one, it would produce an `error` with the appropriate error details, as this state is likely the result of a user error. The choice to include fatal errors that trigger immediately is an explicit design choice, following the "fail-fast"-technique [41]; some conditions should not allow the program to continue running in a possibly corrupt state.
- `tryCatch`. Used to handle failures, and potentially recover from them. This instruction carries to values of `Attic A` inside. Upon evaluation, the first instruction will be executed. If this instruction produces a success result or a fatal error, this result is immediately propagated. But if the first instruction produces a soft failure, the second instruction will be executed, and its result is then returned without modification. A wrapper named `try` exists, which simply does nothing when a soft failure occurs.
- `primitiveTC`. Represents an instruction that requires interaction with the Agda compiler. Such an operation is typically a primitive reflection operation, whose return value is usually either of type `TC A` (where `A` is a type parameter) and in some cases `TC ⊤`, a special case where the return value has the unit type.

  This instruction requires that the primitive operation produces a value of type `Attic A`, which is the next instruction the interpreter evaluates. Accordingly, there is a utility function which is able to 'lift' any such operation producing `TC A` to `TC (Attic A)` so that it may be embedded in the `primitiveTC` instruction.
- `updateState`. A lower level instruction which allows for updating the state of the interpreter. Its only parameter is a function which accepts a `ProofState` value, and produces a pair of `ProofState` and `Attic A`. In practical terms, this function takes the current state, and based on that state, it generates the new state and the next instruction that the interpreter will execute.

  We do not expect that this instruction will be used directly, in most scenarios. Rather, it makes more sense to write some high-level tactic programs that use `updateState` internally. These high-level tactics can then be used when composing larger tactic problems. For example, there is a tactic named `solveGoal` which takes a `Goal` and `Term` value, and then manipulates the state such that the goal is removed from the list of unsolved goals, and the solution is added to the list of solutions. It produces an empty instruction of type `Attic ⊤`.

  Of course it is perfectly feasible to write a tactic that manipulates the state in another way. It is also possible to manipulate the state in such a way that it becomes corrupt; the instruction does not contain any safeguards that verify the correctness of the new state object. It is possible to create a wrapper instruction, say `updateStateSafe`, which performs exactly this tasks. As part of this work, this functionality was not implemented.

  A variant that *was* implemented is a read-only version of the instruction, named `useState`. This function takes a single function of type `ProofState → Attic A`, which only requires that the next instruction be generated, based on the current state. Internally, `useState` constructs an

`updateState` instruction which maintains the current state. Because it does not allow for the generation of a new state and the added convenience of using a shorthand make this function more attractive than directly using `updateState`.

- `branch'`. When this constructor is encountered, the interpreter is instructed to branch the current flow of execution. Internally, it holds an iterator of instructions. This operation is the backbone of the backtracking functionality. It allows the user to specify several possible options as candidate solutions. Using an `Iterator` here ensures that the system is as lazy as possible, as iterator values are only materialised when they are required.

  This instruction also has several functions that wrap its functionality. Note that the default function is named `branch'`. This name is intentional, as it is not meant to be the default instruction in day-to-day use. There is also a function named `branch`, which maps an object of type `Iterator A` to `Iterator (Attic A)`. This shorthand function is much more convenient, as it allows a user to specify the values to branch on, rather than the tactic instructions. Similarly, there is a function named `branchList` which acts as a similar shorthand, but for list values.

5.4 Instances

The `Attic` data structure supports a number of operations that are common in programming languages with functional paradigms. For the purposes of Attic, implementing the functor and monad interfaces was enough. These are the same interfaces that are implemented by iterators (section 3.4); functor and monad. The functor interface is implemented so that we may turn an Attic instruction of one type into an Attic instruction of another type. The monad interface is implemented as it is a good way to implement the chaining of instructions. Both interfaces have thus been implemented for the `Attic` data structure. The `branch'` instruction wraps an iterator, which requires two level parameters in its definition. This is unfortunately not compatible with Agda's standard library. Again, this is not a big problem, since Agda gives us the flexibility to define these functions anyway, and use them if we define them locally. It is still an inconvenience, however, especially if we wish to use the categorical implementations of Attic and those from the standard library in the same scope.

The functor interface is fairly straightforward. If a function of type `A → B` is given, an instruction of type `Attic A` can be mapped to a value of type `Attic B` using the `fmap` function. Any values inside of the instruction are mapped as well. For example, instructions that execute a function on the Agda compiler API (and thus wrap a `TC` value) are mapped using a functor implementation of `TC` values.

Each `Attic` constructor has its own implementation within `bind`. For the `value` constructor, the value binding function is directly applied to the value it carries. The `failure` and `error` constructors both only propagate their messages. In the `tryCatch` case, the bind operation is applied to both instructions it contains. For the `primitiveTC` operation, the binding operation is applied to the `Attic` value that is carried by the `TC` value. With `updateState`, the instruction of the function's return value is bound. The `branch'` performs a mapping operation on the iterator, using the `bind` function as the mapping function.

5.5 Interpretation and branch evaluation

The Attic evaluation mechanism consists of three functions, that can together be used to fully evaluate tactics. There is one function, `runAttic`, that evaluates `Attic` instructions, and keeps track of the proof state as it does so. It decides whether the instructions were able to find a candidate solution. The second function, `unifyAtticResult`, will determine whether the solution is valid. To determine this, it will use the compiler API to propose the solution. If the solution is accepted, the function produces a success result. But if the solution is not accepted, it will result in a failure signal. Finally, the third function `evaluateAttic` wraps both of these functions, and is able to backtrack to other candidate solutions, and test those.

5.5.1 Tactic success evaluation

The initial tactic evaluation occurs in the function `runAttic`, whose type is given below. `runAttic` requires two arguments, and produces a result output type. The first two arguments are rather straight-

```
AtticClosure A = Attic A × ProofState

Rest A = Iterator (AtticClosure A)

OptionalRest A = Maybe (Rest A)

data AtticResult {a : Level} (A : Set a) : Set a where
  atticSuccess : ProofState → OptionalRest A → AtticResult A
  atticFailure : List ErrorPart → OptionalRest A → AtticResult A
  atticError : List ErrorPart → AtticResult A

data UnificationOutcome {a : Level} (A : Set a) : Set a where
  unificationSuccess : UnificationOutcome A
  unificationFailure : OptionalRest A → UnificationOutcome A
  unificationError : List ErrorPart → UnificationOutcome A

data EvaluationOutcome : Set where
  finished : EvaluationOutcome
  incomplete : EvaluationOutcome
  exhausted : EvaluationOutcome
  fatalError : List ErrorPart → EvaluationOutcome
```

Listing 19: Additional data structures used for `Attic` evaluation. Some declarations and details have been omitted from these definitions for printing purposes.

forward; the first is the tactic that needs to be evaluated, and the second argument is the proof state that will be used to evaluate the tactics. The return type of this function requires some more explanation. The outer type is `TC`, which means that the function is able to interact with the Agda compiler. The value it contains is one of type `AtticResult`, whose definition is given in listing 19. This type distinguishes three possible results: success, recoverable failure, or a fatal error.

```
runAttic : Attic A →
           ProofState →
           TC (AtticResult A)
```

If evaluation of the tactic is a success, an `atticSuccess` is returned, which contains both the final proof state, and possibly a *rest* which will be discussed soon. The tactic might also produce a recoverable failure, signalled by `atticFailure`. These failures are expected, and should be recovered from. Another solution should be tried, as this one is invalid, and the backtracking mechanism should be invoked. Finally, tactic evaluation can also result in runtime errors, which are expressed by the `atticError` constructor. These errors indicate a unexpected exceptional behaviour, and may mean that the proof state is invalid. These errors should not be recovered from.

Both the success and recoverable failure carry an optional *rest* value. The `runAttic` function can only evaluate a single branch at once. When a branching instruction is encountered, it will only attempt to evaluate the tactic program at the head of the branching instruction. Any remaining tactic programs are not evaluated, but are stored alongside the current proof state, so that their evaluation may be resumed at a later point. This is represented by the `AtticClosure` type in listing 19. The `Rest` type is an iterator of these closures, and an optional version of this type is also defined.

The implementation of `runAttic` is mostly straightforward. Each of the instruction is evaluated with the main purpose of modifying the proof state. This proof state is passed along with each recursive call. The details for the different instructions in `Attic` is given below:

- `value`. This instruction produces `atticSuccess` with the current proof state as its argument. This result indicates that a candidate solution was found. No rest is returned.
- `failure`. This instruction produces `atticFailure` with the details from the instruction. This

result indicates that this branch of execution did not produce a candidate solution, but backtracking to another branch may yield a result. No rest is returned.

- `error`. This instruction produces `atticError` with the details from the instruction. This result indicates that a fatal error occurred which is likely due to a user error. No recovery should be performed after this error is raised. It also does not return a rest.
- `tryCatch`. This instruction first evaluates the first instruction contained within. If this is a soft failure, return the result of evaluating the second instruction, otherwise (in case of success or fatal error) pass on the result of evaluating the first instruction.
- `primitiveTC`. This instruction evaluates the `TC` operation contained in the instruction. It then recursively calls `runAttic` on the resulting instruction, along with the same proof state, and returns the output of the recursive call.
- `updateState`. The function that the constructor carries is applied to the current proof state. The function will produce a new proof state and next instruction as its output. Together, this new instruction and proof state are used to continue evaluation.
- `branch'`. In this case, the next step of the iterator in the instruction is computed. Now, there are again three cases. If the iterator produced a `Done`, an `atticFailure` and no rest is returned. If the iterator produced a `skip`, an `atticFailure` is returned, along with the tail of the iterator. If the iterator produced a `Yield`, a recursive call is made to `runAttic` to run the instruction, and the rest of the recursive call is combined with the rest of the recursive call before returning the output of the recursive call. This ensures that no instructions are forgotten, and due to this ordering it is also ensured that the instructions are evaluated in the order in which they are defined. The current proof state is included when the rest iterator is constructed.

5.5.2 Unification

The second stage of interpretation is unification. This stage is handled by the `unifyAtticResult`, whose type is given below. It accepts a single argument, which is the output of `runAttic`. It will decide which action to take based on the value of this value. It returns a value of type `UnificationOutcome`, which is defined in listing 19.

```
unifyAtticResult : AtticResult {ℓs} A →
                   TC (UnificationOutcome {ℓs} A)
```

There are three possible tactic evaluation results. If the evaluation succeeded, this means that a candidate solution exists here. This solution is first checked to see if any goals are left unsolved; if that is the case, a fatal error is raised. Otherwise, each element in the list of solutions is processed, so that the original metavariable is unified with the associated solution. This unification is done through the `unify` function, which is part of the compiler API that Agda exposes. If all solutions are unified without errors, a `unificationSuccess` value is returned. It does not carry any further information, as a solution was found.

Alternatively, if unification fails, or if the tactic evaluation failed in the first place, the value `unificationFailure` is returned. This value carries the rest, so that backtracking to partially evaluated Attic programs is possible. No specific failure message is attached to this outcome, as we will likely backtrack from this failure. Finally, if the original tactic produced a fatal error, that error is propagated here. The constructor `unificationError` is used to indicate this outcome. It has only the error message as its parameters, and no rest to backtrack to.

5.5.3 Evaluation and backtracking with rollbacks

The two previous functions are combined in the `evaluateAttic`, which is able to comprehensively evaluate Attic tactics, and can take branching instructions into account as well. Its type is given below. It accepts the maximum search depth, the tactic program to evaluate, and the initial proof state with

which the evaluation is executed. It then evaluates this to an `EvaluationOutcome`, which is described in listing 19.

```
evaluateAttic : (searchDepth : ℕ) →
                Attic A →
                ProofState →
                TC EvaluationOutcome
```

The `searchDepth` parameter is used to determine the starting amount of *fuel* that this function can 'spend' on recursion, to guarantee termination. Specifically, it is used as the input to a local function that is used for looping through recursion. On every recursive call, the fuel parameter is decremented. When the number reaches zero, all processing is terminated, and the outcome is `incomplete`, signalling that no valid solution was found, but that more solutions might exist. Proof authors should either optimize their tactics to produce fewer invalid solutions, or increase the search depth parameter.

Internally, `evaluateAttic` constructs a local function that respects the remaining fuel to ensure termination, takes an `AtticClosure` (which is a product of tactic program and the state with which it should be executed), and finally has a `Rest` parameter to facilitate backtracking. The local function first constructs a computation that evaluates the tactic and tests its solution, using the previous two functions. The evaluation of this computation happens within `runSpeculative`. This function is part of the Agda compiler API, and allows us to run a `TC` computation speculatively. At the end of the computation, we can instruct the compiler to either keep the modified compiler state, or roll back to the state as it was before we performed the computation. In practice this means that we keep the compiler state if unification succeeds, because we have found a solution. But if unification fails, the state is rolled back, as the new compiler state might contain problematic data. For example, a tactic might create a new metavariable and then fail. If the state is not rolled back, the metavariable will never be solved, causing Agda to emit warnings for this unsolved metavariable. If Agda is launched non-interactively (e.g. through a shell script), its exit code will indicate a failure. This property is undesirable, and is thus being remedied with state rollbacks. Unfortunately this solution does not work as well as we had hoped and may result in Agda compiler panics. See section 7.3 for more details.

The local function will discriminate on the output of the unification. If the unification was a success, the function can return immediately, with the `finished` constructor. This indicates to any callers that a valid solution was found. Alternatively, if the tactic evaluation or unification produced a fatal error, this error is simply passed on in the result through the `fatalError` constructor. The most interesting behaviour occurs when a recoverable failure is returned. First, the returned rest (if it exists) is prepended to the existing iterator of rests, so that we can find the next program in either the freshly-returned rest, or the pre-existing rest. The next step in this iterator is then computed. If the iterator is done, the value `exhausted` is returned to signal that all possible solutions of this tactic are evaluated. If the generator yields a tactic closure, we make a recursive call with that closure, a decremented fuel value, and the tail of the rest iterator. If the rest iterator skips, we still make that recursive call, but since we have no tactic closure, we use a backup `AtticClosure` value, whose sole `Attic` is an immediate failure. This ensures that search can continue, and that skips in the rest iterator still count towards the search depth, while incurring little overhead.

5.5.4 Macro conversion

The `evaluateTactic` function is used in the `asMacro` function, which accepts a tactic program of type `Attic A`, and turns it into a function of type `Term → TC ⊤`. This type can be used for macros. The Agda compiler will use the current hole as the argument for the `Term` parameter. The `TC ⊤` value that is returned by `asMacro` is the computation that will try to solve that hole by filling in a suitable value.

When the final parameter, the hole, is specified the `asMacro` function will do a number of things. It will first infer the type of the hole, and gets context of the hole. It also gets the metavariable that is associated with the hole. Based on the data collected here, it will create a `Goal` value that matches the hole, with the type, context and metavariable all set to the appropriate values. Then, the initial proof state is initialised. This proof state contains no solutions, has only the goal we just created in its list of

```
macro doNothing = asMacro (exact tt)

theorem : ⊤
theorem = {! doNothing  !}
-- The given hole is not an unsolved meta: tt
-- when checking that the expression unquote doNothing has type ⊤
```

Listing 20: An example where Agda has automatically decided what the correct solution to the hole should be, meaning that it will not generate a metavariable. If Attic is used on such a hole, it will error. This code demonstrates that; if the 'give' command is used on the hole, the error below it will be returned.

goals, and also has that same goal focused. The `evaluateAttic` function is used next, to evaluate the Attic program against this proof state. If a valid solution is found, `asMacro` simply returns. In the failure cases, the function will raise a type error with a relevant message.

These generated macros can then be used in the place of terms, as the compiler will expand them to proper terms. The tactics are then used as compile time, and can be referred to as *compile-time tactics*. Alternatively, a proof author could place the tactic inside of a hole, and ask the compiler to expand the term. The hole will then be replaced with the term that is generated. Should the tactic implementation change, the term will not be modified. This is known as an *edit-time tactic*. Whichever application is best depends on the needs and wishes of the proof author, and the proof they are working on. Both workflows are equally well supported by the compiler.

An interesting point to note is that Attic requires that the hole term is, in fact, the `meta` constructor, which carries a metavariable. Otherwise, there would be nothing to solve. This might appear to be a redundant check, but it really is required to handle some cases. For example, Agda may fill in the solution when it can definitely determine what the solution is. An example is given in listing 20. If we declare a variable of the unit type `⊤`, leave its value as a hole, and then try to run a tactic within that hole, Attic will immediately produce a type error. The error reads "The given hole is not an unsolved meta: tt" which implies that Agda automatically filled in the `tt` value for the hole (in this case, through eta-expansion for records [6, Record Types]). This behaviour was unexpected for holes, however. We have not seen any other instances where the metavariable is replaced with another syntax term, but more may exist.

5.6 Safety of implementations

Back in section 2.3.2 we described some of the checks that Agda runs to ensure that the code we write is total. These checks are mechanical termination checking, coverage checking for completeness, and strict positivity enforcement on data types. Termination and positivity checking can be disabled for definitions by placing a pragma above it. Such pragmas have been applied to various functions and data types in Attic.

Perhaps the most notable pragma has been included in listing 18 already. This code listing contains the definition of the `Attic` instruction type. The `Attic` type is used throughout the library, and represents the underlying instructions that are evaluated when a tactic program is run. It might be surprising to see that this fundamental structure is considered unsafe by Agda. But `Attic` appears in the types of the parameters of some of its constructors. Thus, the definition is not strictly positive. The non-inductive recursive use of `Attic` is however required to provide functionality such as compiler interactions or branching.

Additionally, a handful of functions in Attic make use of unsafe recursion. These uses mostly occur in the functions that process values of the `Attic` type directly. One example is `runAttic`, which is the function we described in section 5.5.3. The `bind` and `fmap` functions for `Attic` similarly require that the termination checker is disabled for their definitions. In practice these functions will always terminate, as they still represent a finite tree structure. Agda is however unable to prove that they will terminate because the definitions do not look like inductive data types syntactically.

A consequence of the use of unsafe code is that Attic cannot be used in files where the `--SAFE` option

```
-- Get focused goal or first goal
getFocusedGoal : ProofState → Maybe Goal
getFocusedGoal state =
  ProofState.focusedGoal state <|> head (ProofState.goals state)

-- Gets the currently focused goal
getGoal : Attic Goal
getGoal = useState λ state →
  maybe' return (error $ errStr "No goal is focused.") (getFocusedGoal state)
```

Listing 21: The implementation of the `getGoal` tactic, which obtains the current goal.

```
-- Adds a goal to the list of goals
addGoal : Goal → Attic ⊤
addGoal goal = updateState λ state →
  goalDoesNotExistOrSolved goal state ,
  record state { goals = goal :: ProofState.goals state }
```

Listing 22: The implementation of the `addGoal` tactic, which adds a goal to the list of unsolved goals.

is used, as Agda will then not allow the usage of the unsafe pragmas. This also applies transitively to modules that are imported by that file. It thus becomes impossible to use Attic tactics in a file where the safety option is enabled. A workaround is disabling the safety option temporarily, and then making use of the tactics to generate a term that is stored in the source code. Once the term is filled in using the tactic (this can be done with the *Elaborate and give* command that Agda-aware editors provide), Attic is no longer necessary, and the safety option can be re-enabled.

5.7 Common abstractions

Some of the operations in the `Attic` type are rather low-level. It is possible to build abstractions on top of these instructions that make it easier to perform common operations. The most notable instruction here is the `updateState` constructor. This constructor makes it possible to read and change the proof state during tactic evaluation. It is quite powerful, but can also be a bit cumbersome to use directly. Thus, it makes sense to abstract away some state-related operations that are often used. Various such operations have been built into the Attic library, because they can be used in most tactic programs. These tactic programs can then be composed with or embedded into other tactics to form larger programs.

5.7.1 Building blocks

Let us take a look at one of these tactics that are used as building blocks, the `getGoal` tactic. It is shown in listing 21. The tactic works as follows. At its core, the tactic is an `updateState` instruction, and thus carries a function that receives the state, and produces a product of the next proof state and next instruction. But in this tactic we have used the `useState` function, which allows us to consume the state, but not modify it. It *does* require us to produce the next instruction to execute.

The instruction we generate depends on whether a focused goal can be found. The `getFocusedGoal` function tries the focused goal that is stored within the proof state, but falls back to the first goal in the goals list if no focused goal is set. If the list of goals is empty (meaning that the first goal does not exist), `getFocusedGoal` will indicate that no focused goal exists by returning an empty optional value. The optional eliminator `maybe'` is used to branch on the existence of a focused goal: if the goal exists, we wrap it in the `Attic` type using the `return` function. If the goal does not exist, a fatal error is generated. The `getGoal` tactic is used fairly often, since the information contained within the goal (and often the goal itself) is necessary for use in other tactics.

Another example of a tactic that deals with the current proof state is the `addGoal` tactic, shown in

```
-- Construct trivial iterators
someNaturals = fromList $ 1 :: 2 :: []
someStrings = fromList $ "A" :: "B" :: []

branching : Attic ⊤
branching = do
  n ← branch someStrings
  m ← branch someNaturals
  log "branching" Debug $
    errStr $ "(\"" str+ n str+ "\" , " str+ (showℕ m) str+ ")"
  failure $ errStr ""
```

Listing 23: An example tactic that prints the values it obtains from branching instructions. Printing is done using the `log` tactic. The `_str+_` function is the string concatenation function.

```
("A" , 1)
("A" , 2)
("B" , 1)
("B" , 2)
No solution was found (options exhausted).
```

Listing 24: The output from the logging demo in listing 23. The debug buffer and evaluation result are combined.

listing 22. As its name implies, it adds a goal to the list of unsolved goals. This tactic does change the current proof state, and as such uses the `updateState` instruction directly. It returns a product of the next instruction, and the accompanying proof state. The state that is returned is simply the old proof state, but the list of unsolved goals has been changed to one where the new goal has been prepended to the existing list of goals. This is achieved using Agda's syntax for records; we can create a new record based on an existing one, and change only some fields. The next `Attic` instruction we generate here is more interesting. It is an invariant that checks whether the goal does not already exist, and that the goal was not already solved previously. If the invariant holds, the instruction is a no-op. If the invariant does not hold, a fatal error is generated and execution is halted immediately. After all, if the goal already exist in the proof state, adding it again is a user error. Abstractions such as `getGoal` thus make it easier to perform common operations without having to deal with low-level instructions, but can also make the program safer by using invariants that rule out certain incorrect behaviours.

There are various other tactic programs that are not intended to be used as standalone programs, but are meant to be used as building blocks to build other tactics. For example, there is a `solveGoal` tactic, which accepts a `Goal` and a solution `Term`. It will then use these two values to construct a new `Solution` object, adds that to the list of solutions, and removes the goal from the list of unsolved goals. It includes an invariant which verifies that the goal exists and is unique within the list of unsolved goals. Another tactic is the `admitAll` tactic, which takes all unsolved holes, and turns them into solutions, where the solution is the original hole. These solutions are then added to the list of solutions. In the end each hole will thus be unified with itself, which is essentially a no-op and leaves those holes as open interaction points. This is useful for debugging purposes, because Attic verifies that no unsolved goals remain, and as far as Attic cares, all goals are resolved. Agda will still list them as unsolved. The tactic can thus be used to expand the macro to a term where the holes are still present.

For debugging purposes, a tactic named `log` exists. It is a small abstraction over Agda's internal `debugPrint` function, but produces values of type `Attic ⊤`, so that can be used in tactic programs. The debug printing function is at its core an operation with a side effect; the information we would like to print will be shown in the Agda Debug Buffer. The `log` tactic requires three arguments, the first is a string describing the component that is doing the logging, the second is the log level (debug, info, error). Finally, the body of the log is a list of `ErrorPart`, Agda's built-in data structure to describe errors. It supports strings, terms, and qualified names. These contents are passed to `debugPrint` as-is, as it

uses the same data structure to describe its messages. An example of the `log` tactic usage is shown in listing 23. Two iterators are defined, both producing two elements. We also have a tactic that branches on both of these iterators, turns the values it gets into a string representation, and then logs these strings. Finally, the tactic produces a soft failure so that backtracking will be initiated. When using this tactic on a hole, Agda shows the output in listing 24. Evaluating this tactic resulted in the expected values being printed, and the tactic did not find a viable solution.

5.7.2 Combinators

In addition to these building blocks that perform relatively simple tasks, it is also possible to build instructions that are more complex. The examples so far have been smaller tactics that abstract away state updates, or perform tedious operations that could also be done by hand. Implementing larger programs requires more care, but can be very valuable when complex tactics need to be developed. This is where combinators come in; functions that can combine tactics to produce new behaviour. In other proof assistants these are sometimes referred to as "tacticals" [1].

We will first take a look at the `multi` function, which accepts a list of goals and a single tactic, and then evaluates the tactic on each of these goals. The type is given below. A polymorphic version of the unit type is chosen here so that it is compatible with all universe levels; its level is automatically inferred from the implementation of the function.

```
multi : ∀ {ℓ} {A : Set ℓ} → List Goal → Attic A → Attic ⊤ℓ
```

This function arose from the need to apply the same tactic to a list of goals. Semantically, it should be possible to execute the tactic program on each of the goals in parallel; the holes must be independent. The instruction was originally meant to be a constructor of the `Attic` type, with the interpreter running the tactic on each goal independently, and combining the results afterwards. This turned out to be impossible to implement, as the `runAttic` function must also deal with the rests generated by branching instructions. If the single tactic that we apply to every goal is a branching tactic, combining all their rests so that each possible permutation is evaluated is not feasible.

Fortunately, we can avoid this problem by *not* implementing `multi` as a core instruction. If instead of parallelising the evaluations, we serialised them, we could use the default chaining operation to automatically evaluate all possible permutations. The `multi` function can therefore implement the exact behaviour we want to have, but without a dedicated instruction. By manipulating the proof state carefully, it is possible to run the tactic on all the goals serially.

For each goal, we can generate an `updateState` instruction which captures the starting state. It returns a nearly empty proof state, which has that goal set as its only unsolved goal, as well as its focused goal, and contains no solutions. The instruction it returns is one that runs the `Attic` instruction that is to be executed for each goal, with another `updateState` instruction chained to it. This additional instruction captures the state *after* the tactic was evaluated. It will check whether the original goal now has a solution. If the solution exists, it will take all solutions from the *after* state, add them to the *starting* state, and return this new state. If the solution does not exist, it will return the *starting* state without modifications. The instruction it returns will always be the program produced by the recursive call to `multi` (with the tail of the goals list), so that all goals will be processed by the tactic.

The tactic is evaluated in relative isolation, so it is not possible to access solutions that were previously found, or even other goals. This is not a use case that Attic was designed for: goals are meant to be mostly independent. If one tactic fails, the entire `multi` program will produce a failure, due to the sequential nature and implementation of the `Attic` composition. We consider this to be working as intended however, since the same behaviour would have been implemented for a 'parallel' version of the function. It can be worked around by using the `tryCatch` instruction, which can be used to catch soft failures and produce a success instead.

Using the `multi` function, we can implement another tactic combinator that works on multiple goals. In Coq, the operator ; (semicolon) exists. When used as $t_1; t_2$, its meaning is informally described as "apply t_2 to every subgoal produced by the execution of t_1 in the current proof context" [42]. We can now

emulate this tactic using the functionalities provided in Attic. The largest challenge is deciding which goals are generated in t_1, and then running t_2 on only those goals.

In Attic, this tactic has been named `_:▷+·_`, and is defined as a left-associative infix operator, allowing for chaining. When used as `a :▷+· b`, it will produce a tactic that first captures the starting state. It will then execute tactic `a` as usual, but it will chain another instruction to it to capture the ending state. The goals that exist in the before and after states are examined, and all the goals that exist after `a` was evaluated, but not before, are selected. This list of goals is then used as the input to the `multi` tactic, along with the tactic `b` that is to be executed on all these new goals. If no new goals are created, tactic `b` is not executed, but the whole `:▷+·` operation succeeds.

This method of computing which new goals have been created is not entirely safe. In theory, the first tactic `a` can do anything with the current proof state as it pleases. It could clear all goals, add bogus goals that do not relate to the original hole, or modify existing goals (by replacing them, but keeping its identifying metavariable the same). The best that the `_:▷+·_` function can do to work in the most general case is to compare the lists of goals, and decide which goals are new. The tactic `a` is simply treated as a black box. In practice, this approach works very well, and as long as tactics do not perform any unsafe state modifications (or indeed any other changes we have not thought of), this combinator works as expected. Due to implementation details of `multi`, the `b` tactic does not have access to previously found solutions, or other unsolved goals.

6 Evaluation

The work which most closely resembles, and has served as a direct inspiration for Attic, is the Ataca project originally developed by Jesper Cockx [9]. Ataca aims to fulfil the same purpose as Attic: both share the goal of providing proof authors with tactics, but more importantly the ability to create new tactics. Both libraries make use of Agda's reflection API to carry out their tasks. Although their inner workings are not the same, both libraries share a similar architecture. They take different approaches and make different choices in certain trade-offs. As a result, we believe it makes sense to directly compare the projects, as both have their own strengths and weaknesses that the other does not. This section will outline a comparison between the Ataca and Attic projects. We will describe some of the similarities between the projects, but the main focus here will be the points where both projects differ from one another.

Ataca has served as the main inspiration for the Attic project. In fact, some of the core concepts are directly adapted. As described in section 5 (which details the inner workings of Attic), Attic uses a system where tactic programs are a chain of instructions. These instructions are then interpreted by a function that can execute and evaluate them. Ataca uses a similar implementation. Some instructions are present in both projects (albeit with a different name), whereas others exist in only one of the two projects. Still, the core concept of dynamically generating programs that are then interpreted, with the goal of producing a solution for the hole the program executes in, was inspired form the Ataca implementation. The mechanism of turning such a chain of instructions into a `TC` operation is also inspired by Ataca, but behave differently. The `toMacro` function in Ataca simply prints some diagnostic output and runs the tactic program. On the other hand, the `asMacro` function in Attic first collects the necessary data to create the initial goal and proof state, and then starts the process of finding and trying out the candidate solutions.

6.1 Performance

Upon introduction to the Ataca project, it was mentioned that the performance degraded as tactic programs became bigger, or had to do more searching. One of the design goals of Attic was ensuring a consistent performance. This means that the evaluation time of tactics should not become extremely long as the tactic instruction grows. Historically, no attempt was made to investigate why the runtime of Ataca tactics would become longer. This thesis will also not investigate this phenomenon, beyond some surface investigations. However, it is thought that the performance issues are related to the frequent use of compiler services, requiring many type checking operations.

As a result, in Attic the number of direct interactions with the compiler are minimised. This is achieved by changing what data is stored by Attic as the tactic programs are evaluated. For example, in Ataca most tactics tend to start with a call to `inferType` on the hole. The Agda compiler will then decide the type of the hole, which is vital information when a term must be constructed to fit in the hole. Attic, on the other hand, will ask the compiler what the hole's type is at the start of evaluation, and then passes this value around using the `Goal` record. As the goal is stored, we no longer need to ask the compiler to infer it. When new goals are created during evaluation, these are also assigned a goal type value if the type is known.

Another example is the tracking of the current context. In Ataca, the context is always obtained using a call to the `getContext` function. Obtaining the context in this way may occur several times during the evaluation of a tactic program. Attic again changes this approach, and stores the context within its `Goal` data structure. It will collect the context from the compiler at the start of the program, and from that point onward the tactic author is responsible for ensuring that the context is set to the correct value. Reducing the dependency on the compiler interface should speed up tactic evaluation.

To check this, we have set up an experiment to measure type checking times, and compare the resulting times of Ataca and Attic. Both tactic libraries are not very extensive, and only contain basic tactics. If we wish to directly compare both libraries, we should only use tactics that are supported by both libraries. We have thus settled on using the `intro` tactic (which introduces a local variable/generates a function) and the `exact` tactic which fills in an Agda term. These tactics exist in both libraries. The exact test code, as well as the numeric results are given in appendix B, but we will discuss the experiment itself here.

```
introTest0 : ℕ
introTest0 = solution

introTest1 : ℕ → ℕ
introTest1 = solution

introTest2 : ℕ → ℕ → ℕ
introTest2 = solution

introTest3 : ℕ → ℕ → ℕ → ℕ
introTest3 = solution
```

Listing 26: The automatically generated experiment test cases.

```
repeat : ℕ → Tac ⊤ → Tac ⊤
repeat zero tac = return tt
repeat (suc k) tac = tac >> repeat k tac

repeatTry : ℕ → Tac A → Tac ⊤
repeatTry zero tac = return tt
repeatTry (suc k) tac = (tac >> repeatTry k tac) <|> return tt
```

Listing 27: Different implementations of a tactic repeater. `repeat` is originally implemented in Ataca, and we have added the `repeatTry` implementatation. The `Tac` type is equivalent to the `Attic` type.

The idea here is quite simple: through automated code generation, we create a number of definitions. Each definition is a function with `n` parameters of type $\mathbb{N}$, and returns a value of type $\mathbb{N}$. We can generate several of these by ranging `n` from 0 to 100, and placing them in the same file; see listing 26 for an illustration. By generating a large number of test cases with increasing complexity, our hope was to observe a meaningful difference in performance between Ataca and Attic. Although we expected there to be a slight difference, we did not expect this difference to be very meaningful.

The tactic we used to generate the term value for these definitions was fairly simple. In both Ataca and Attic we repeatedly applied the `intro` tactic, up to 100 times. This was followed by a single application of the `exact` tactic, which simply fills in the value 5.

We ran these tests on Agda 2.6.2.2, on Ubuntu 20.04.4 LTS, which itself ran within Windows Subsystem for Linux 2 on Windows 10 21H2. The host machine uses an Intel® Core™ i7-8750H CPU and has access to 16 gigabytes of RAM. The test module was named `Perftest.agda`. We obtained the type checking times by first deleting the Agda interface files (`.agdai`) which store the results of the type-checking process [6, Interface files], and then running `agda -vprofile.modules:100 Perftest.agda`. This option enables profiling functionality and makes Agda print how much time it spent (in milliseconds) on type-checking each module, allowing us to exclude the time it spent type-checking imported modules. We have used this metric as our data source.

It turns out that the observable difference was enormous, far beyond what we initially expected. This difference turned out to be unfair, since the implementation of repeated application that Ataca uses by default is not very optimised. It will execute the tactic 100 times exactly. Once the `intro` tactic no longer succeeds, it will never succeed again, but Ataca continued regardless. Attic uses a different implementation, where the repeated application is stopped once the tactic fails the first time, which can save a lot of time. After all, if a tactic fails once, applying it again should generally not work. The implementation difference is shown in fig. 6. To eliminate this difference, we have implemented the optimised version in Ataca as well. We have measured the performance for both the unoptimised and the optimised version, in both libraries. We refer to these as the slow and fast version respectively. The performance difference between these implementations illustrates that, although their observable behaviour is the same, their runtimes are very different. Writing code that does not perform lots of useless work remains imperative.

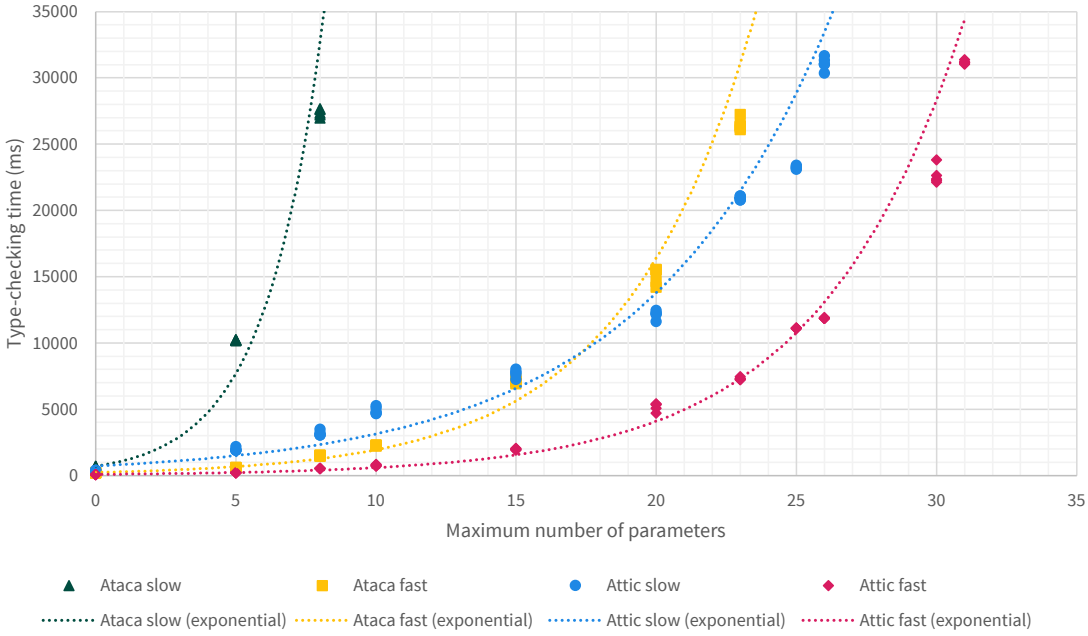


Figure 6: A comparison of type checking times between Ataca and Attic.

An interesting point to note is that, in some cases, the Agda compiler does not seem to finish type checking within a reasonable time. Especially with the slow implementations, Agda starts using unreasonable amounts of memory (around four gigabytes) and does not finish type checking, even if left undisturbed for several minutes. CPU and memory usage remains high, but the Agda executable does not exit. The slow Ataca implementation reached 8 parameters, but did not exit at 9. The fast Ataca implementation reached 23 parameters. The slow Attic implementation reached 26 parameters, and the fast Attic implementation reached 31 parameters before Agda seemed to hang on type checking.

A scatter plot of the performance is given in fig. 6. This scatter plot was not made using the average values shown in appendix B, but uses all measurements we have collected. We have drawn exponential trend lines through the points, but these primarily exist to illustrate and guide visually, as we are not sure that the increase in time is accurately characterised by an exponential function. We also wish to point out that the times we have measured here are cumulative; every successive measurement includes the time of the previous measurement. The slow Ataca implementation performs the slowest, and is the earliest to stop working. The fast Ataca implementation starts out faster than the slow Attic version, but they soon switch places with Attic overtaking Ataca. The fast Attic implementation outpaces the others, and is the last to cause Agda to hang.

These results should be taken with a grain of salt. The tactics are not particularly complex, and do not involve any of the branching and backtracking mechanisms that Attic features, but Ataca lacks. Conversely, we have not used tactics that deal with complicated types, as Attic cannot deal with these yet, while Ataca can (this will be discussed in section 7.2). We are also unsure why Agda seems to hang in some of our test cases, this might require further investigation.

6.2 Data storage

Compared to Ataca, Attic stores some additional information during evaluation. Although this optimisation was originally intended to increase the runtime performance of the tactics (see section 6.1), it also enables greater flexibility when implementing a tactic program. Most notably, the ability to set the context on a per-goal basis turns out to be very useful. While this machinery does require more bookkeeping on the side of Attic, the flexibility we gain makes up for this.

Take for example the `destruct` tactic. This tactic takes an existing hole, and fills it with a term that

discriminates on a value, so we may get multiple branches. In each branch of the resulting term, a new hole is created. This allows us to continue evaluating tactics for each branch of the destruct term. Some of the resulting patterns in the branches might bind new variables in the context. We would like the bindings of each branch to be available to each of their respective holes (and the associated goals) that are created.

The ability to set the context per-hole is something that is not currently available in Ataca. This means that setting the context of these holes is not possible. Attic stores the context of a hole inside the `Goal` structure that is used to represent that hole. Choosing to focus on a different goal makes it possible to switch between contexts easily, without having to ensure that the current context in which we are operating (which can be modified with `extendContext` or `inContext`) is set to the correct environment. Interactions with the compiler that depend on the context are wrapped such that the context is always set using `inContext`. Other operations within Attic abstract away from this notion of a global context, and keep the context local to the goals.

6.3 Branching

Ataca and Attic use different methods to allow the selection of multiple options. Ataca comes with the `chooseTac` instruction, which takes two `Tac` programs. It will try to execute the first program. If that succeeds, the resulting state is preserved. If the first program fails (that is, it produces no solutions), the state is rolled back and the second program is executed. The automatic state rollback happens because of Agda's built-in `runSpeculative` function, which accepts a `TC` operation that returns a Boolean value as its output. If the value is false, the state is rolled back. If the value is true, the Agda compiler will retain the state. By chaining and nesting the `chooseTac` instruction, it becomes possible to produce a list of different options that can be tried. It does not support backtracking, however.

Attic on the other hand takes a different approach. It has the `branch'` instruction, which contains an iterator of `Attic` tactic programs. Each of these tactic programs could generate a (partial) candidate solution for the hole(s) they act on. If the program fails to generate a solution, the next program (if any) is obtained from the iterator, and executed. If a program manages to produce a solution, the solution is submitted to the Agda compiler for unification. If this process succeeds, a solution was found and we are done. If the proposed solution is rejected, the `branch'` iterator state is used again to generate the next tactic program which, if it exists, is then evaluated to see if it yields a solution.

6.4 Unification methods

The timing of unification between Ataca and Attic differs greatly. Both libraries use the `unify` function that is provided by Agda. This function accepts two terms, and unifies both terms, potentially solving metavariables in the process. This is also the function that allows us to set the content of a hole, if we unify the hole metavariable with a term. The main difference is in when the function is used. In Ataca, unification happens instantly. If a tactic program has a candidate solution for an open term position, it will immediately make a call to `unify`. Attic however is more lazy; if a tactic produces a candidate solution, this solution will be added to a list of solutions. Once the tactic is fully evaluated, the list of solutions is processed one by one, and each solution is unified with its associated metavariable. Within Attic, this is known as *deferred unification*, since the final step of actually committing to a solution is only done when an entire branch of the tactic program is evaluated.

Both approaches have their own advantages and disadvantages. In Ataca, a possible solution is unified immediately. This is relatively straightforward to implement, it ensures that the compiler knows about this value, and we will know about type errors instantly if the value results in one. Ease of implementation should be readily apparent, because whenever we come up with a candidate solution, we can simply use that solution, and if it does not work we can catch the error and handle it appropriately. The second point is the use of the term when obtaining type checking information. If a term is filled in, the compiler can often use that value to make new inferences about the types that are expected in other position, and refine them further. Finally, it is useful to get immediate feedback if it turns out that a value cannot be used. Trying to unify an invalid value with the existing solutions is a clear signal that this path should not be explored further, if other choices can be made.

The deferred unification approach has its own advantages. The main advantage is that it gives proof authors greater control over how non-deterministic tactics generate solutions. This makes Attic more flexible. Using the primitives that Attic provides for iterators (as well as some of tactics that encapsulate iterator functionality) many different combinations can be constructed. Backtracking can also become cheaper; instead of having to rely on the compiler to indicate that something is awry, we can rule out entire values easily before a unification attempt is made. These benefits aren't easily achieved with immediate unification.

The biggest drawback of deferred unification might be the lack of type information that the compiler can provide when deferred unification is used. Figuring out what the type of a goal should be can be very challenging in some cases. Inductive data types have recursive definitions, so some of their constructors have parameters that are the same type as the constructor's type. Internally, Agda represents this as a variable that points to a location in the type's telescope. We cannot use this type directly, because that telescope may not always be available. Instead, it is possible to normalize the types, and substituting any variables that reside in them. Such a normalisation is complex, and Agda does not offer an API that does this for us. It could be implemented manually in Attic, but the information we gain from it is likely not as complete as the rich type information that Agda offers. For example, determining indices on types would require even more pattern matching beyond mere substitution.

6.5 Backtracking

Another difference between Ataca and Attic is the backtracking mechanism. Ataca does feature some backtracking. This is achieved using an instruction that attempts to execute one tactic program first, and if that fails it tries the second tactic. If a tactic produces a failure (explicitly, or due to a unification failure) this mechanism is triggered. Many of these failures will be the result of catching errors that were thrown from the type checker.

A failure mechanism also exists in Attic, but Attic makes the distinction between two types of errors. It supports hard errors which indicate a failure state from which no recovery should be attempted, and the interpreter will not attempt to backtrack from these. Alternatively, there are soft errors which simply indicate that evaluation of the current branch should not continue. These errors are recoverable and as such the interpreter will continue with another branch, if one exists. Most tactics will produce the soft failure (not applicable, no solution exists), whereas the hard failure is reserved for erroneous states (invariant does not hold, no unsolved goal exists). Agda's type checker should not produce errors during evaluation, only during the unification phase.

7 Future work

In this section we will discuss some of the work that could be done to improve the Attic project. Some of these topics consist of features for which there was not enough time to complete them in the timeframe of this project. Others are suggestions that would improve the reflection API in Agda. And finally, some are bugs in the project that were known at the time of writing, but have as of yet not been resolved.

7.1 Propagation of solved metavariables

As goals get solved by tactics, the metavariables that they represent get solved. This solving is done at the end, using the `unify` function from the compiler API. A goal is marked as solved by removing it from the list of unsolved goals, and adding a solution (that points to the original goal) to the list of solutions. Other goals remain unchanged. This is a problem, because these other goals may depend on the metavariable that we have just solved.

Instead of leaving the other goals unchanged, we should propagate our solution through the other unsolved goals. If any of these goal's types contain the metavariable that we have solved, we should substitute this metavariable with a concrete (or rather, more refined) term. This currently does not happen. If deferred unification was not used, and immediate unification was used instead, we could simply query the compiler for the expected type. Because of deferred unification, we cannot do this. To implement this properly, we would have to implement some of the functionality that the compiler's type checker already implements, but is not accessible to Agda code.

7.2 Normalisation of types containing variables

Attic was built on the assumption that the type of a hole (and by extension, a goal) is known. This way, each tactic could simply examine the goal's target type and decide whether it applied to this goal, and if so, how to proceed to generate a term for this type. In practice this turns out to be harder than expected when generating subgoals. Suppose that we use a tactic to generate a value of type `Vec Bool 2`, and use repeated application of the `constructor` tactic to fill it. This tactic should, through repeated trial, find a value that fits in this hole. The correct first step of the solution is to use the `_::_` constructor, which will spawn two subgoals. One subgoal has the type `Bool`, and the second has the type `Vec Bool 1`.

In practice this turns out to be much more complicated. The `_::_` constructor has a number of parameters, both explicit and implicit (see listing 28). Internally these variables depend on each other, which Agda represents internally using variables in De Bruijn notation. See fig. 7 for an illustration.

```
_::_ : {a : Level} →
       {A : Set a} →
       {n : ℕ} →
       (x : A) →
       (xs : Vec A n) →
       Vec A (suc n)
```

Listing 28: The full type of the vector `_::_` constructor.

When Attic evaluates the constructor `_::_`, it will naively decide that the type of the second hole must be `Vec (var 2) (var 1)`, which is simply the type that the compiler gives us when we query the constructor type and match these up with the type arguments on the original vector type. This is perfectly fine when these variables are actually in scope (as sorts), but when we store this type in a goal as-is (without the type telescope) Agda will rightfully complain that De Bruijn index 2 does not exist. Our problem would be solved entirely if we could resolve this type to `Vec Bool 1` before turning it into a goal, as this would eliminate the variables from the type, and we would once again have a concrete type.

In practice this turns out to be quite complicated. We would need to traverse the type expression and substitute all `var` terms by the term they actually point to. This would not be entirely trivial, but it would be doable. We can step through each of the type parameters one-by-one, and replace the variables, while

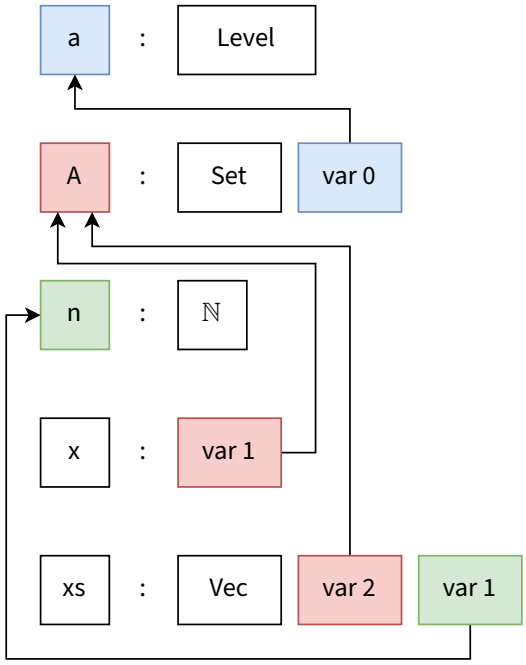


Figure 7: Internal representation of the parameters of the `_::_` constructor of the `Vec` type in Agda's standard library, including implicit parameters. Their types are shown as well. The parameters are `a` (level of the data structure), `A` (the type of elements it contains), `n` (the length of the vector), `x` (the head of the vector), `xs` (the tail of the vector). De Bruijn indices are indicated with arrows and colour coded.

we also extend our telescope at each step to make sure that the De Bruijn indices remain in lockstep with our parameters. More difficulties appear when we have to start resolving the type indices (which is, for the `Vec` type, only the length parameter `n`). With every next tail, the length value decreases. Ensuring that this solution works for any indexed data type is not trivial, and would require pattern matching to figure out what the next index should be. To make matters worse, types are sometimes simply not concrete. Universally quantified types will always contain variable terms, as they necessarily refer to the variable that is generalised over.

A solution to this problem is not obvious to us. It may be possible to implement an algorithm that can handle all these cases well. But at the same time we must wonder whether implementing such an algorithm in Attic would be worth it. After all, the Agda compiler already handles these cases. Perhaps it would be favourable to build a hybrid system, where unification typically happens at the end (keeping the deferred unification for most cases) but also allowing some terms to be unified early. Once unification of such a term occurs, we could interrogate the compiler about our existing unsolved goals, and see if it has refined them, and if so, adopt these refined types and replace the ones we currently have stored.

7.3 Garbage collection of metavariables

When tactics create new subgoals, they must create a metavariable which represents the subgoal's hole. Metavariables can only be created through compiler interactions, as they are part of the compiler's internal state. When metavariables are not solved, the compiler will emit warnings for them, and compilation will fail. As such, section 5 describes a rollback mechanism: if evaluation of a branch does not succeed, the entire compiler state is rolled back (using the `runSpeculative` operation). The rollbacks act as a garbage collection mechanism for metavariables that are no longer in use.

However, it turns out that this approach is not correct. When the Attic interpreter encounters a branching instruction, it will evaluate only the first branch. The other possible branches are stored away as closures which can be evaluated later, such that computation resumes where it left off. The unfortunate reality is that this approach does not take side effects into account.

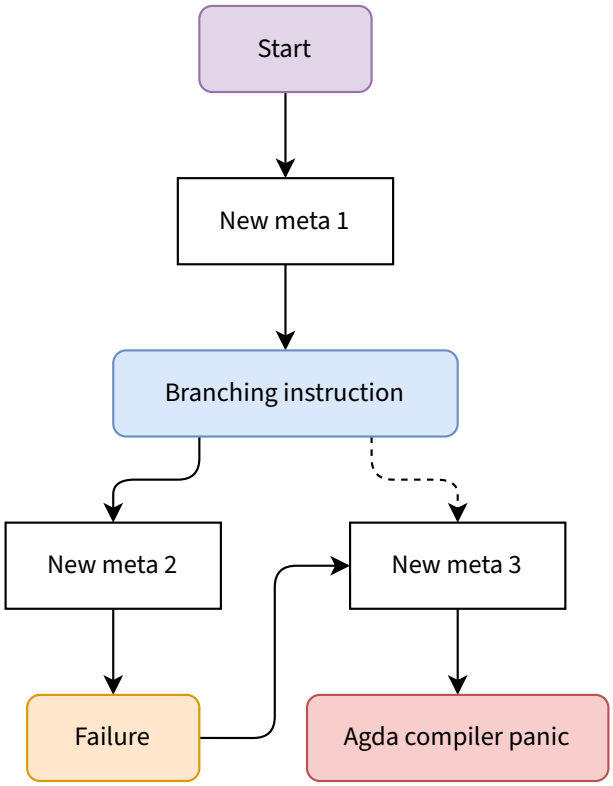


Figure 8: An evaluation flow displaying how incorrect garbage collection results in an Agda panic.

This is illustrated with an example in fig. 8: we have a tactic which creates a metavariable (*meta 1*) and then branches. The first branch creates its own metavariable (*meta 2*) but ultimately fails to produce a useful result. The compiler state of the entire path is rolled back, so that all metavariables we created along the way are destroyed. Evaluation resumes in the second branch, which also generates its own metavariable (*meta 3*). But when this path attempts to use *meta 1* in any way, the compiler immediately panics and halts computation, telling us that the metavariable does not exist. This is true; the rollback destroyed the metavariable, but Attic is not aware of this fact.

Obviously, *meta 1* should still exist in the second branch, but *meta 2* should not. The best way to solve this would be to introduce rollback opportunities in more program locations, so that more granular rollbacks can be performed. Such a solution would allow us to delete *meta 2* from existence, without affecting *meta 1*. Regrettably, we are unsure how such granular rollbacks can be implemented given the current evaluation design of Attic, and the available API functions that Agda exposes. Perhaps it would be best to not use closures at all, but to simply restart the entire evaluation for every branch. This would at least allow us to start from an empty state, so there would be no metavariable pollution from other branches.

An alternative to the granular rollbacks would be to keep track of the metavariables that have been created and ensure that they are, in some way, cleaned up when the tactic evaluation finishes. This would require some refactoring of the current Attic evaluation mechanism, but would likely not require any significant architectural changes. We can then, at the end, filter out which metavariables were never paired with a solution, and ensure that they no longer result in compiler warnings. The best way to do this is to ask Agda to forget about the metavariable, but this is not currently possible. Instead, we could unify the metavariable with a simple term (such as `tt`, `true` or `zero`)[6]. Such a solution ensures that the metavariable is no longer unsolved, and since it is not used anywhere, we can fill in whatever term we please. We do believe that this solution is not a very nice one, and ought to be described as 'a bit of a hack'.

[6]A caveat we wish to note is that unifying with a simple term only works if that term is an instance of the hole's type. If the term is of a different type, Agda will reject this unification. In Attic this would work well, because all generated holes have the type `unknown`. Any closed term can thus be used as a solution. In the general case, this is untrue.

Alternatively, changes could be made to the Agda reflection API to allow for the deletion of unused metavariables. Instead of unifying the metavariables with a useless term, we could simply ask Agda to delete them. This is a potentially dangerous operation, as deleting a metavariable that is used elsewhere may result in the panics we described earlier. An nicer approach would be to ask Agda to 'forget' about metavariables that are not used anywhere, so that we do not get warnings if one is not solved. This approach could still cause problems for users of the reflection API, if they ask Agda to garbage-collect the unused metavariables and then try to use these deleted metavariables. We do not know how feasible such functionality would be, and if the current Agda compiler implementation would allow for it.

7.4 Name preservation in the reflection API

Internally, Attic keeps track of the (string) names of variables it creates. This is achieved because the data type that Attic uses to represent telescopes[7] has a string name annotation for each variable. When Attic generates a match pattern on a constructor, it will use the names of that constructor's parameters internally. This is admittedly a heuristic, as the reflection API does not allow us to specify a name when declaring match pattern variables, only an index. When Agda pretty-prints the pattern to source code, it will try to use the same name as the constructor's parameters specify, but if that name is already taken it will generate fresh names. The functions that deal with telescopes (`getContext` and `inContext`) also do not use any string names.

In most cases this is not a problem. Attic generates variables using De Bruijn indices, and uses these internally for all purposes, as they are not ambiguous. There is an edge case that can be encountered when we *elaborate and give* a tactic in which not all metavariables are solved (and thus result in holes in the source code). Although these metavariables were created with the correct context, Agda's pretty-printer can (and will) assign different names to the variables in patterns and the variables that are associated with a metavariable. For example, if we match on a natural number, and generate a pattern match for the `suc` constructor, Agda will pretty-print this pattern as `(suc n)` . Attic correctly prepends this variable to the telescope. But when we then execute the *Goal Type and Context* editor command in the generated hole, Agda will say that the first variable in the context is " `x : ℕ` (not in scope)", as Agda has internally assigned another variable name ('x') to the new variable that Attic prepended to the telescope.

This is not a major problem, and will be automatically corrected when we reload the file. Reloading causes Agda to match the names in the metavariable context with the names in the pattern. Still, it is a mildly annoying problem that cannot be worked around by Attic. Fixing this gap requires a change to the Agda compiler API, providing the ability to specify names of variables. This would not be a trivial change, however. Adding string names here and there would require breaking changes to the API, and it would likely introduce problems, as the ambiguity of strings are the main reason that De Bruijn indices are used in the first place.

7.5 Less restrictive type assignability

Some tactics require a function that decides whether a value can be used in a certain place. This requires checking type compatibility. Take for example the `assumption` tactic, which goes through the variables that are already in scope, and tries to solve the current goal with each of these variables. Should any variable fit, it will certainly be found. However, we could reduce the number of tries by first filtering the the list of variables in scope to only those that have a chance of fitting. After all, a natural number could never be used where a Boolean value is expected.

This could be decided by a function. The function could take two types, and then decide whether they are 'compatible' enough to warrant trying. We are purposefully not defining this compatibility, because it is intended to be a heuristic that is good enough for most purposes. The actual assignability should always be checked by the compiler. This brings us to the issue at hand; Attic currently has a rudimentary (and incorrect) assignability check, which uses the standard library to check if types are definitionally equal. This check is too restrictive, and rules out variables that may actually fit, even though their types (which includes implicit arguments) are entirely equal. By removing the filtering behaviour, the `assumption`

[7] The telescope that Attic uses is simply the `Telescope` type as defined in Agda's standard library.

tactic behaves correctly again, but it may suggest solution that could be quickly ruled out by a heuristic. In the future, this heuristic could be added.

7.6 Less restrictive multi-operator

The `multi` operator (see section 5.7.2) evaluates the same tactic on a list of goals. It currently creates an entirely new proof state for each goal it evaluates, which contains only that one goal in the list of unsolved goals, focuses that single goal, and contains no solutions. Tactics thus do not have access to any previously found solutions, and are not able to make changes to the existing proof state. It could almost be said that the tactic runs in some sort of 'sandbox', completely isolated from the other tactics. This also makes it easier to observe what the tactic has done, and copy the solutions it has found to the original proof state. Perhaps this could be improved in the future, so that tactics that are run within `multi` retain access to the original proof state, including its unsolved goals and found solutions.

7.7 Development of more tactics

A fairly obvious improvement that we believe still should be touched on is the lack of usable tactics. Attic contains a few tactics, but nowhere near enough to be a viable tool to prove complex theorems. Attic was not intended to be a fully featured proof assistant built on top of Agda. Instead, its purpose is to be library upon which new approaches to tactics can be developed, with which authors can develop new tactics. Still, shipping with a number of tactics built-in would be a welcome addition.

7.8 Improvements to `cartesian`$\mathbb{N}$

The Cartesian operator that was implemented for iterators (section 3.3.2) was originally implemented as a demonstration of the power of iterators, showing that non-trivial operations on these data structure are feasible, while adhering to the totality requirements that are set by Agda. The operation is already quite general; it supports an arbitrary number of iterators, each with their own type, and requires users of the operation to provide a function that combines each of these values into a single value. The only restriction is that each of these types must have the same level. It is almost certainly possible to generalise the type of the operation even further, such that every type can have a different level. This was not implemented in Attic because we have not deemed it necessary to do so for our purposes, but a more general implementation would be desirable.

A small performance improvement could be achieved by storing the iterator elements differently. Every time the source iterators are stepped, the resulting output values are appended to the end of the vectors we use to store the obtained elements. This makes it easier to index the elements later. Appending to the right is a relatively slow operation because of how lists are implemented. By using left-append only, and changing how indexing is performed, this slowdown can be avoided. We do not expect this change to result in a significant speed-up.

8 Related work

This section describes some of the previously existing works that relate to the problem that this thesis attempts to solve. We have already discussed the main inspiration, Ataca, in section 6. This section aims to give a further overview of related techniques and approaches.

Before diving into the related work, we first wish to note that Agda does have a mechanism for automated proof search [6, Automatic Proof Search (Auto)]. This function is exposed to editors that are aware of Agda through editor commands, where it is typically named 'Auto'. It is also referred to as 'Agsy', which is the external tool that implements the search algorithm. Any solution that is generated by Auto is still checked for correctness by Agda. Auto cannot usually find solutions for larger problems, and there are various other limitations [6]. It ships with Agda and is written in Haskell, making it hard to customise, and in some cases it will generate solutions that turn out to be invalid. Despite these limitations, it is still a useful tool for both proving and programming.

8.1 Auto in Agda

Using Agda's reflection interface to implement programs that search for solutions is not a new idea. In fact, this appears to be the primary reason for its existence [6]. An earlier approach at simplifying the user experience of creating custom tactics is 'Auto in Agda' [43]. (Note that this is not the *auto* functionality we described above.) In this work a library is developed which allows for proof search within Agda itself. It has the advantage of writing proofs in Agda itself, so users need not learn a new language in which they must express proofs. Instead of writing tactics directly, users of the library are expected to create *hints*, which are the rules that may be used during resolution. These hints can be added to a hint database, which is then fed into the `auto` tactic that creates the initial proof state. Both values are then passed into a `solve` function, which aims to find solutions to the open goals that still remain.

In some regard, 'Auto in Agda' is similar to Attic. Both are libraries that use the reflection API to prove theorems. Both systems generate a data structure of candidate solutions which they search through; a rose tree in 'Auto in Agda,' and an iterator in Attic. The way iterators are used could be seen as a flattened depth-first search through a tree. Overall though, the libraries are quite different; instead of trying to automatically find solutions using user-defined rules, Attic operates through user-defined programs to generate and evaluate candidate solutions, using non-deterministic tactic programs and compiler functionality. It may be possible to emulate 'Auto in Agda' in Attic by creating a `branch'` instruction that tries many different (possibly high-level) tactics.

8.2 Tactics in the Coq realm

The language L_{tac} (often written as Ltac when typesetting is not available) is the default tactic language for the Coq proof assistant [15, Ltac]. L_{tac} has served as a superficial inspiration of what the tactics in Attic should look like. L_{tac} aims to be "the real link between the primitive tactics and the implementation language (Objective Caml) used to write large tactics" [2]. It is designed in particular to deal with the details of proof automation that can be tedious for humans. A specification of the Coq internals is not required to write tactics, the code length is shorter and more readable, and as a consequence of the latter proofs are more maintainable [2]. Attic aims to fulfil these outcomes as well. By providing the building blocks to define user-created proof tactics, both L_{tac} and Attic enable authors to write large tactics that manipulate the proof state on a high level. The proof tactic itself does not have to be large; it can be composed of smaller tactics, resulting in a tactic that is shorter, easier to read, and thus easier to maintain.

We have chosen not to emulate L_{tac} in full; instead we have adopted only some of its syntax and semantics. The original L_{tac} has been described as having unclear semantics, lacking expressivity, and being error-prone and fragile, among other accusations [15, Ltac2] that we would prefer to avoid. Work on a successor to L_{tac} has already begun in past years; Ltac2 was meant as a proposal to replace L_{tac} [44] and has since grown to become an experimental part of Coq, with the authors encouraging users to test it [15, Ltac2]. Ltac2 is designed to be similar to L_{tac}, while avoiding its pitfalls and adding language features that are commonly found and desired in modern languages. For example, it has a type system, and allows for the definition of data types. It also provides well-defined monadic semantics and ensures

that all variables are bound statically. As our language is defined as a domain-specific language in Agda, we get most of these benefits automatically.

Additionally, a proof automation effort using reflection has also been developed for Coq. The R_{tac} system was developed to make it less cumbersome to build reflective tactics in Coq [45]. Instead of building full proof objects, computations that result in a proof term can be used to prove a theorem. R_{tac} tactics are fully evaluated inside Coq, just like Attic tactics (and other reflection programs) are evaluated inside Agda. Reflection tactics in Coq do turn out to be much faster than regular proofs written in L_{tac} (not manual proof term construction) in some cases, even though they are checked twice; once for constructing the proof, and once for checking by the proof kernel [46].

8.3 Edit-time tactics in Idris

Idris is another programming language with support for dependent types. Although it can be used as a proof assistant, it is intended to be a general purpose programming language, as an answer to the question "What if Haskell had *full* dependent types?" [47] and touts modern features that mature languages already support, but are not as prevalent in most proof assistants. Idris has the concept of editor actions backed by the compiler machinery, which is similar to the editor commands that we have seen in Agda. These editor actions are hard-coded into the Idris compiler, and adding new commands would require modifying the compiler source code, which is not very practical. To make the creation of custom editor actions more accessible, metaprogramming support was added to Idris by [3].

Much of the work done in [3] focuses on the compiler infrastructure that is required to support metaprogramming. This includes communication with the compiler, serialisation of data, and the definitions of data structures in the Idris standard library. Such a reflection API has already been added to Agda previously [48] and was used to create proofs through reflection [49], [50]. As a demonstration, [3] has added a two custom editor actions that can perform various tasks. The first action, when used on a value declaration, can add an accompanying definition clause. The second action is a tactic named Hezarfen, which is able to decide intuitionistic propositional logic theorems.

Although Attic and 'Edit-Time Tactics in Idris' do not attempt to accomplish exactly the same goal, there are some similarities between Attic and the Idris side of tactic implementations. In Agda the `TC` monad is used to specify computations that involve the compiler, while in Idris we use the `Elab` monad (which is a shorthand for elaborator reflection). Similar data and operations are provided by both compilers. Both libraries by default operate on the raw syntax terms that the compiler provides. But Attic introduces non-determinism and deferred unification to reflection-based tactics. Perhaps it could be said that the tactics in [3] are currently more similar to the Ataca project than they are to Attic.

8.4 Iterators: backtracking and streaming

In [51], a library named LogicT is presented with which backtracking computations can be added to any Haskell monad. This paper gives an overview of existing mechanisms for backtracking, including the `List` monad, as well as the `MonadPlus` interface. It extends these mechanisms with new operations that are intended to make the existing non-determinism evaluations fairer. For example, it adds an *interleave* operator which, as its name implies, takes two non-deterministic values and interleaves the results they produce. Another example are the *cut* functions, which allow for pruning of the search results. These implementations have been adapted to the iterator type, allowing users to create and combine iterators in fair ways. Iterators also serve as the primary data structure that is used for backtracking in the `Attic` type, fulfilling the same role as the backtracking mechanisms in Haskell.

Iterators are indirectly based on [33], specifically the data types `Step` and `Iterator` that are defined inside the Attic library, which are based on the `Step` and `Stream` types respectively. Unfortunately, we were not aware of this contribution until relatively late in the writing of this thesis, and have rediscovered many of the ideas and operations implemented therein. The same operations that exist on the `Stream` type can be implemented for `Iterator`: both support mapping, filtering and folding. The `concat` function is also implemented for nested iterators, and when composed with `map` serves as the implementation for the monadic `bind` function. We have added the *bound* field, which can give a rough estimate of how long the sequence should be, which can be a useful indicator for consumers of iterators.

9 Conclusion

In this thesis we have implemented a new library with which proof authors can write their own custom tactics for Agda. These tactics are based on the reflection system that Agda exposes, and can be turned into Agda macros with ease. In addition to a new way of writing macros, we have implemented a new data structure in Agda that can represent both finite and infinite sequences, opening up new possibilities in Agda. Together, these implementations can suggest solutions, check these solutions with the compiler, and fill holes in the source with correct terms.

From the start, there have been two main goals: the tactic system must offer backtracking mechanisms and let users control evaluation; and usage of tactics must not be too complex and low-level. Additionally, as a secondary goal, the performance of tactic evaluation should be good. With these goals in mind we have built a the machinery that can support a tactic language with non-deterministic evaluation strategies. We have also implemented tactics that abstract away many of the common operations that are required to generate terms that serve as proofs. This allows for the development of high-level proof scripts. These tactics can be composed, adapted or combined in other ways to facilitate complex problem solving behaviour.

To make searching for values easier, we have implemented an iterator type in Agda. With iterators it is possible to generate infinite sequences, but they may also stop producing new values. These iterators can be combined in interesting ways, and can be used for serious applications. Iterators also serve as the data type that is used for the branching instruction in tactics. As a result, tactics can theoretically branch on infinitely many values. Evaluation still remains finite, but may be incomplete.

Attic is still highly experimental, and the tactic library is still very incomplete. Basic theorems can be proven with Attic, but generating terms for larger and/or more complex types is still out of reach. This is mostly due to issues in Attic that have not been solved yet. We suspect that the most problematic issues so far are types containing variables (section 7.2) and the lack of solution propagation when metavariables are solved (section 7.1). These problems are both related to the deferred unification mechanism that Attic employs: usually one would rely on the compiler to tell us which type is expected, but here we cannot do that, as we have tried to minimise our interactions with the compiler.

One of the secondary goals of the thesis project was to build a system whose runtime performance is better than that of the Ataca project. From the small performance comparison, we can conclude that Attic appears to perform better in terms of run time. An extensive comparison is not yet possible; both Attic and Ataca are stubs of tactic languages, and both languages do not have enough tools available to significant conclusions about their performance.

Overall, we believe that this project has resulted in useful insights regarding tactics in Agda. Adding backtracking mechanisms to the core instructions means that tactics become much more flexible, and can allow tactic authors to find solutions more easily. At the same time, the deferred unification mechanism makes it so that gathering type information is much harder than we would like. The various optimisations do make the evaluation of tactics quite a bit faster. Perhaps a hybrid approach could be created from the best of both worlds, where stored solutions can be committed halfway, and new information would then be retrieved from the compiler. We are sure that this brings about more engineering challenges, but might also result in a sweet spot of tactic development for Agda.

Bibliography

[1] C. Paulin-Mohring, 'Introduction to the coq proof-assistant for practical software verification,' in *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–95, ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_3. [Online]. Available: https://doi.org/10.1007/978-3-642-35746-6_3.

[2] D. Delahaye, 'A Tactic Language for the System Coq,' in *Logic for Programming and Automated Reasoning*, M. Parigot and A. Voronkov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95, ISBN: 978-3-540-44404-6.

[3] J. Korkut, 'Edit-Time Tactics in Idris,' Ph.D. dissertation, Wesleyan University, 2018.

[4] L. de Moura, S. Kong, J. Avigad, F. van Doorn and J. von Raumer, 'The Lean Theorem Prover (System Description),' in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds., Cham: Springer International Publishing, 2015, pp. 378–388, ISBN: 978-3-319-21401-6.

[5] R. Milner and R. S. Bird, 'The Use of Machines to Assist in Rigorous Proof,' *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 312, no. 1522, pp. 411–422, 1984, ISSN: 00804614. [Online]. Available: http://www.jstor.org/stable/37442 (visited on 2022-8-24).

[6] Agda. 'Agda 2.6.2.2 documentation.' (2022), [Online]. Available: https://agda.readthedocs.io/en/v2.6.2.2/index.html.

[7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Edition. Prentice Hall, 1988, ch. 4.11.2, ISBN: 978-0131103627.

[8] E. Kohlbecker, D. P. Friedman, M. Felleisen and B. Duba, 'Hygienic macro expansion,' in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 151–161, ISBN: 0897912004. DOI: 10.1145/319838.319859. [Online]. Available: https://doi.org/10.1145/319838.319859.

[9] J. Cockx. 'ataca: A TACtic library for Agda.' (2021), [Online]. Available: https://github.com/jespercockx/ataca/ (visited on 2022-5-30).

[10] H. Barendregt and H. Geuvers, 'Proof-Assistants Using Dependent Type Systems.,' *Handbook of automated reasoning*, vol. 2, pp. 1149–1238, 2001.

[11] '5. The Rules of the Game,' in *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, T. Nipkow, M. Wenzel and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 67–104, ISBN: 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9_5. [Online]. Available: https://doi.org/10.1007/3-540-45949-9_5.

[12] W. A. Howard, 'The formulae-as-types notion of construction,' *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.

[13] J. Lambek and P. J. Scott, *Introduction to higher-order categorical logic*. Cambridge University Press, 1986, vol. 7.

[14] B. Stroustrup, *The C++ Programming Language*, 4th. Addison-Wesley, 2013, ISBN: 0321563840.

[15] Coq. 'Coq 8.15.2 documentation.' (), [Online]. Available: https://coq.inria.fr/refman/index.html (visited on 2022-8-16).

[16] L. Cardelli and P. Wegner, 'On understanding types, data abstraction, and polymorphism,' *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985.

[17] A. Kennedy and D. Syme, 'Design and Implementation of Generics for the .NET Common Language Runtime,' in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01, Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 1–12, ISBN: 1581134142. DOI: 10.1145/378795.378797. [Online]. Available: https://doi.org/10.1145/378795.378797.

[18] C. Parnin, C. Bird and E. Murphy-Hill, 'Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored,' in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 3–12, ISBN: 9781450305747. DOI: 10.1145/1985441.1985446. [Online]. Available: https://doi.org/10.1145/1985441.1985446.

[19] T. Altenkirch, C. McBride and J. McKinna, 'Why Dependent Types Matter,' *Manuscript, available online*, 2005.

[20] P. Wadler, W. Kokke and J. G. Siek, *Programming Language Foundations in Agda*. Aug. 2022. [Online]. Available: https://plfa.inf.ed.ac.uk/20.08/.
[21] B. C. Smith, 'Reflection and Semantics in LISP,' in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84, Salt Lake City, Utah, USA: Association for Computing Machinery, 1984, pp. 23–35, ISBN: 0897911253. DOI: 10.1145/800017.800513. [Online]. Available: https://doi.org/10.1145/800017.800513.
[22] J. Cockx. 'The Agda's New Sorts.' (May 2018), [Online]. Available: https://jesper.sikanda.be/posts/agdas-new-sorts.html (visited on 2022-7-30).
[23] G. H. Compiler. 'Glasgow Haskell Compiler / GHC.' (2022), [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/tree/97655ad88c42003bc5eeb5c026754b005229800c.
[24] Ecma International, 'ECMAScript® 2015 Language Specification,' Jun. 2015. [Online]. Available: https://262.ecma-international.org/6.0/.
[25] P. W. Authors. 'Generators - Python Wiki.' (Dec. 2020), [Online]. Available: https://wiki.python.org/moin/Generators (visited on 2022-8-22).
[26] 'yield (C# Reference).' (Sep. 2021), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/yield (visited on 2022-6-20).
[27] S. Klabnik, C. Nichols and Contributions from the Rust Community, *The Rust Programming Language*. 2022, ch. 13.2. [Online]. Available: https://doc.rust-lang.org/book/ch13-02-iterators.html (visited on 2022-1-18).
[28] M. Shaw, W. A. Wulf and R. L. London, 'Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators,' *Commun. ACM*, vol. 20, no. 8, pp. 553–564, Aug. 1977, ISSN: 0001-0782. DOI: 10.1145/359763.359782. [Online]. Available: https://doi.org/10.1145/359763.359782.
[29] R. E. Griswold, D. R. Hanson and J. T. Korb, 'Generators in Icon,' *ACM Trans. Program. Lang. Syst.*, vol. 3, no. 2, pp. 144–161, Apr. 1981, ISSN: 0164-0925. DOI: 10.1145/357133.357136. [Online]. Available: https://doi.org/10.1145/357133.357136.
[30] B. Liskov, 'A History of CLU,' in *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 471–510, ISBN: 0201895021. [Online]. Available: https://doi.org/10.1145/234286.1057826.
[31] 'Language Integrated Query (LINQ) (C#).' (Feb. 2022), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 2022-7-5).
[32] M. Snoyman. 'Iterators and Streams in Rust and Haskell.' (Jul. 2017), [Online]. Available: https://www.fpcomplete.com/blog/2017/07/iterators-streams-rust-haskell/ (visited on 2022-1-17).
[33] D. Coutts, R. Leshchinskiy and D. Stewart, 'Stream Fusion: From Lists to Streams to Nothing at All,' in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '07, Freiburg, Germany: Association for Computing Machinery, 2007, pp. 315–326, ISBN: 9781595938152. DOI: 10.1145/1291151.1291199. [Online]. Available: https://doi.org/10.1145/1291151.1291199.
[34] N. Calkin and H. S. Wilf, 'Recounting the rationals,' *The American Mathematical Monthly*, vol. 107, no. 4, pp. 360–363, 2000.
[35] S. Heubach and T. Mansour, 'Compositions of n with parts in a set,' *Congressus Numerantium*, vol. 168, p. 127, 2004.
[36] P. Wadler, 'Comprehending Monads,' in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, Nice, France: Association for Computing Machinery, 1990, pp. 61–78, ISBN: 089791368X. DOI: 10.1145/91556.91592. [Online]. Available: https://doi.org/10.1145/91556.91592.
[37] C. Omar, I. Voysey, M. Hilton, J. Aldrich and M. A. Hammer, 'Hazelnut: A Bidirectionally Typed Structure Editor Calculus,' in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17, Paris, France: Association for Computing Machinery, 2017, pp. 86–99, ISBN: 9781450346603. DOI: 10.1145/3009837.3009900. [Online]. Available: https://doi.org/10.1145/3009837.3009900.
[38] N. G. de Bruijn, 'Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,' *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, 1972, ISSN: 1385-7258. DOI: https://doi.org/10.1016/

1385-7258(72)90034-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/1385725872900340.
[39] V. Sjoberg and A. Abel, *[Agda] origin of "Agda"?* [Online]. Available: https://lists.chalmers.se/pipermail/agda/2016/008867.html (visited on 2022-8-23).
[40] C. contributors, *Alternative names · coq/coq Wiki*, Sep. 2021. [Online]. Available: https://github.com/coq/coq/wiki/Alternative-names (visited on 2022-8-23).
[41] J. Shore, 'Fail fast [software debugging],' *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004. DOI: 10.1109/MS.2004.1331296.
[42] W. Jedynak, M. Biernacka and D. Biernacki, 'An Operational Foundation for the Tactic Language of Coq,' in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '13, Madrid, Spain: Association for Computing Machinery, 2013, pp. 25–36, ISBN: 9781450321549. DOI: 10.1145/2505879.2505890. [Online]. Available: https://doi.org/10.1145/2505879.2505890.
[43] Kokke, Pepijn and Swierstra, Wouter, 'Auto in Agda,' in *Mathematics of Program Construction*, Hinze, Ralf and Voigtländer, Janis, Ed., Springer International Publishing, 2015, pp. 276–301, ISBN: 978-3-319-19797-5.
[44] Pédrot, Pierre-Marie, 'Ltac2: tactical warfare,' in *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, 2019, pp. 13–19.
[45] G. Malecha and J. Bengtson, 'Extensible and Efficient Automation Through Reflective Tactics,' in *Programming Languages and Systems*, P. Thiemann, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 532–559, ISBN: 978-3-662-49498-1. DOI: 10.1007/978-3-662-49498-1_21.
[46] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas and D. Dreyer, 'Mtac2: Typed Tactics for Backward Reasoning in Coq,' *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. DOI: 10.1145/3236773. [Online]. Available: https://doi.org/10.1145/3236773.
[47] E. Brady, 'Idris, a general-purpose dependently typed programming language: Design and implementation,' *Journal of Functional Programming*, vol. 23, no. 5, pp. 552–593, 2013. DOI: 10.1017/S095679681300018X.
[48] P. v. d. Walt and W. Swierstra, 'Engineering proof by reflection in Agda,' in *Symposium on Implementation and Application of Functional Languages*, Springer, 2012, pp. 157–173.
[49] P. van der Walt and W. Swierstra, 'Engineering proof by reflection in agda,' in *Implementation and Application of Functional Languages*, R. Hinze, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 157–173, ISBN: 978-3-642-41582-1.
[50] W. Jedynak. 'wjzz/Agda-reflection-for-semiring-solver: A simple demonstration of the Agda Reflection API.' (Apr. 2012), [Online]. Available: https://github.com/wjzz/Agda-reflection-for-semiring-solver (visited on 2022-8-23).
[51] O. Kiselyov, C.-c. Shan, D. P. Friedman and A. Sabry, 'Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl),' *SIGPLAN Not.*, vol. 40, no. 9, pp. 192–203, Sep. 2005, ISSN: 0362-1340. DOI: 10.1145/1090189.1086390. [Online]. Available: https://doi.org/10.1145/1090189.1086390.

A Glossary

- **API**: An interface between different software components, where one piece of software exposes services or routines to another piece of software.
- **Closed term**: A term that contains no free variables.
- **Desugaring**: The removal of syntactic sugar. Syntactic sugar is syntax that only exists for convenience of reading and writing code, but is not actually a language construct. Syntactic sugar is usually transformed into another lower-level representation.
- **Edit-time**: The term "edit-time" is a wordplay on the words "run-time" and "compile-time", and refers to the time when a program is still being written in an editor [3], [37].
- **Metaprogramming**: Metaprograms are programs that operate on programs. By extension, metaprogramming is the act of programming programs that operate on programs. This typically refers to the creation of programs that write or modify programs.
- **Pragma**: A comment that specifies additional information to the compiler. These comments are not ignored by the compiler, but actually carry meaning that may change the compiler's output.
- **Telescope**: Within the Agda compiler, a telescope is a list of variables that are bound in the current scope. In some telescopes, each entry has an associated string name; other telescopes only carry the types.
- **Thunk**: A value that is yet to be evaluated. In Haskell (and by extension Agda) these objects are used to implement lazy evaluation, which means that an expression is only evaluated when it is actually required.

B Ataca and Attic time comparison

This appendix contains the code and results of the performance comparison between Ataca and Attic. Listing 29 displays the code of which the type checking time was measured in the Ataca library, and listing 30 displays the code of which the type checking type was measured in the Attic library. Each measurement was run five times, to rule out random noise. Table 1 lists the averages of our measurements. The baseline compilation times are given in table 2, to give an overview of how long type checking takes when no tactic macros are used. Details of this experiment and discussion can be found in section 6.1.

```
module Ataca.Perftest where

open import Data.Nat using (ℕ)
open import Data.Unit.Polymorphic renaming (⊤ to ⊤ℓ; tt to ttℓ)
open import Ataca.Tactics.BasicTactics
open import Ataca.Tactics.Exact
open import Ataca.Tactics.Intro
open import Ataca.Core
open import Ataca.Utils

repeatTry : ℕ → Tac A → Tac ⊤ℓ
repeatTry zero tac = return _
repeatTry (suc k) tac = (tac >> repeatTry k tac) <|> return _

intros100Slow = repeat 100 (intro' <|> return _)
intros100Fast = repeatTry 100 intro'

solution' = intros100Fast >> (exact' {A = ⊤} (quoteTerm 5))

macro
  solution = runTac solution'

-- Paste generated cases below this line

introTest0 : ℕ
introTest0 = solution

introTest1 : ℕ → ℕ
introTest1 = solution

introTest2 : ℕ → ℕ → ℕ
introTest2 = solution

introTest3 : ℕ → ℕ → ℕ → ℕ
introTest3 = solution

-- etc...
```

Listing 29: The performance evaluation code in Ataca. This code includes the `repeatTry` function we added, which allows Ataca to run much faster than the default implementation.

```
module AtticTest.Perftest where

open import Data.Nat using (ℕ)
open import Attic.BasicTactics
open import Attic.Macro
open import Attic.Monad
open import Attic.Tactics.Exact
open import Attic.Tactics.Intro

intros100Slow = repeat 100 (try intro)
intros100Fast = repeatTry 100 intro

solution' = intros100Fast >> (exactSyntax (quoteTerm 5))

macro
  solution = asMacro solution'

-- Paste generated cases below this line

introTest0 : ℕ
introTest0 = solution

introTest1 : ℕ → ℕ
introTest1 = solution

introTest2 : ℕ → ℕ → ℕ
introTest2 = solution

introTest3 : ℕ → ℕ → ℕ → ℕ
introTest3 = solution

-- etc...
```

Listing 30: The performance evaluation code in Attic.

Max. parameters	Ataca slow (ms)	Ataca fast (ms)	Attic slow (ms)	Attic fast (ms)
0	629.6	200.4	349.8	57.4
5	10200.8	574.8	1954.6	194.2
8	27352.4	1493.2	3175.6	512.8
10		2244.6	4894.2	748.0
15		7087.4	7679.0	1976.4
20		15068.0	12133.2	5164.6
23		26459.4	20922.2	7302.4
25			23241.6	11094.6
26			31073.4	11854.8
30				22633.4
31				31172.6

Table 1: Averages of the type checking times. Empty cells represent the lack of a measurement value because the compiler did not finish type checking.

Baseline measurement	Ataca slow (ms)	Ataca fast (ms)	Attic slow (ms)	Attic fast (ms)
1	181	184	53	53
2	181	195	55	52
3	189	181	53	55
4	187	183	55	55
5	180	183	52	56
Average	183.6	185.2	53.6	54.2

Table 2: Baseline compilation times for both methods if no tactic macros are used.