Search space reduction for
Dafny tactics

Final year dissertation

Vytautas Tumas
Software Engineering Year 4

supervised by
Dr. Gudmund Grov

School of Mathematics and Computer Science
Heriot-Watt University
April 24, 2016
Abstract

Program verifiers such as SPARK2014, Spec# and Dafny rely on SMT solvers to prove program properties. If the solvers fail, users guide the proof with annotations. Tacny tool introduces high-level proof encodings, called tactics for the Dafny program verifier. Together with Gudmund Grov we implemented the Tacny prototype during the summer of 2015. The tool allowed users to develop reusable tactics, however, the tool was very slow and required a lot of resources to execute even the simplest of tactics. The aim of the project was to identify and rectify the performance bottlenecks in the Tacny tool, thus reducing the execution time and memory usage. We identified four bottlenecks, of which three had optimisation solutions designed and implemented. The analysis of the optimized tool showed a great reduction in the physical memory requirements and improvement in execution time.
Acknowledgements

I owe sincere and earnest gratitude to my supervisor Dr. Gudmund Grov, who challenged and encouraged me throughout the year. Without his support, help and guidance I could not have written the dissertation to its fullest. I would also like to thank my second reader, Prof. Nick Taylor, for the insightful feedback he gave me.
Declaration of Authorship

I, Vytautas Tumas confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Vytautas Tumas
Date: April 24, 2016
# Contents

## 1 Introduction

1.1 Introduction ................................................. 1
1.2 Summary .................................................. 2

## 2 Technical Background

2.1 Theorem Provers ........................................... 4
2.1.1 Interactive Theorem Provers ......................... 4
2.1.2 Automated Theorem Provers .......................... 5
2.2 C# programming language ................................. 6
2.3 Program Verification ....................................... 8
2.3.1 Design by Contract .................................... 8
2.3.2 Boogie IVL ............................................. 9
2.3.3 Dafny .................................................. 10
2.3.4 Tacny .................................................. 12
2.4 Optimisation techniques ................................. 21
2.4.1 Data parallelism ..................................... 21
2.4.2 Lazy Evaluation ...................................... 23
2.4.3 Search Strategies ................................... 23

## 3 Analysis

3.1 Test Data .................................................. 25
3.2 Performance Analysis ..................................... 30

## 4 Design

4.1 Type lookup ............................................... 37
4.2 Lazy evaluation .......................................... 38
4.3 Search framework ........................................... 41
4.4 Parallel execution ........................................... 43

5 Implementation ........................................... 44
  5.1 Type lookup ............................................... 44
  5.2 Lazy evaluation ........................................... 46
  5.3 Search Framework ......................................... 49

6 Evaluation .................................................. 53
  6.1 Base performance .......................................... 53
    6.1.1 Execution time analysis ................................ 53
    6.1.2 Search Space and Memory analysis ......................... 54
  6.2 Optimisation Analysis ...................................... 56
    6.2.1 Type Lookup ........................................... 56
    6.2.2 Lazy Evaluation ........................................ 58
    6.2.3 Search Strategy Comparison ................................. 61

7 Conclusion and Future Work .................................. 65
Chapter 1

Introduction

1.1 Introduction

Software correctness can be decided with respect to an independent specification of what it should compute. Dynamic software analysis is the most common approach used in industry. It involves executing the system with a set of different input parameters in the hope to find a state under which the program will break. Once a bug is identified it gets fixed and the process is repeated until all test cases pass. However, because the majority of the software worth testing is extremely complicated with millions of possible states, it is not possible to test every state of the program.

Static analysis does not involve the execution of the software; instead the analyser is working directly with the representation. This may be the code, a model or a system design. Static analysers can be applied at early stages of the development process. They range from simple heuristic code checkers like Lint [15] - a static code analyser for C and C++, to imperative programming languages with inbuilt verifiers, such as SPARK2014 [2] or Dafny [16]. These languages use contracts to express program properties: given a precondition the system guarantees the postcondition is satisfied. These contracts are converted to verification conditions (VC), which are mathematical conjectures. The VCs are sent to a theorem prover - a computer program designed to prove these conjectures.

Upon verification failure[^1], the user must provide guidance to the underlying theorem prover over the proof. The guidance is carried out by combination of altering the ghost state; a program state which is used for the verification process and does not affect the software functionality. 

[^1]: Assuming the code is correct
and providing auxiliary notations such as assertions and loop invariants to show that a loop preserves some asserted property and variants to show loop or recursion termination. The proof guidance may be a repetitive process, which involves a great number of trial-and-error to find the correct proof. Furthermore, there is a lack of reuse in the guidance process, it is not possible to encode common proof patterns and allow the computer to search for the correct proof.

The Tacny tool attempts to introduce high-level proof encodings for the Dafny program verifier. It introduces tactics, which allow users to encode abstract verification patterns. The inspiration for Tacny was drawn from tactics which allow users to reuse code patterns for Interactive Theorem Provers. Together with Gudmund Grov, I worked on Tacny by implementing a framework and integrating it with Dafny, as an intern during the summer of 2015. At the end of the summer project, we have developed the tool prototype. With the tool we could encode proof patterns called tactics, and using these generate concrete proofs. However, it became apparent that the tool is very slow. The aim of the dissertation is to optimise the Tacny tool.

1.2 Summary

The aim of the project was to identify and develop solutions for performance bottlenecks inside the Tacny tool. In chapter 3 we have identified four bottlenecks:

The main issue was the search space generated by the tool, which in turn causes long execution time and high memory usage. The majority of generated search space contained invalid Dafny programs - programs that are not type correct. The type checking failure is caused by Tacny atom statement resolvers. During Dafny code generation, the resolvers do not carry out any type checking, therefore, majority of the generated solutions are invalid. To prevent the solutions from being generated a type lookup table was implemented. Any atom statement which generates Dafny code can refer to this table to type check any variables being used to ensure only correct type parameters are being passed. The results show a high reduction in the execution time and memory consumption. The design of this and other optimisations is discussed in chapter 4.

Currently tactic evaluation is carried out eagerly, that is, Tacny first generates the entire search space and only then searches for the correct solution. As the result, the entire search space is kept in memory, this negatively impacts the memory usage. Furthermore, not all nodes need to be generated to reach the final solution. To counter these issues the lazy evaluation
strategy was implemented. It generates the solutions up on request, one at a time, as the result we can generate and verify the nodes one at a time, thus, to find the final solution only the minimum search space is explored. With lazy evaluation the largest of test samples used just a fraction of the memory, the original tool required to execute the sample. To implement the optimisation, the majority of the tool was rewritten, the implementation details are discussed in chapter 5.

Currently Tacny uses a breadth-first-search (BFS) strategy to find the correct solution. The strategy prioritizes expanding the width of the tree before exploring the deeper nodes. As the result, in cases, when the final solution is at the bottom of the tree, the algorithm will require to traverse the entire tree to reach it. To counter this, we have developed a framework that allows users to specify the search strategy for each tactic. In addition to the existing BFS algorithm, the DFS strategy was developed, it prioritizes exploring the depth of the tree, before branching out to the sides and thus, in some cases it offers a better performance than BFS strategy.

Finally, tactics are resolved sequentially, that is, the Dafny program is scanned top to bottom resolving each found tactic application one at a time. To speed up this process we propose to modify the Tacny tool to resolve tactics in parallel, thus reducing the overall time required to resolve the whole program. However due to some technical issues this optimisation was not implemented, we cover these in detail in section 4.4.

The performance evaluation described in chapter 6 showed that the optimisations drastically improved the performance of the Tacny tool, in some cases the search space was reduced from over 33 million nodes, to just over few hundred. In turn, this greatly improved the execution time, and reduced the memory usage. All of the optimisations were implemented with an on/off switch, thus we were able to analyse how each of them affected the tool.
2.1 Theorem Provers

A proof in mathematics plays two roles, it convinces the reader that a statement is correct and explains why it is correct. A theorem prover is a piece of software which is designed to prove mathematical problems. Software designed to search for conjecture proofs has been around for nearly half a century [9]. These days, in addition to proving mathematical correctness, they are making their way into software and hardware verification. Verification includes model checking and formal program validation and verification (V&V). Model checking is an algorithmic way to check whether a finite-state system satisfies a given specification. The goal of V&V is to ensure that the representation of the program satisfies the behaviour it is specified to have. In this paper we will concentrate on program verification.

2.1.1 Interactive Theorem Provers

*Interactive Theorem Provers (ITP)*, such as Isabelle [26] and PVS [27], also known as *Proof Assistants*, are computer systems which allow the users to define and prove mathematical theories. The user can define a mathematical theory, define their properties and carry out logical reasoning with the theories. In addition, these systems allow definitions of functions and computation with them. The theorems are then proved by writing a proof script, which will guide the theorem prover through the proof. In these scripts we can distinguish two main languages: a proof language, which is a set of proof primitives and a tactic language, which enables the user to write their own automatic proof schemes. A *tactic* is function, which given a goal, will break it down to 0 or more sub-goals. For instance, to prove the predicate *(goal)* $A \land B$, we
have to break it down to two separate predicates \( A \) and \( B \) and prove them individually. A tactic, when applied to a goal, reduces the goal to its sub-goals [12].

The length of a proof highly depends on the automation level of the proof assistant. If an automatic proof scheme does not exist for the particular problem, the users can write the proof by hand using the proof language or implement the automatic proof scheme (tactic) using the tactic language. However, the tactic language is quite different from the proof language, which means that the users have to spend extra time to learn this language. Recently, there have been more user-friendly tactic languages developed, one of such languages is \( Ltac \) [11]. The old versions of the Coq [5] theorem prover used \( Objective\ Ca ml \) as the tactic language and the implemented tactics were used to automate the specific problem. Thus there was a need for a higher-level, intermediate language, integrated in the prover, to write tactics for automating small parts of the proofs. \( Ltac \) is the new tactic language available in Coq. \( Ltac \) is a functional meta-level programming language, which can be used directly in the proof scripts or as a reusable top-level definition (tactic declaration). The top-level tactic declarations are constructed with atomic tactic statements and tactic expression, which are used to combine the atomic statements.

### 2.1.2 Automated Theorem Provers

At the other end of the spectrum there are Automated Theorem Provers (ATP). ATP system has a number of well chosen decision strategies which allow specific formulas to be proven automatically. The first significant ATP is the Logic Theory Machine developed by A. Newell [25]. The Machine was based on propositional logic, the proofs were constructed from 3 deduction rules and a set of propositional axioms. It also introduced a method of proving problems backwards starting from the problem goal working the way up [25]. These systems, although powerful, have limited expression power, as the result it is impossible to construct a generic mathematical theory for ATPs. Deciding formula validity depends on the underlying logic, propositional logic problems are decidable but co-NP complete thus the problems can be solved with exponential-time algorithms. Some of the well known Automated Theorem Provers include Vampire [31] and E [32], which are first-order logic theorem provers, and MetiTarski, an ATP designed to prove theorems involving real value function [1].
SMT Solvers

A SAT or Boolean Satisfiability Problem, is a decision problem for determining whether there exists an interpretation of a boolean formula, which satisfies the formula. That is, if there exist a combination of TRUE/FALSE values for the formula variables, such that the formula is evaluated to TRUE. The Cook’s Theorem \cite{Cook} proves that SAT problems are NP-Complete. This means that, using a non-deterministic Turing Machine, it is possible to solve the SAT problem in polynomial time. However, in practical situations such as circuit design or ATP, the problem can be solved at a level of efficiency using SAT solvers that employ heuristic search.

Satisfiability Modulo Theories (SMT) is a generalization of SAT problem, with the addition of arithmetic, quantification and other first-order theories. An SMT solver is a computer system, which decides the satisfiability (or validity) of formulas in the theories. SMT solvers have many applications that range from test case generation \cite{Bloem}, compiler verification \cite{Dillig} to model testing \cite{Wang}.

Z3 \cite{Z3} is an SMT Solver developed by Microsoft Research. It is specialized in solving problems arising in software analysis and verification. It supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifier theories. Because Z3 is a low level tool, it is mainly used as a component in other tools that require logic problem solving. Therefore, it offers a number of APIs for different languages to map problems into Z3. One of such tools, which uses Z3 as the underlying SMT solver is Dafny. We return to the details of how Dafny uses Z3 in section \ref{sec:2.3.3}.

2.2 C# programming language

Dafny and Tacny tools are implemented in the C# programming language, in this section we will discuss some of its’ details. C# is a object-oriented (OO) multi-purpose programming language. It supports basic object-oriented principles such as encapsulation, inheritance and polymorphism.

All variables and methods are encapsulated in class definitions. A class can inherit a single parent class, and multiple interfaces. The virtual keyword is used to specify a method, which can be overridden in a derived class. A method that overrides a virtual method requires the override keyword, to ensure a method is not accidentally redefined. Furthermore, it is possible to completely hide members of the inherited class. The new keyword, when used as a declaration
modifier, hides the base class version of the declaration, and replaces it with the derived version. Members are hidden by declaring them in the derived class with the same name as the base class and adding the `new` modifier, for example:

```csharp
class A {
    public void Method() { }
    protected void OtherMethod() { }
}
class A : B {
    new public void Method() { }
}
```

Code 2.1: Hiding base class members

In the code sample 2.1, B.Method hides A.Method, however, A.OtherMethod is left unaffected. In addition to standard OO principles, C# supports a number of innovative constructs [23]. Developers can use the Language-Integrated Query (LINQ) to use standard queries over various data sources. An example of such query can be seen in code 2.2.

```csharp
public void Method(List<string> stringList) {
    string x = stringList.FirstOrDefault(x => x.ToLower() == "hello world");
    if (x != null)
        Console.Out.WriteLine(x);
}
```

Code 2.2: LINQ query

The method `FirstOrDefault` returns the first occurrence of `x`, such that lower cased `x` is equal to "hello world". If such element does not exist in the list a null value is returned instead. C# also supports properties for private member variables. A property provides a mechanism to read, write, or compute the value of a private variable, an example of such property is listed in code sample 2.3.1.

```csharp
class MyClass {
    private int _i;
    public int i {
        get { return _i; }
        set { _i = value > 0 ? value : 1; }
    }
}
```

Code 2.3: Property in C#

We declare a private field `_i` and a public field `i`, which defines the accessor properties `get` and `set`. The set property uses the `value` keyword, which defines the value being assigned,
to determine if the input is greater than 0, if so the given value is written to \texttt{i}, otherwise the default value 1 is written.

\section*{2.3 Program Verification}

Program verification is a method of analysing a piece of program text to check whether it satisfies a given specification. The need to reason about programs was evident to the early pioneers of computing \cite{21}. In his paper, \cite{13} A. Hoare specified Floyd-Hoare logic to reason about the correctness of a computer program. The main feature of the logic is the Hoare triple, which is used to specify partial correctness of a program. A program is partially correct when upon termination it will return the correct result, however it does not imply termination. The triple takes a form of \{P\}C\{Q\}, P and Q are pre and postconditions written in predicate logic, and C is an arbitrary command. In this section we will discuss modern verification techniques.

\subsection*{2.3.1 Design by Contract}

\textit{Design by Contract (DbC)} is a method of writing correct software. It uses pre and post conditions to assert the changes in a program state. The term was originally coined by B. Meyer and implemented as simple assertions in the Eiffel Language \cite{20}. The basic idea behind DbC in object oriented program is as follows:

- Each method in an arbitrary class gets assigned a set of pre and post conditions.

- Whenever a method is called, the caller guarantees that all the arguments passed to the method satisfy the specified preconditions. At the same time, at the point of return, the method guarantees to satisfy the specified postconditions.

- In addition to pre and post conditions, assertions and invariants may be included in the method body to assist in honouring the contracts.

Both Tacny and Dafny implementations use code contracts internally to ensure correctness. The code contracts in C\# are provided by the \texttt{System.Diagnostics.Contracts} library. It supports runtime checking - ensure that contracts are satisfied during the program execution, and static checking - analyse the code for contract violations without executing the program.
Precondition  Method preconditions are expressed with `Contract.Requires`. They specify the state when method is invoked and generally are used to check valid parameter values. Any members used in the precondition, must be accessible at the method level. The following precondition specifies that input $y$ is greater than 0:

```csharp
public void MyMethod (int y)
{
    Contract.Requires(y > 0);
    ...
}
```

Postcondition  Postconditions in C# contracts are expressed with `Contract.Ensures`. It expresses a condition which must be true upon the termination of the method. The postcondition specifies that some output $x$ is not null:

```csharp
public int MyMethod(int y)
{
    Contract.Ensures(this.X == y);
    this.X = y;
}
```

Object Invariants  Object invariants are used to specify a correct instance of a class upon its initialization and after each call to any of the class members. Invariant methods are marked with the `ContractInvariantMethod` attribute. During run-time checking, invariants are checked at the end of each public method, these checks are also carried out whenever the class is re-entered. Invariants are expressed as follows:

```csharp
[ContractInvariantMethod]
protected void ClassInvariant()
{
    Contract.Invariant(this.is_valid = true);
    Contract.Invariant(this.y >= 0);
}
```

2.3.2 Boogie IVL

An Intermediate Verification Language (IVL) serves as a method of encoding computer programs into a common language, while maintaining the program logic and state information. It carries out a similar function as a Intermediate Representation (IR) language for compilers. IR is a way to represent a program between a source code and a target language. The main
advantage of using IR is that it reduces the semantic gap between the source language and the
target language, in addition it reduces the number of compilers that have to be written. Assume
there are \( n \) source languages and \( m \) target machines, without an IR, \( n \times m \) compilers would have
to be built. However, with IR only \( n \) translators to IR have to be written, and then the IR is
responsible for compiling the program for the target machine. A similar concept is applied
for IVLs. The verification language is positioned between the program source language and the
language of the theorem prover. IVLs allow users to implement simple converters from source
language into common IVL syntax, which is then converted to a set of VCs for a theorem
prover.

Assume two programming languages: C# and Java, even though they share a number
of semantic and syntactic similarities they are two distinct languages. To translate either of
them to a set of VCs for a theorem prover, the programmer will have to implement a unique
translation algorithm which is a time consuming and error prone task. However, by using an
IVL, the developer will have to only implement a simple translation from either of the languages
to the IVL, which will then get converted to a set of VCs by a more sophisticated software. A
strong IVL will allow different syntaxes to be effortlessly encoded while preserving the semantics
of the source language.

Boogie \cite{18,3} is an IVL developed by Microsoft Research, designed to encode VCs for
imperative object-oriented programs. Boogie is designed to be a layer on which software verifiers
will be built of other languages. A number of verifiers have been built using Boogie: VCC \cite{7},
Dafny and Spec# \cite{4}. Boogie is also a tool, which accepts the boogie IVL as input and generates
a set of VCs which are passed to the Z3 solver.

2.3.3 Dafny

The project is targeted at the Tacny tool, which introduced tactics for the Dafny programming
language and program verifier. Dafny \cite{16} was developed by K.R.M Leino at Microsoft Research
and is targeted at the .NET platform. The programming language is imperative, object-oriented
and is designed with program verification in mind. It comes with built-in specification constructs
to assert facts about the state of the program. The constructs include preconditions: (\textbf{requires})
to specify the input requirements, postconditions: (\textbf{ensures}) to specify the requirements which
have to be satisfied upon method termination, and termination metrics: (\textbf{decreases}). To
write richer specifications users can manipulate the \textit{ghost-state}. The ghost constructs and
specifications are used only by the Dafny verifier and thus are ignored by the compiler. To prove functional correctness of a Dafny program, the program text is translated to Boogie language and is sent to the Boogie tool. The Boogie tool generates a set of VCs which are passed to the Z3 theorem prover, where the VCs are proved. Because the translation to Boogie IVL is transitive, if the verification fails, the failure can be translated back to the Dafny program code. Upon successful verification, the Dafny program is compiled to C# code.

If verification fails, the user must guide the theorem prover. This is accomplished by asserting program properties in the code. Statement correctness can be expressed through the assert statement, loops can be verified using the invariant assertion and recursion and loop termination is proved with the decreases clause. A ghost state is made up from ghost methods and ghost variables any variable declared in the ghost method is treated as a ghost, these state are used only to guide the verified and cannot modify the actual Dafny program. A type of ghost method is a lemma. These can be used to specify very rich and complex program properties. Lemma body is a proof which satisfies the ensures clause given the requires clause.

To show case Dafny, consider a lemma\(^1\) which states that given any prime number there always exists a greater prime number. The input variable \(k\) is a number, for which the lemma will prove that there exists a greater prime. The ensures contract specifies what the lemma will prove. \(\text{IsPrime}(p)\) is a predicate which is true iff \(p\) is a prime number. The loop invariant \(\text{AllPrimes}(s, j)\) ensures that every element in \(s\) is a prime number and that it contains every prime up to \(j\). The loop termination is specified with \(k - j\), which says that the loop will terminate when \(j\) is greater or equal to \(k\). \(\text{GetLargerPrime}(s, j)\) will return the next greatest prime, which is not yet in \(s\). If the next prime is greater than \(k\) the lemma terminates, otherwise, every prime number up to \(p\) is added to the the list \(s\) and \(j\) is set to the largest prime in \(s\).

\[\begin{align*}
\text{lemma AlwaysMorePrimes}(k: \text{int}) \\
\text{ensures } & \exists p \bullet k \leq p \land \text{IsPrime}(p); \\
& \{ \\
& \text{var } j, s := 0, \{\}; \\
& \text{while true} \\
& \text{invariant } \text{AllPrimes}(s, j); \\
& \text{decreases } k - j; \\
& \{ \\
& \text{var } p := \text{GetLargerPrime}(s, j); \\
& \text{if } k \leq p \{ \text{return} \}; \\
& \text{set } x \mid 2 \leq x \leq p \land \text{IsPrime}(x); \\
& \}
\end{align*}\]

\(^1\)The example is taken from Primes.dfy, available on the Dafny webpage [16].
2.3.4 Tacny

Metaprogramming

Before going into details about the Tacny tool, I will first introduce the concept of *metaprogramming*. A program at its’ core can be defined with the following expression \[34\]:

\[
\text{Program} = \text{data structure} + \text{algorithm}
\]

A program executes a set of data manipulations on the input data producing output data. The manipulation instructions are described in the algorithm. We can then describe programming as a process of writing programs, that automatically solve arbitrary computational problems, producing output data as the solution to the problem.

Meta-programming extends defacto programming by generalizing the input data and the manipulating algorithm. Data and programs greatly differ in their semantics, a program is a solution to a computational problem, data is a set of symbols on which computation is performed. However, programs and data are similar in a sense that they both are a set of syntactic symbols that follow a specified grammar. Furthermore, programs produce data, that is given input data, a program will generate output data, therefore we can say that a program is a generalization of data. A generalized algorithm is a set of operations, that take programs as input, but not concrete data. We can describe a meta-program as a higher-level program, which takes a high-level program as input, performs a set of program manipulations on it, and produces a lower-level program as an output. From this, we can define meta-programming as a process of higher-level programming, for describing how to perform program manipulations \[33\].

Many modern verifiers depend on the user annotating the program with guidance to verifier. This process may be repetitive, tedious and low-level, in addition this impacts code readability, development costs and increases the overall production time. Tacny is a tool, that takes a Dafny program, which contains proof abstractions called *tactics*, and outputs a Dafny program, with concrete proofs. By extending the original Dafny syntax with a set of non-intrusive syntax adjustments, we can encode proof patters as Dafny tactics. As the result, manual search and
verifier guidance can be replaced with calls to such tactics whilst keeping in the familiar Dafny setting.

The Tacny tool (Figure 2.1) takes a Dafny program extended with tactics (.tacny). The program is parsed in the **Parser** and passed to the **Interpreter** where the tactics are resolved. The **Generator** creates a valid Dafny program (.dfy) by removing all tactics and replacing the tactic calls with the generated Dafny code from the interpreter. The result from the Generator is then sent to Dafny **Resolver** where type checking is carried out and the program is prepared for translation to Boogie. Dafny Verifier takes a resolved Dafny program as an input and sends it to Boogie where the verification is carried out, the result is sent back to the Interpreter via the Dafny Verifier.

**Tacny Syntax**

Tacny introduces two new types:

- **Term** used to represent expressions of the Dafny mathematical language. Syntactically, this type is the same as an expression, however the actual terms are used and manipulated;
- **Element** is the ‘name’ of an entity of a language, e.g. a construct (if, while), method, lemma, constructor, variable etc.

Tacny also introduces a new ghost method called **tactic** which has the following syntax. The syntax has been taken from the Tacas2016 [13] paper. A tactic is a special ghost method with the following syntax:

\[
\text{tactic \ Id(Params)\{} \ TStmt \ \}\n\]

Where the **Id** is the name of the tactic, **Params** is a set of parameters, where a single parameter is defined as **Id : Type**. **Id** is the name of the parameter, the type refers to a Dafny type extended with **Element** and **Term**. **TStmt** is an extension of a subset of the Dafny statement **Stmt** [19] it is defined as follows:

---

2 Note that these are not valid Dafny types
\[ TStmt := \text{Atom} \]
\[ \mid TStmt \ || \ TStmt; \]
\[ \mid \text{tvar} \ Id := TExpr; \]
\[ \mid Id := TExpr; \]
\[ \mid \text{tvar} \ Id :: TExpr; \]
\[ \mid Id :: TExpr; \]
\[ \mid Id(TExprs); \]
\[ \mid \text{if} \ TExpr \{ \ TStmts \} \]
\[ \mid \text{if} \ TExpr \{ \ TStmts \} \text{else} \{ \ TStmts \} \]
\[ \mid \text{while} \ TExpr \text{Inv} \{ \ TStmts \} \]

The \textit{if} and \textit{while} statements should not be confused with Dafny if and while statements. These statements, and their bodies are resolved during tactic resolution. The \textit{Atoms} are the atomic steps and can be seen as a small functionally correct kernel of the system:

\[ Atom := \text{cases}(\text{Element}) \{ \ TStmts \} \]
\[ \mid \text{perm}(\text{Element}, \text{seq}<\text{Element}>); \]
\[ \mid \text{id}(); \]
\[ \mid \text{fail}(); \]
\[ \mid \text{add}_\text{invariant}(TExpr); \]
\[ \mid \text{add}_\text{variant}(TExpr); \]
\[ \mid \text{changed} \{ \ TStmts \} \]
\[ \mid \text{try} \{ \ TStmts \} \text{catch} \{ \ TStmts \} \]
\[ \mid \ldots \]

Tacny supports simple mathematical expressions, that is, it can evaluate variable comparison, and basic arithmetic functions, in addition there is a number of Tacny expressions:

\[ TExpr := \text{Expr} \]
\[ \mid \text{variables}() \]
\[ \mid \text{lemmas}() \]
\[ \mid \text{params}() \]
\[ \mid \ldots \]
To illustrate a Dafny program extended with tactics, we will develop a tactic for the
\texttt{AsimpConst} lemma\footnote{Lemma has been taken from the Nipkow-Klein-chapter3.dfy}. The original lemma can be seen in the code snippet \ref{asimpconst}. \texttt{aexp} is a
datatype which represents a simple arithmetic expression. The expression can be either an
integer, a variable name, or the addition of two arithmetic expressions. State \texttt{s} is a map from
\texttt{vname} to an \texttt{int}. \texttt{Total(s)} states that the state must be total, i.e. the \texttt{vname}
must have a value assigned to it. \texttt{aval(a, s)} is the evaluation of the expression \texttt{a} in the state \texttt{s}. \texttt{asimp\_const(a)} performs constant folding of the arithmetic expression. Any sum of constants in the arithmetic
expression \texttt{a} is replaced by the value of their sum. The \textit{lemma} \texttt{AsimpConst} proves that constant
folding preserves the behaviour of the state.

![Datatype aexp and lemma AsimpConst](image)

To prove this lemma using pattern matching a case is introduced for each constructor, and
for the recursive \texttt{Plus(a0, a1)} constructor the lemma is applied for each argument recursively.

Consider \texttt{lemma BsimpCorrect}, which proves a similar property to \texttt{AsimpConst}, but for
applied binary expressions. We can observe that the two lemmas follow a similar proof pattern,
both lemmas use pattern matching on the inductive data type and the bodies of the case
statements contain up to two calls to a lemma. Therefore we can develop a single tactic to
generate the proofs for both lemmas.

![Datatype bexp and lemma BsimpCorrect](image)

\footnote{Lemma has been taken from the Nipkow-Klein-chapter3.dfy}
The tactic in mind is `CasePerm`, seen in the code example 2.7. The input for the tactic is expected to be an inductive data type. The tactic will generate a match statement with a case statement for each constructor of `b`. For each case statement, the body of the `cases` will be evaluated. Inside the body we declare a `tactic variable` `v`, which holds a list of all Dafny variables declared in the current scope, merged with the list of method parameters. For each lemma in the program, we declare a loop counter `i` and execute the loop. For each iteration of the while loop, we generate a call to the lemma `l` using all possible combinations of variables from the list `v`, and increment the loop counter. For each result of the loop iteration, we continue executing the loop until the guard expression fails. As the result, we can generate a `case` body which can contain up to two calls to the lemma 2.6.

```
tactic CasePerm (b : Element) {
  cases b {
    tvar v := merge(variables(), params());
    tvar l : l in lemmas();
    tvar i := 0;
    while (i < 2) {
      perm(l, v);
      i := i + 1;
    }
  }
}
```

Code 2.7: Tacny tactic [13]

We replace the lemma body with the call to the tactic and let Tacny generate the proof. After the proof has been generated the call to the tactic is replaced with the valid proof. Using Tacny to generate Dafny proofs has two advantages. First, because Tacny verifies each generated program and terminates if the verification is successful the resulting proof will be the shortest proof required for the particular problem. Secondly, as showed in the previous example, we can reuse a single tactic to prove multiple lemmas.

**Tacny implementation**

The optimisations may require an extensive overhaul of the Tacny tool, thus I will cover the implementation details in this section. Tacny is implemented as a stand alone tool, however it
Figure 2.2: Tacny System Architecture
extends the original Dafny Syntax tree, with a new member declaration for tactic and new statement, for the cases Atom. Like Dafny, Tacny is implemented in C#.

The system architecture is depicted in figure ?? . Inside the TacnyDriver class, Dafny and Boogie are installed and Tacny Interpreter class initialized and calls the ResolveProgram method, which finds and resolves any Tactic calls. The method iterates over the members of the Dafny Program and for each member calls ScanMemberBody method. In the ScanMemberBody we prepare the global and local contexts and iterate over the body of the member. If a statement is a call to the tactic we call the ResolveTactic method, which resides inside the Atomic class.

The tactic resolution work is carried out inside the Atomic class. Each atomic statement and expression has an associated resolver class which implements the IAtomicStmt interface. The interface defines the Resolve method. In this method we define the logic required to resolve the atomic statement. Inside the ResolveTactic method, for each statement inside the tactic body, we call the StatementRegister class, where we find the associated resolver class. The class is then instantiated and the statement is resolved.

When all the statements have been fully resolved, the resulting list of solutions is returned back to the ScanMemberBody method. After the member body is fully scanned, the solution list is searched for the valid proof. In the further paragraphs I will go into more detail about each of the components that make up the Tacny system.

Tacny Program The Tacny Program is the extension of the Dafny Program class. It holds a reference to the program AST, a list of verification errors and a cache of the program members. In addition, the class has method definitions for calling Dafny program resolver and Boogie verification.

StatementRegister Most of the tacny atomic statements and expressions do not have a class representation inside the AST. Inside the AST they are represented as simple member calls. Therefore, to differentiate between these calls and easily find the associated resolver class we created the StatementRegister class, which holds the mapping call name → IAtomicStmt. Inside the StatementRegister class, GetStatementType, given an UpdateStmt will return an instance of System.Type of the associated resolver.

Context The Tacny context is a record, which contains the contextual information of the interpreter state. Tacny uses two types of contexts: global and local. The data held in Global
Context is readonly and can be accessed at any stage of the tacny program resolution.

Global Context holds:

- *datatype* definitions
- Global variable declarations
- Different counters for statistical purposes

Local Context holds the contextual information for tactic interpretation. A new instance of the Local Context is initialized for each tactic call.

The following data is held in Local Context:

- The instance of tactic currently being resolved
- Local variable declarations
- Any intermediate results
- Tactic body counter

**Solution List** During tactic resolution, executing the *atomic* statements may result in multiple versions of generated Dafny code. There is no way to determine which (if any) of the versions are correct, thus they are captured as *solutions* and, later, generated to Dafny programs which are then resolved and verified.

The *solution* class captures the following information:

- *state* The state of the tactic resolution. This may be final or intermediate result.
- *isFinal* A flag which determines whether the state is final (the tactic is fully resolved) or intermediate.

The solutions are held in the *SolutionList* class:

- An list of intermediate solutions, for the tactic currently being resolved.
- A list of final solutions for the program.

**Atomic** The *Atomic* class contains the reference to the *global* and *local* contexts and all the required methods for tactic resolution such as Tacny variable registration, statement body resolution etc. Tactic body resolution is carried out inside the *ResolveTactic* method, tactic resolution is covered in detail in the paragraph. For each statement inside the tactic body the *CallAction* method is called. Here, we call the *StatementRegister* class to determine the type of the statement. If the type is known, we use C# reflection to create the instance of the type and call the *Resolve* method, to resolve the statement. Reflection provides the necessary classes to dynamically an instance of a *System.Type*. Once the instance is created,
we can invoke any methods or access its’ fields and properties. If the type of the statement is unknown, we first check if the statement is a variable declaration, if it is, we register the variable and move on to the next statement. If the statement is not variable declaration, it will be inserted directly into the resulting solution.

**Tactic resolution** Tactic resolution works as follows:

- A solution with an unresolved tactic is initialized and added to the intermediate result list in the *SolutionList* class.
- While the *SolutionList* is not final (there are solutions in the intermediate list), for every solution *X*, in the *state*, the statement at the current tactic body counter is resolved.
- For each new solution *X’*, the tactic body counter is incremented.
- Lastly, *X* is replaced with the list of *X’*

Once the tactic is fully resolved, the solutions are moved to the *final* list. Figure 2.3 depicts how solution list evolves during tactic resolution. Solution List is initialised with unresolved solution *a*, the interpretation results in two new solutions *b*, *c*. The resolver takes first solution in the list *b* and resolves it to *d’*, *e* and *d’* is *final*. Because the list is iterated using breadth first, next the interpreter resolves *c* to *f*. If the list is not final, the interpreter would skip *d’* because it is final and resolve *e*. This would carry on until each solution in the list is final.

**perm() Atom** The *perm()* atom is responsible for the the highest branching factor, that is why it will be the main target for optimisation, thus we will cover in detail the inner workings of this atomic. *perm(m, v)*; generates all possible applications of method *m* with the arguments found in any subset of *v*, it was inspired by R.Leino - the main developer of the Dafny verifier. *perm()* is responsible for branch generation hence I will go into detail of it’s inner workings, later in the paper I will discuss possible optimisations for it.

Method applications are generated in two steps: first for each consecutive *perm(m, v)*; a call to *m* is generated, secondly the results are permuted again and for each combination a solution is
generated.

The first step works as follows:

1. From the \( m \) signature extract the number of parameters (\( \#P \)) it takes.

2. Generate all possible input combinations from \( v \) of size \( \#P \).

3. For each input combination, create a call to \( m \).

4. Get the next Statement in body, if it’s a \( \text{perm}() \) statement, recursively repeat step 1, otherwise terminate.

Assume there are 3 consecutive \( \text{perm}(m, v); \) calls, where each \( m \) is bound to a different method, the solutions will be generated as follows. First, Tacny will generate a solution for each application generated by the first \( \text{perm}() \) call. Secondly, it will generate all combinations of the first two applications and create a solution for each of these. Finally, combinations of the three calls will be generated and a solution for each result will be created. When generating the combinations (Step 2), Tacny does not validate whether the generated power set is type correct, i.e. Tacny is unaware of the variable types in \( v \), thus some of the generated branches contain invalid Dafny programs. In addition, because the combinations are generated eagerly and stored in memory, the atomic has a great impact on the space complexity of the program.

2.4 Optimisation techniques

The aim of the project is to optimize the Tacny tool. In this section we will overview the techniques, which we expect, will improve the performance of the tool.

2.4.1 Data parallelism

In Data parallel execution, the same operation is performed concurrently on a set of data. In these operations, the data source is partitioned in such way that different independent threads can operate on each partition concurrently. Because, each thread is working on separate partition, a race condition is not created and thus the data source does not require any locks.

A good parallel implementation will use one thread per CPU core. As the result, the processor power will be fully utilized. When thread count exceeds core count, the operating system will have to perform a context switch between the threads resulting in wastefully spent overhead. Context switching is costly for a number of reasons, the need to save the threads state
before the operating system switches the thread out and replaces with a new one. In addition, context switching has a negative impact on the cache. When a new thread is started, it will often require data to run, however because the system cache contains data for the previous thread, an expensive operation to fetch the data from the main memory will have to be carried out.

In C# data parallel execution can be implemented with the `Parallel.For` or `Parallel.ForEach` methods. The `For` method takes three parameters as an input: inclusive lower-bound, an exclusive upper-bound and a delegate which will be invoked for every iteration. For example, consider the code example 2.8:

```csharp
class MyClass
{
    public void MyMethod(List<Data> dataList)
    {
        Parallel.For(0, dataList.Count, index =>
        {
            ProcessItem(dataList[index]);
        });
    }
}
```

Code 2.8: Parallel For loop

The `dataList` is partitioned and is executed on number of threads. The thread count depends on the system environment the code is running in. For every iteration of the loop, the delegate will be executed with the `index` as the input. The loop returns a `ParallelLoopResult` instance which contains the details of the executed loop. The `For` loop is designed to iterate over a specific kind of dataset. A more generic iterator can be implemented with the `ForEach` loop. The method works the same way as `Parallel.For`, but instead of using lower and upper bounds, it directly iterates over the data structure. The code sample rewritten with the `Parallel.ForEach` can be seen in listing 2.9:

```csharp
class MyClass
{
    public void MyMethod(List<Data> dataList)
    {
        Parallel.ForEach(dataList, (item) =>
        {
            ProcessItem(item);
        });
    }
}
```

Code 2.9: Parallel ForEach loop
2.4.2 Lazy Evaluation

There is a number of expression evaluation strategies. In a *call-by-value* evaluation strategy function arguments are evaluated before execution, thus arguments are evaluated only once. When employing the *Call-by-name* (eager evaluation) strategy, an argument is evaluated any time it’s value is used inside the function body, as the result arguments are evaluated zero or more times. However, in practice, eager evaluation uses memorization to store the results of the evaluation, thus it is executed once. In *call-by-need*, (lazy evaluation), the argument is evaluated only when it’s value needed and the value is stored for further use, thus arguments are evaluated at most once. Assuming there are no side-effects lazy evaluation is the most powerful of all three as it does not require argument revaluation. To illustrate *call-by-value*, consider the code snippet 2.10. When method is called, both *x* and *y* argument will be evaluated even though *y* is never used in the method. If for example, *y* requires expensive computation the time spent evaluating it will be wasted.

```csharp
static int fun (int x, int y) {
    if (x > 0)
        return x - 1;
    return x;
}
```

Code 2.10: Call by value evaluation in C#

```hs
fun :: Int -> Int -> Int
f x y = case x > 0 of
    True -> x - 1
    False -> x
main = print $ fun 2 (product [1..])
```

Code 2.11: Lazy evaluation in Haskell

The same method implemented in Haskell - a well known programming language, which uses lazy evaluation, can be seen in code snippet 2.11. Running this snippet with 2 and (product[1..]) as arguments well output 1. We can see, even though *y* is a product of an infinite list, because the argument is never used, the compiler will not try to evaluate it.

2.4.3 Search Strategies

Tacny uses *breadth-first* search strategy to search for the correct solution. The algorithm starts at the root node, first explores the adjacent neighbor nodes starting from the left, only when all the nodes in the current level have been explored the algorithm moves down to the next level. The space complexity of *breadth-first* search can be expressed as \(O(v)\), where \(v\) is the number of vertices in the tree. However, if it is known that the search key is located in the leaf node, therefore this search strategy is not ideal for deep trees as some of the branches would never require exploration.
**Depth-First search**  This issue can be tackled with depth-first DFS search strategy. The algorithm, similarly to breadth-first, starts at the root node, exploring the branch until the leaf node is reached, and then backtracking to the first node with unexplored branches, this is repeated until the entire tree is explored. The worst time complexity of the algorithm is expressed with the same equation as breadth-first: $O(v)$. However, because the algorithm fully explores a branch before moving on to the next one, at best the space complexity will be $O(vb)$, where $vb$ is the number of vertices in the first branch. However, this method is not efficient if the goal node is shallow, but at the very right in the tree width.
Chapter 3

Analysis

In this chapter we analyse the performance of the existing tool and identify the performance bottlenecks. In the first section we discuss the test data used for performance analysis. In section 3.2 we identify 4 optimisation issues that affect the performance of the tool.

3.1 Test Data

To capture a wide range of test data, several Dafny programs were used for analysis. These programs were taken from the Dafny project site [16] and are listed in table 3.1. The programs use one of the two tactics to generate the proofs. The first tactic is listen in code sample 3.1.

```
tactic CasePerm(b : Element)
{
    cases b {
        tvar v := merge(variables(), params());
        tvar l ::= l in lemmas();
        tvar i ::= 0;
        while (i < 2)
        {
            perm(l, v);
            i ::= i + 1;
        }
    }
}
```

Code 3.1: CasePerm tactic [13]

The tactic is explained in detail in section 2.3.4, thus we will not go into detail of its’ workings. In short, given a datatype b it will generate a match statement, with up to two lemma calls for each of the case. An example of the generated proof by this tactic can be seen in code sample 3.2.
### Table 3.1: Test programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lemma</th>
<th>Argument Count</th>
<th>Local Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BreadthFirstSearch</strong></td>
<td>( \text{Lemma_IsPath_Closure} )</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>( \text{Lemma_IsPath_R} )</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td><strong>Substitution</strong></td>
<td>( \text{Lemma} )</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>( \text{Theorem} )</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>NipkowKlein_chapter3</strong></td>
<td>( \text{AsimpConst} )</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>( \text{BsimpCorrect} )</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td><strong>InductionVsCoinduction</strong></td>
<td>( \text{SAappendIsAssociativeC} )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td><strong>CoqArt_InsertionSort</strong></td>
<td>( \text{aux_equiv} )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{aux_sorted} )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Streams</strong></td>
<td>( \text{Theorem0} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Theorem1} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Theorem3} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Theorem4} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Prepend_Lemma} )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Lemma_FlattenAppend0} )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>( \text{Lemma_FlattenAppend0} )</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Code 3.2: AsimpConst lemma

```latex
\text{lemma \ AsimpConst}(a: \ aexp, s: \ state) \\
\text{requires \ Total}(s) \\
\text{ensures \ aval}(\text{asimp\_const}(a), s) = \text{aval}(a, s) \\
\{
\text{match} \ a \\
\text{\; \ case} \ N(n) \Rightarrow \\
\text{\; \ case} \ V(x) \Rightarrow \\
\text{\; \ case} \ \text{Plus}(a0, a1) \Rightarrow \\
\text{\; \ \; \ AsimpConst}(a0, s); \ AsimpConst(a1, s); \\
\}\n```

However, the tactic has a limitation. The variable \( l \) is bound to a single lemma, thus it is impossible to generate two distinct lemma applications. To counter this we have developed a tactic seen in example 3.3.

#### Code 3.3: CasePerm2 tactic

```latex
\text{tactic \ CasePerm2}(b: \ Element) \\
\{
\text{cases} \ b \{
\text{\; \ \; \ tvar} \ v := \text{merge}(\text{variables}(), \text{params}()); \\
\text{\; \ \; \ tvar} \ l :: | l \text{ \ in} \ \text{lemmas}(); \\
\text{\; \ \; \ tvar} \ l1 :: | l1 \text{ \ in} \ \text{lemmas}() \& l1 \neq l; \\
\text{\; \ \; \ perm}(l, v); \ \text{perm}(l1, v); \\
\}\n\}
```


The CasePerm2 tactic, for each lemma \( l \) in the program, takes a different lemma \( l_1 \), and generates two calls, using any combination of variables \( v \) as input. The two 'such that' statements, will generate \( x(x - 1) \) solutions, where \( x \) is the number of lemmas in the program. Furthermore, for each of the solutions, another \( |perm1| \times |perm2| \) solutions are generated. To illustrate, let us assume there are 4 lemmas in the program, each program takes 3 arguments as an input and there are 6 variables in \( v \). First permutation will generate \( 6^3 \) combinations, then for each of these, the second permutation call will generate another \( 6^3 \) calls. In total, the cases body will generate:

\[
4(4 - 1) \times 6^3 \times 6^3 = 12 \times 6^6 = 559,872 \text{ solutions}
\]

As we can see in the following section, CasePerm2 generates much larger search spaces than CasePerm, thus it is ideal for analysing the performance of the tool.

The BreadthFirstSearch.dfy program verifies the BreadthFirstSearch algorithm.

```latex
\begin{verbatim}
datatype List<T> = Nil | Cons(head: T, tail: List)

lemma Lemma_IsPath_Closure(source: Vertex, dest: Vertex, p: List<Vertex>, AllVertices: set<Vertex>)
    requires IsPath(source, dest, p) \land source in AllVertices \land
    IsClosed(AllVertices);
    ensures dest in AllVertices \land
    \forall v \cdot v in elements(p) \implies v in AllVertices;
{
    match p {
        case Nil \Rightarrow
        case Cons(v, tail) ⇒
            Lemma_IsPath_Closure(source, v, tail, AllVertices);
    }
}

lemma Lemma_IsPath_R(source: Vertex, x: Vertex, p: List<Vertex>, AllVertices: set<Vertex>)
    requires IsPath(source, x, p) \land source in AllVertices \land
    IsClosed(AllVertices);
    ensures x in R(source, length(p), AllVertices);
{
    match p {
        case Nil \Rightarrow
        case Cons(v, tail) ⇒
            Lemma_IsPath_Closure(source, x, p, AllVertices);
            Lemma_IsPath_R(source, v, tail, AllVertices);
    }
}
\end{verbatim}
```

Code 3.4: Original BreadthFirstSearch lemmas

The first lemma Lemma_IsPath_Closure takes 4 arguments: source is the starting point in the tree, dest is the destination and p is the path from source to dest. The path is defined as a List datatype, which is either empty, or a head, that holds the value and the tail, that refers
to the remaining list. The set \texttt{AllVertices} is the whole search space. The lemma proves, that, if \texttt{p} is the path from \texttt{source} to \texttt{dest}, and \texttt{source} is in the search space, then every element of \texttt{p} is also an element of the search space. The proof is done by carrying out a pattern match on \texttt{p}, and calling itself, using the head of the list as the destination, and the tail of the list as the path to the destination.

The second lemma \texttt{Lemma.IsPath.R} takes the same arguments as \texttt{Lemma.IsPath.Closure}. It proves that given the same condition as in the previous example, the destination can be reached in number of steps equal to the number of vertices in the path list. To prove this property, we use pattern matching on the path list. If the list is empty we do nothing, otherwise \texttt{Lemma.IsPath.Closure} is called to prove that \texttt{p} is part of the search space, and then calling itself, to prove the same property for the head and the tail of of path.

The \texttt{Substitution.dfy} models variable substitution in expressions. The data types and lemmas are given in the sample 3.5.

\begin{verbatim}
datatype List = Nil | Cons(Expr, List)
datatype Expr = Const(int) | Var(int) | Nary(int, List)

lemma Theorem(e: Expr, v: int, val: int)
  ensures Subst(Subst(e, v, val), v, val) = Subst(e, v, val);
{
  match e {
    case Const(c) =>
    case Var(x) =>
    case Nary(op, args) =>
      Lemma(args, v, val);
  }
}

lemma Lemma(l: List, v: int, val: int)
  ensures SubstList(SubstList(l, v, val), v, val) = SubstList(l, v, val);
{
  match l {
    case Nil =>
    case Cons(e, tail) =>
      Lemma(tail, v, val);
      Theorem(e, v, val);
  }
}
\end{verbatim}

\textbf{Code 3.5: Substitution.dfy} lemma signatures

An expression is either a constant (\texttt{Const(int)}), a variable, which is identified by an integer (\texttt{Var(int)}), or an operation (\texttt{Nary(int, List)}), where \texttt{List}, is a list of expressions. \texttt{Theorem} takes 3 arguments as an input, an expression, a variable and the value for the variable. The lemma proves that substituting variable \texttt{v} for value \texttt{val} in the expression \texttt{e} replaces all occur-
references of \( v \) with \( \text{val} \). The proof pattern matches on the expression, and calls the lemma \text{Lemma} for the operation constructor. The \text{Lemma} proves a similar property as \text{Theorem}, but for lists. It takes 3 arguments as an input, \( l \) a list of expressions, \( v \) target variable and the value \( \text{val} \). The lemma proves, that substituting \( v \) for \( \text{val} \) in every expression in \( l \), replaces \( v \) with \( \text{val} \) in every member of \( l \). The lemma performs pattern matching on the input list, it calls \text{Theorem} for the head of the list, and calls itself for the tail of the list.

The \text{CoqArt-InsertionSort.dfy} example is a special case to showcase that tactics generate the simplest proof for a problem. The lemma is listed in code example 3.6.

```coq
lemma aux_sorted(l: List<int>, x: int)
  requires sorted(l);
  ensures sorted(aux(x, l));
{
  match l {
  case Nil ⇒
  case Cons(_, l′) ⇒
    match l′ {
    case Nil ⇒
    case Cons(_, _) ⇒
  }
  }
}
```

\text{Code 3.6: CoqArt-InsertionSort.dfy lemma}

The \text{aux_sorted} lemma proves that inserting argument \( x \) into the sorted list \( l \), will not disrupt the order of the list. The \text{sorted} function, takes a list \( l \) and proves that the list is sorted in descending order. The \text{aux} function, takes two arguments as an input, a sorted list \( l \) and a variable \( x \), and inserts \( x \) into the right position in \( l \). The lemma uses a nested pattern matching on the list to prove the property, however, it does not require any further guidance.

The \text{Streams.dfy} example is used to prove tactic re-usability and further test proof simplification. Each lemma contains a match statement with empty \text{case} bodies, an example can be seen in 3.7.

```coq
codatatype Stream<T> = Nil | Cons(head: T, tail: Stream)
colemma Theorem3(M: Stream<X>)
  ensures append(M, Nil) = M;
{
  match (M) {
  case Nil ⇒
  case Cons(x, N) ⇒ Theorem3(N);
  }
}
```

\text{Code 3.7: Streams.dfy lemma example}
CHAPTER 3. ANALYSIS

The codatatype $\text{Stream}$ defines an infinite list of values of type $T$. The colemma $\text{Theorem3}$, proves that appending a $\text{Nil}$ value to the stream does not change the stream. It performs pattern matching on the stream, if the list is empty nothing is done. Otherwise, $\text{Theorem3}$ is called to prove the property for the remainder of the list. By applying the $\text{CasePerm}$ tactic to this colemma we generate a simplified proof seen in example 3.8.

```
codatatype Stream<$T>$ = Nil | Cons(head:: $T$, tail:: Stream)

colemma Theorem3 (M:: Stream<$X$>)
    ensures append(M, Nil) == M;
{
    match (M) {
        case Nil =>
        case Cons(x, N) =>
    }
}

Code 3.8: Streams.dfy lemma result example
```

If we compare the two proofs, we notice that the generated proof is missing the recursive call, thus simplifying the proof.

### 3.2 Performance Analysis

**Data collection**  Each of the tests were executed 7 times. From the results, the smallest and the highest values were discarded and the remaining ones were averaged. This has been done in order to counter the inconsistency in the computer performance. To analyse the data, a number of different metrics were collected. Tactic resolution data was collected within the Tacny tool. To collect the resource data, the Tacny tool was executed from Python script, which allowed us to monitor and log the resources used by the process. Furthermore, the script supports parallelization, thus multiple instances of Tacny were ran at once.

To gain insight of the execution process of the tool, the Visual Studio debugger was used extensively. Visual Studio allows the user to attach the debugger to a running process. As the result, we were able to add break points to control the flow, view variable values, but most importantly see the stage of execution the program is in. By pausing the process mid execution, we can see if it is search for new solutions, or if it still generating the solutions.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel i7 2.20ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Memory (RAM)</td>
<td>4GB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows 10</td>
</tr>
</tbody>
</table>

Table 3.3: Computer parameters
<table>
<thead>
<tr>
<th>Program</th>
<th>Tactic</th>
<th>Exe time</th>
<th>#Nodes</th>
<th>#Inv Nodes</th>
<th>#Vld nodes</th>
<th>#VC Fail</th>
<th>#Boogie</th>
<th>Boogie Wait</th>
<th>#Dafny</th>
<th>Dafny Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreadthFirstSearch</td>
<td>CasePerm</td>
<td>12</td>
<td>26114</td>
<td>173</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>180</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm2</td>
<td>-</td>
<td>3359230</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Substitution</td>
<td>CasePerm2</td>
<td>127</td>
<td>578908</td>
<td>13316</td>
<td>19</td>
<td>18</td>
<td>19</td>
<td>23</td>
<td>13335</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>5</td>
<td>1127</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>NipkowKlein-Chapter 3</td>
<td>CasePerm</td>
<td>29</td>
<td>2106</td>
<td>442</td>
<td>22</td>
<td>21</td>
<td>22</td>
<td>22</td>
<td>464</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>21</td>
<td>4298</td>
<td>933</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>952</td>
<td>1</td>
</tr>
<tr>
<td>InductionVsCoinduction</td>
<td>CasePerm</td>
<td>13</td>
<td>1523</td>
<td>626</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>635</td>
<td>1</td>
</tr>
<tr>
<td>CoqArt-InsertionSort</td>
<td>CasePerm</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Streams</td>
<td>CasePerm</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CasePerm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Base performance data
Table 3.2 refers to the test data gathered during tactic resolution whilst running the original Tacny tool. Each of the columns refer to different parameters:

- **Program**: the name of the resolved program.
- **Tactic**: the executed tactic.
- **Exe time**: execution time for the tactic.
- **#Nodes**: the total number of generated solutions.
- **#Inv/#Vld Nodes**: number of valid and invalid solutions.
- **#VC Fail**: number of times verification failed.
- **#Boogie**: Number of times Boogie was called.
- **Boogie Wait**: Total time taken to verify the solutions.
- **#Dafny**: Number of times Dafny was called.
- **Dafny Wait**: Total time taken to resolve the solutions.

The number of calls made to Dafny also implies the explored search space before a solution was found.

We can observe that the test data for CasePerm2 in BreadthFirstSearch is not fully filled. This is due to the complexity of the program. It total search space of the tactic quickly overwhelms the memory capacity of the testing machine, thus, the total search space is an estimation, which was calculated as follows: There are 5 lemmas in total in the program, thus after executing the first three lines of the cases body there will be 20 solutions generated. Each lemma in the program, on average, takes 4 arguments. For the original body of the Lemma_IsPath_R lemma we can see that the pattern matching on the input list, introduces 2 extra variables Cons(v, tail), thus the total number of variables in tactic var v is 6.

Because, each lemma takes 4 variables as an input, there are $6^4$ combinations that the perm call will generate. Furthermore, the second perm call will generate another $6^4$ calls. The total number of generated solutions by the two perm calls is $6^8$.

Finally, the permutations will be carried for each combination of (1, 11) lemmas, thus the total size of the search space is 33.5 million solutions. None of the accessible hardware was powerful enough to execute this tactic, even doubling the size of the memory on the test machine did not yield any results. Therefore, the given search space size is an estimation. The second number we can estimate is the time required to type check every solution. By averaging the
Dafny Wait times, we get that a single program validation takes around 0.01 second. Thus the estimated time to type check the entire search space is 93.312 hours. Of course, in reality this number will be smaller, because it is unlikely that the entire search space will be explored to find the solution.

Evaluating the CoqArt-InsertionSort and the Streams produced unusual behaviour. From table 3.2 we can see that both example programs called Boogie for verification exactly once for each lemma. However, for both examples, verifying the first solution took 5 and 6 second respectively, where as the rest of the verification tasks took 1 second to execute. In fact, for all the test samples, the first call to the Boogie tool took much longer than the subsequent ones. The cause of this is the setup of Boogie: it takes more time to execute the first call because the tool is verifying the whole program, and then caching the results. The cashed results allow Boogie to verify only the parts of the program that have changed. This increases the execution time of the first call, but substantially reduces any subsequent verification tasks.

Comparing the execution times and the size of the search space, we can infer that the size of the search space greatly affects the execution time. Both BreadthFirstSearch and Substitution programs call the same tactics. Resolving the Lemma IsPath Closure lemma, generates 26 thousand solutions and takes 12 seconds to execute, where as Theorem, which generates 1127 solutions, only takes 5 seconds to execute. Similar property is observed from the Lemma and the Lemma IsPath R lemmas. The first one generated over 560 thousand nodes and took 127 seconds to resolve, and the the Lemma IsPath R lemma is estimated to generate over 33.5 million nodes, which are too much to validate. Therefore, to improve the execution time, it is in our best interest to reduce the search space as much as possible.

The single most important factor which affects the execution time is the location of the correct solution. That is, if talking about the solutions in terms of a search tree, the left most solution will be found much sooner than the right most. However, this factor highly depends on the overall order of the program, the order of the arguments passed to the lemma, locally declared variables etc. This factor is difficult to affect, therefore we will not consider it in this section. However, to a degree, we can change other performance affecting parameters. These parameters and their combination is the main cause of slow performance:

- Total number of generated branches.
- Total number of calls made to Dafny.

The generated branch count is the main culprit behind the space and time complexity of
CHAPTER 3. ANALYSIS

<table>
<thead>
<tr>
<th></th>
<th>Memory (MB)</th>
<th>Search Space</th>
<th>Time Taken (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BreadthFirstSearch</strong></td>
<td>6812</td>
<td>33618434</td>
<td>-</td>
</tr>
<tr>
<td><strong>Substitution</strong></td>
<td>2194.09</td>
<td>5800035</td>
<td>132</td>
</tr>
<tr>
<td><strong>NipkowKlein-chapter3</strong></td>
<td>101.94</td>
<td>6404</td>
<td>50</td>
</tr>
<tr>
<td><strong>InductionVsCoinduction</strong></td>
<td>85.1</td>
<td>1523</td>
<td>13</td>
</tr>
<tr>
<td><strong>Streams</strong></td>
<td>67.17</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td><strong>CoqArt-InsertionSort</strong></td>
<td>63.55</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3.4: Search Space vs Execution Time

the tool. Because original implementation is eager, before solution analysis starts, the whole search space must be generated first. Table 3.4 illustrates the time, search space and memory usage comparison. **Memory** refers to memory usage in megabytes, **Search Space** refers to the generated search space for the whole program, and the **Execution Time** is the time taken to resolve all tactic calls.

**Execution Time Analysis** From the table we can see that the generated search space directly affects the memory usage and the execution time. **BreadthFirstSearch** sample illustrates the relationship between the three fields quite well. Loading the running Tacny process in debug mode revealed that the tool ran out of memory before it finished generating the search space. A similar observation can be made for the **Substitution** example. The collected metrics show that it took **132** seconds to resolve the whole program, of which, **47** seconds were spent waiting for Dafny and Boogie, thus the remaining **85** seconds were spent generating the solutions. The **NipkowKlein-Chapter3** program resolution took **50** seconds, of which **43** seconds were spent waiting Dafny and Boogie, and it took **7** seconds to generate the search space. Because the total search space is smaller, it took much less time to generate it. These examples show that generating the search space eagerly, especially when the problem space is large, takes a lot of time and memory. Therefore, to improve this, there is a need to implement a different evaluation strategy.

**Generated Search Space** In addition, the size of the search space affects the number of calls made to Dafny, because each final solution has to be type-checked before verification. From the test data we can observe that the **Substitution** sample program generated **580** thousand solutions and made **13** thousand calls to Dafny and waited for it for **23** seconds. In comparison, the **NipkowKlein-chapter3** example, in total generated **6.4** thousand solutions, made **1.4** thousand calls to Dafny and spent only **4** seconds waiting for it. We can observe a trend in the data, the greater the search space, the larger memory usage and the longer execution.
time. To improve the overall performance of the tool the search space has to be reduced to the bare minimum. However, the generated search space, does not affect the number of calls made to Boogie. This is because the number of type-correct solutions is not affected by the number of invalid solutions. The issue is that, the few valid solutions in the search space are obscured by the high number of invalid ones.

**Search Strategy and Sequential Resolution** As mentioned in section 2.3.4, Tacny uses BFS strategy to traverse the search tree. However, in some cases this may not be the best strategy. Consider the *NipkowKlein-Chapter3*, we know, that for the majority of cases, the tactics have to generate two calls for the *case* body. Consider one of the solutions generated mid resolution, the resulting program is listed in the code example \[\text{3.9}\]

```haskell
lemma BsimpCorrect (b: bexp, s: state) 
  requires Total(s) 
  ensures bval(bsimp(b), s) = bval(b, s) 
{ 
  match b 
    case Bc(v) ⇒ 
    case Not(op) ⇒ 
    case And(a0, a1) ⇒ 
    case Less(l0, l1) ⇒ 
      AsimpCorrect(b, a1); 
}
```

*Code 3.9: NipkowKlein-Chapter3* bad solution

From the original lemma we know that the *Less* case requires two *AsimpCorrect* calls. However, due to the nature of BFS strategy, Tacny will first explore all single lemma call solutions, and only then go down to the leaf nodes to explore the two call solutions. To illustrate this consider the search tree for the *BsimpCorrect* lemma from the *NipkowKlein-Chapter3* test program, seen in figure \[\text{3.1}\]. The valid solution is marked as [Valid], to reach this solution the BFS algorithm will first traverse the first (1) layer, and only then move down to the second (2) layer. The [Valid] solution is the child node of the *AsimpCorrect* node, thus, exploring the first layer any further than the first node is wasteful. Therefore, a different search strategy should be used to traverse this tree. The tool should support more than one search strategy, and provide non-intrusive means to switch in-between them. For this particular example, DFS strategy should improve the performance, as it will fully explore the *AsipCorrect* branch and terminate, thus improving the execution time as none other nodes on the first layer are explored. In addition, all tactic calls in a single program are executed sequentially. That is, Tacny tool scans the program, finds a tactic call, resolves the call, and continues scanning until the program
is exhausted. To improve the execution time one could resolve tactic calls in parallel, thus improving the execution speed.
Chapter 4

Design

In last chapter we have analysed the base performance of the tool and identified 4 performance bottlenecks:

- The size of the generated search space
- Eager solution generation
- Inefficient search strategy
- Sequential tactic resolution

In this chapter we will design an optimisation for each of the bottlenecks.

4.1 Type lookup

The size of the generated search space directly affects how long it will take to resolve a tactic. The test data also showed that most of the visited branches before the valid solution was found failed to resolve, and it is unclear how many more branches in the generated search space are type incorrect. The first step in reducing the search space is to prevent the type incorrect branches from being generated.

Error: incorrect type of method in-parameter 2 (expected int, got List)

Code 4.1: Error message

The resolution errors, such as one seen in figure 4.1, tell us that type-checking fails because the permutations, generated by the perm atom, are passed incorrect type arguments. The perm atom does not carry out type-checking during call generation therefore, it will generate a number of invalid solutions. Tactic, which use this atom, will have a larger search space. It is impossible to predetermine the variable types that are passed to the perm atom, unless its’ type
is specified by the user, because Dafny variable types are resolved at runtime. As the result, during call generation, it is unknown whether passed arguments to the generated method call are type-correct.

This issue can be solved by a type lookup table. The table stores (var, type) pairs, where var is the variable name and type is a Dafny type associated with the variable. The perm atom can refer to the table to type-check the variables passed to method application being generated. The variable types can be acquired in two ways: If the variable type is given by the user, we just add the variable type pair to the table. Otherwise, we send the method to the Dafny resolver where the types are determined, and extract the types from the resolved method.

To design of the type lookup table only required small updates the the system:

First, the lookup table is added to the global context. The lookup table is written into the global context so that in case the tactic contains nested tactic application, the types would be accessible. Secondly, during tactic application search, if a tactic application is found in the method body, the method is sent to the Dafny resolver, where the types of declared variables, are resolved. The resulting method, is iterated, adding a new (var, type) pair to the lookup table. Afterwards, the tactic resolution is resumed. Once the resolution is finished the lookup table is cleared. This is done to avoid any inconsistencies when resolving any other tactic calls.

The perm atom has to be updated to access the lookup table. As discussed the in the Tacny implementation section 2.3.4, the perm(lm, vars) will create list of variables arg list for each argument the method lm takes and fill every list with variables from vars. Afterwards the lists are permuted to generate all possible applications of lm. To ensure that only type correct applications are generated, each arg list is assigned a type based on the type of the corresponding argument. For each variable v in vars, we get its’ type and insert it to every arg list the type matches. Once the vars list is exhausted, each arg list is check to ensure it is not empty. If the argument list is empty we can safely terminate the tactic generation, as the atom will not generate any applications.

4.2 Lazy evaluation

As discussed in the previous chapter, eager tactic evaluation causes high memory usage and increased execution time. To illustrate the main drawback of greedy evaluation consider the search tree in figure 4.1. The numbers in the node represent the order it was generated in. We
We use the default BFS strategy to traverse the tree. First the search space is generated, then it is traversed to find the correct solution. The search starts at the root node, and each layer is explored from left to right. The 3rd node is final, thus, we try to resolve and verify it. However it fails to type-check, thus the search is resumed. The next node in line is node 4, again, it is final thus we resolve and verify it. The node is successfully verified and the algorithm terminates.

We can see that to reach the 4th node, we did not need to generate any of leaf solutions for node 2. However, because the tree is generated before it is traversed this cannot be avoided, thus memory and execution time is wasted. A second drawback, is that even if the 3rd branch failed to type-check it is not discarded from memory, it persists in memory until the algorithm is terminated.

Lazy evaluation, generates the tree one node at a time, once a node is generated, if it is final, the node is resolved and verified, otherwise it is resolved further. The algorithm will generate the smallest search space required to find the correct solution. To illustrate how lazy evaluation works consider the search tree in figure 4.2. It is the same search tree as in previous example generated using lazy evaluation. The dashed line indicates the search space which will not be generated, it was generated as follows:

We generate the unresolved tactic to the root solution and start resolving it. A check is car-
ried out whether the 1st generated solution is final. The check fails, thus the 2nd solution is generated. Because the 1st solution is not final, further resolution yields node 3. The 3rd node is final thus it is resolved by Dafny resolver however it fails to type-check, thus it is discarded. We further try to evaluate the 1st node, which yields node 4. It is final, therefore it is resolved and successfully verified, thus the algorithm is terminated. The algorithm was terminated after node 4 was verified, thus none of the children of node 2 were generated.

To implement lazy evaluation in C# the `yield` keyword was used. The `yield` keyword allows to yield a single result from the method, and save the state of the execution in memory, this is done by constraining the method return types to `IEnumerable<T>`. The `IEnumerable` interface exposes the required properties, so that the method could be iterated in a `foreach` loop.

The yield keyword has two uses:

- **yield return `<expression>`**: to return a single result.
- **yield break**: to terminate the iterator.

The method, which uses the yield keyword is consumed as a standard iterator either by using a language integrated query or a `foreach` loop. Each iteration of the loop calls the iterator method, when the yield return statement is found the `<expression>` is returned, and the state of the iterator is saved in memory. On further method calls the execution is resumed. When the yield break statement is reached the iterator terminates.

To implement the lazy evaluation the Tacny framework has to be heavily modified. As described in section 2.3.4, each atomic statement resolver implements the `IAtomicStmt` interface, which corresponds to the function:

```
solution → List < Solution >
```

Given a single solution it will return a list of solutions. In figure 4.3 we can see the updated signature for lazy evaluation. Instead of generating all the solutions and returning a list, the updated resolver takes a single `Solution` as an input and returns an enumerator `IEnumerable`,
which generates the solutions one at a time. In other words, the function has been changed to:  
\[
solution \rightarrow \text{IEnumerable<Solution>}
\]
A function which takes a single solution as an argument and returns an enumerator for solutions. In addition to updating the interface, a number of other updates had to be made, these are discussed in detail in section 5.2.

### 4.3 Search framework

Currently Tacny tool supports a single search framework, in the analysis section 3.2 we have showed that under some cases it would be more efficient to run the DFS strategy instead of the BFS. In this section we will design a framework, which would allow the users to implement multiple search strategies and discuss possible ways to specify which strategy should be used.

Figure 4.4 shows the overall structure for the search framework. The `SearchStrategy` is the master class which implements the `ISearch` interface. The interface defines the signature for the `Search` method which holds the logic for each search strategy. It takes the unresolve tactic `atomic` and a boolean flag `verify` as an input. The unresolved tactic is used to generate the initial root node, and the boolean flag is used to indicate whether a final solution should be verified. This flag is used for traversing atomic statements with bodies, such as `cases` or `while`, where the final atomic in the statement body does not have to be resolved, as it is resolved by
the parent search strategy.

In addition, the `Search` method is used as the single point of entry into the `SearchStrategy` class. It provides a single access point for each search strategy defined in the framework. It holds the logic for executing the search strategy based on the `ActiveStrategy` variable. The `ActiveStrategy` variable is an instance of the `Strategy` type. The type uniquely identifies each implemented search strategy. Each defined strategy inherits the `SearchStrategy` class and holds the logic of the algorithm inside the `Search` method.

An important objective of Tacny tool is to ensure that the tools’ syntax is as non-intrusive as possible. That is why we chose to use exiting attributes system to specify the search strategy. Dafny attributes take the form of `attribute <expression>`. It is an attribute followed by an expression, which is most commonly a literal expression. The code sample 4.2 illustrates how search strategies are declared. The tactic is given the `search` attribute followed by the `<search-strategy>` name.

```daml
  tactic {:search <search-strategy>} myTactic(...) { ... }
```

Code 4.2: Search strategy

In addition to specifying a search strategy for the whole tactic, users should be able to set a strategy for the atomic tactics, which have bodies, such as `cases`, `while` etc. To specify a search strategy for a nested body, we propose the `set_strategy(<search-strategy>);` expression. It takes the name of the strategy as an input, and changes the search strategy for the local scope. Consider the tactic example listen in figure 4.3.

```daml
  tactic {:search BFS} MultipleStrategies(b : Element) {
    cases(b) {
      set_strategy(DFS);
      tvar args = merge(variables(), params());
      tvar l :| | in lemmas();
      perm(l, args);
    }
  }
```

Code 4.3: Expression `set_strategy()`

The `:search` attribute attributes sets the search strategy to BFS for the whole body. When the tactic resolver enters the `cases(b)` body, the strategy is switched to DFS for that scope, once the resolver exits the cases atom, the strategy is switched back to BFS. If the strategy is switched in the middle of the scope, any following statements inside the tactic would be resolved using the new strategy. This expression will allow the user to specify how each part of the tree is generated, thus tailoring the performance of the tactic.
4.4 Parallel execution

Currently tactics are executed in sequence, thus only a single processor is being used. To explore all the resources of the computer tactics should be executed in parallel. The parallelization can be implemented at two levels:

Firstly the parallelization could implemented at program level. Currently, to find tactic application two nested foreach loops are used. The outer loop iterates the program member list and the inner loop goes over each statement in the body of the member and calls the Tacny resolver for every found tactic application. To parallelize this, we create a thread for each member of the program. The thread is responsible for resolving each tactic call in the member body, and returning the result. We want to use one thread for each member in case there are multiple tactic calls in the body. Resolving two or more tactic calls in a single body with multiple threads will lead to a race condition when generating the final result, thus it is safer to use a single thread to resolve all tactic calls in a single member.

Initial experiments showed that all threads bottleneck when calling Boogie and execution time is significantly increased. The reason for this is that Boogie is called as a static library, thus to avoid unexpected behaviour from race conditions, a lock had to be added at the location Boogie is called so only a single thread is using the library at any given time. When the block is locked, other threads are queued at the lock on the first in first out basis.

Secondly, parallelization could be implemented at tactic level. The search engine would create a number of threads, based on the system the tool is executed on, to generate and traverse the search space. Once a thread finds a solution the code execution is terminated. However, similar limitations apply as only one thread can call Boogie at a time. Parallel execution seems to be a great way to improve the performance of the tool, however, more experiments have to be carried out to produce any concrete results.
Chapter 5

Implementation

This chapter holds the implementation details for each optimisation. Each optimization can be toggled on and off by providing the associated command line argument. This allows for a detailed analysis on how each of the optimizations affect the performance and discover any emergent behaviour.

5.1 Type lookup

The type lookup table is represented as a dictionary `Dictionary<IVariable, Type>`. The `IVariable` is an interface, which every Dafny variable implements, the `Type` is a Dafny type, which is one of the base types such as, integer, char, boolean or real, or a rich type like array, string etc. The lookup table is placed into the `GlobalContext`, which can be accessed by nested tactic resolvers.

![Algorithm 1 ScanProgram pseudo-code](image)

Figure 5.1 refers to the pseudo-code of the `ScanProgram` algorithm. The method takes an
instance of Dafny program extended with tactics, and calls the ResolveTactic method for each tactic application. We iterate the members of the program, for each statement in the members’ body we check if the statement is a tactic application by calling the IsTacticCall method. The method checks if there exists a tactic in the program, with the same name as the statement signature. If the statement is a tactic application, we copy the member, remove tactic calls from the copy, and resolve it. We then call the ExtractTypes method that creates a dictionary containing (variable, type) pairs for each variable declaration. The dictionary, the original method and the statement are used to create a new instance of the resolver state, which is passed to the ResolveTactic method.

Algorithm 2 Updated Perm Algorithm

```
procedure Perm(method Member, varList List<IVariable>)
    args ← new List<List<IVariable>> (method.Input.Count)
    for i = 0; i < method.Input.Count; i + + do
        type ← method.Input[i].GetType()
        current ← new List<IVariable> ()
        for var ∈ varList do
            if context.LookupTable[var] == type then
                current.Append(var)
            end if
        end for
        if current.Count == 0 then
            yield break
        end if
        args[i] ← current
    end for
    for solution ∈ PermuteInput(method, args) do
        yield return solution
    end for
    yield break
```

The Perm atom resolver was extended with a call to the lookup table. In figure 5.2 we can see the update, the Perm method takes a method of type Member, this is the uniform method for all Dafny program members, and a list of variables varList. We declare a two dimensional list args, which will hold the type correct variables for each argument the input method takes. For each method argument, we get its’ type, declare an intermediate list current for holding the variables, and iterate the varList. In the nested For loop, the variable type is fetched from the lookup table, if the type matches the type of the method argument, the variable is added to the intermediate list. When the varList is exhausted, if the current list is empty the algorithm can be terminated as no valid applications can be generated. Otherwise the list is added to the args list. Finally, the PermuteInput method is called, which creates a single method application at a time using args as input.
5.2 Lazy evaluation

As discussed in the previous chapter, lazy evaluation was implemented using the \texttt{yield} keyword. To support it Tacny required an extensive reconstruction. Each atomic statement resolver was rewritten to implement the new \texttt{IAtomicLazyStmt} interface. However, it was not enough to just implement the interface, the logic of each atomic was changed to generate solutions one at a time. The atomic statement resolvers can be classed in to two categories:

- Single solution resolvers, which are guaranteed to generate a single solution. These include expressions such as \texttt{lemmas()}, \texttt{variables()} etc.
- Multi-branching solutions that generate zero or more solutions. These are \texttt{perm}, \texttt{cases}, \texttt{such-that} etc.

Algorithm 5.3 refers to the pseudo-code of a single branching atomic statement.

\begin{algorithm}
\caption{Single branch resolution}
\begin{algorithmic}
\Procedure{ResolveStatement}{statement : UpdateStmt}
    \State \texttt{result} $\leftarrow$ \texttt{ExecuteLogic}(statement)
    \State \texttt{yield return} result
    \State \texttt{yield break}
\EndProcedure
\end{algorithmic}
\end{algorithm}

The method takes the call to the atom statement as an input, which is used to access any arguments passed to the statement. The \texttt{ExecuteLogic} method contains the logic for the particular atomic statement resolver, it is guaranteed to return a single solution. Once the solution is generated, it is yielded. The next iteration of the method will hit the \texttt{yield break} statement, and the iterator will terminate. In the code example 5.1 we can see the original implementation of the \texttt{Id()} atom. The atom is the identity atom, which returns a solution that is resolved to \texttt{true}.

\begin{verbatim}
public void Resolve(Statement st, ref List<Solution> solution_list) {
    List<Expression> args = null;
    IVariant localVar = null;
    InitArgs(st, out localVar, out args);
    Dafny.LiteralExpr lit = new Dafny.LiteralExpr(st.Tok, true);
    localContext.AddLocal(localVariable, lit);
    solution_list.Add(new Solution(this.Copy()));
}
\end{verbatim}

Code 5.1: Eager \texttt{Id()} implementation

The method takes a \texttt{Statement} \texttt{st} and a reference to the \texttt{List<Solution> solution_list} as arguments. First we declare an expression list \texttt{args} which holds any arguments passed to the
atom. The `localVariable` declaration is used if the atom is being assigned to a value e.g. `tvar id := id();`. The `InitArgs` call initializes the arguments and the `localVariable`. We create an instance of literal expression `lit`, that has value `true` assigned to it and register it as a local declaration. Finally, a new solutions created from a copy of the resolver state, and is added to the solution list.

```csharp
class IdAtomic : Atomic, IAtomicLazyStmt {
    public IEnumerable<Solution> Resolve(Statement st, Solution solution) {
        List<Expression> args = null;
        IVariable lv = null;
        InitArgs(st, out lv, out args);
        Dafny.LiteralExpr lit = new Dafny.LiteralExpr(st.Tok, true);
        localContext.AddLocal(lv, lit);
        yield return new Solution(this.Copy());
        yield break;
    }
}
```

Code 5.2: Lazy Id() implementation

The updated version of the same atom is listen in figure 5.2. The method signature has been changed, the method return type was changed to `IEnumerable<Solution>` and the list of solutions has been changed to a single solution. The logic of the method has not changed, the `solution_list.Add(new Solution(this.Copy()));` was replaced with `yield return new Solution(this.Copy());` and a break statement to terminate the iterator.

**Algorithm 4** Multi branching resolution

```csharp
procedure ResolveStatement(statement : UpdateStmt)
    for result ∈ ExecuteLogic(statement) do
        yield return result
        yield break
```

The algorithm for resolving multi-branch statements can be seen in algorithm 4. The method takes the same input as the single branch resolver. The `ExecuteLogic` method creates a solution one at a time, each solution is yielded. Once the all the solutions have been generated the iterator is terminated with the `yield break` statement. The code sample 5.3 depicts the implementation of of meta-level `while` statement.

```csharp
class WhileAtomic : BlockAtomic, IAtomicStmt {
    public void Resolve(WhileStmt whileStmt, ref List<Solution> solution_list) {
        if (EvaluateGuard(whileStmt)) {
            List<Solution> result = null;
            ResolveBody(whileStmt.Body, out result);
        }
    }
}
```
CHAPTER 5. IMPLEMENTATION

The `EvaluateGuard` method takes a `whileStmt` instance and returns a boolean value indicating if the guard is resolved to true or false. If the guard is evaluated to true, a list for solutions is declared and the body of the while statement is resolved in the `ResolveBody` method. The method takes a statement body and a reference to the result list, and evaluates the body using the BFS strategy. The result is added to the solution list. If the guard is evaluated to false the method execution terminates. The same method implemented using lazy evaluation is listed in code sample 5.4.

```csharp
class WhileAtomic : BlockAtomic, IAtomicLaStmt {
    private IEnumerable<Solution> ExecuteLoop(WhileStmt whileStmt, Solution solution) {
        if (EvaluateGuard(whileStmt)) {
            foreach (var item in ResolveBody(whileStmt.Body)) {
                yield return item;
            }
        }
        yield break;
    }
}
```

Code 5.4: Lazy meta-level `while` statement implementation

The body of the resolver was updated to return solutions generated by the `ResolveBody` method one at a time.

The eager BFS strategy used to resolve the body was changed to lazy evaluation. The pseudocode for the lazy BFS strategy is listed in figure 5.

```
Algorithm 5 Tactic Resolution Algorithm

procedure LAZYBREADTHFIRSTSEARCH(state : Atomic)
    Result ← INITQUEUE(newSolution(state))
    while !Result.IsEmpty() do
        solution ← Result.Dequeue()
        for each item ∈ RESOLVESTATEMENT(solution.state.statement, solution) do
            if item.IsFinal() then
                yield return item
            else
                Result.Enqueue(item)
        yield break

The queue Result is initialized with the unresolved state, which can either be a tactic, or a
statement body. While the queue is not empty, a solution is dequeued, and the last unresolved statement is resolved. As we showed earlier, ResolveStatement returns an iterator, if the generated solution is final, it is yielded, otherwise the result is added to the back of the queue for further resolution. Once the queue is exhausted the iterator is terminated.

We can now show the updated ResolveBody method. The source code is given in code example 5.5.

```csharp
public IEnumerable<Solution> ResolveBody(BlockStmt body)
{
    ISearch strat = new SearchStrategy(Strategy.BFS);
    Atomic ac = this.Copy();
    ac.localContext.tacticBody = body.Body;
    ac.localContext.ResetCounter();
    foreach (var result in strat.Search(ac, false))
    {
        result.state.localContext.tacticBody = this.localContext.tacticBody;
        result.state.localContext.tac_call = this.localContext.tac_call;
        result.state.localContext.SetCounter(this.localContext.GetCounter());
        yield return result;
    }
    yield break;
}
```

Code 5.5: Lazy body resolution

The ResolveBody method takes a block statement as an input. We initialize the search framework strat and create a new copy ac of the current resolver state is then created, this is done so that we could separate the scopes of the outer and inner bodies. The state still contains the old body and the old body counter, therefore, we update the body and reset the counter to 0. The fresh state is passed to the search strategy. For each generated solution, the state body and counter are changed back to the original value, and the result is yielded. Finally, once the foreach loop terminates, the iterator is stopped.

The implementation has uncovered an unexpected property. The solutions are generated one at a time and only non-final solutions are added to the intermediate list for further evaluation. The final solutions are generated to programs, resolved and verified. If it fails either of those, the solution is discarded and the memory space it occupied is released. Therefore, at most, there is only a single final solution in memory, we discuss the implications in detail in section 6.2.2.

5.3 Search Framework

In the design chapter we discussed the design features of the search framework. In this section we discuss the implementation details of the design.
Firstly, the ResolveTactic and ResolveBody methods were updated. Originally, the search strategy was implemented directly in the body of the method. However, to support multiple strategies, the methods were updated to call an instance of the SearchStrategy class. The code example 5.5 displays the updated ResolveTactic method body, similar changes were also made in the ResolveBody method. We declare an instance of the SearchStrategy class, the argument passed to the constructor specifies the active strategy of the framework. In the foreach loop guard, we call the Search method, which executes the strategy given in the constructor.

The implementation details of the Search method are given in figure 5.6. The Search method takes the resolver state atomic and the verify flag as an input. The state is used to create the root solution of the search tree, and the flag indicates whether the leaf nodes should be verified. The flag is set to false when the strategy is called from the ResolveBody method. The reason for that is, nested body leaf nodes, may not be the leaf nodes of the enclosing body, thus the solutions are only verified at the top most search strategy. In the method body we declare IEnumerable<Solution> enumerable and pattern match on the ActiveStrategy. Each case block is assigned to an implemented strategy where a value to the enumerable is assigned. Finally, the instance of the enumerable is returned.

```csharp
public class SearchStrategy : ISearch {

    public IEnumerable<Solution> Search(Atomic atomic, bool verify = true) {
        IEnumerable<Solution> enumerable = null;
        switch (ActiveStrategy) {
            case Strategy.BFS:
                enumerable = BreadthFirstSeach.Search(atomic, verify);
                break;
            case Strategy.DFS:
                enumerable = DepthFirstSeach.Search(atomic, verify);
                break;
            default:
                enumerable = BreadthFirstSeach.Search(atomic, verify);
                break;
        }
        return enumerable;
    }
}
```

Code 5.6: SearchStrategy.Search implementation

The SearchStrategy class also holds two helper methods:

- **GetSearchStrategy** method is used to extract the :search attribute from a tactic, and return corresponding identifier. The

- **VerifySolution** method is used to verify a final solution, the method was added to reduce code duplication across the different search strategies.
CHAPTER 5. IMPLEMENTATION

Algorithm 6 Depth First Search algorithm

```plaintext
procedure DepthFirstSearch(state : Atomic, verify : Bool)
    solution ← new Solution(state)
    enumerator ← Atomic.ResolveStatement(solution)
    stack ← new Stack(enumerator)
    while !Stack.IsEmpty() do
        enum ← stack.pop()
        sol ← enum.GetNext()
        if sol == Nil then
            continue
        stack.push(enum)
        if sol.IsFinal() then
            if verify then
                if sol.ResolveAndVerify() then
                    yield return sol
                    yield break
                else
                    yield return sol
                    yield break
            else
                enum ← Atomic.ResolveStatement(sol)
                stack.Push(enum)
            yield break
        end
    end
```

Depth-First-Search We gave the algorithm for the lazy BFS strategy in the previous section, therefore we will only focus on the DFS implementation. The motivation behind the DFS strategy is to find the correct solution with minimum expansion of the tree. The final solutions are guaranteed to be at the leaf nodes of the tree, thus, if we were to use BFS to find the correct solution, there will be branches generated, that do not lead to the final solution. On the other hand, DFS strategy prioritizes exploring the depth of the tree before branching to the side, thus, in cases when the final solution is at the bottom of the tree, it will generate fewer nodes to reach it.

Figure 6 shows the pseudo-code of the DFS strategy implementation. We initialize the root solution from the state and create an enumerator for resolving the statement, and push it to the stack. While the stack is not empty, we pop the enumerator and call the `MoveNext()` method which in turn resumes the execution of the atom statement resolver. If the generated solution `sol` is null, the atomic statement resolver has terminated, and the iterator will not return any solutions, therefore it is discarded, and a new enumerator is popped from the stack. Otherwise, if the enumerator is not exhausted it is pushed back on top of the stack. If the generated solution is final, then depending on the `verify` flag we either try to resolve and verify the solution, or
yield it without resolution. If the solution is not final, we call the ResolveStatement method to further resolve the state and push the resulting enumerator back on to the stack. Once the stack is depleted the algorithm is terminated.

The optimisation introduced an extensible search framework, and a new DFS search strategy. As discussed in section 6.2.3 in some cases DFS strategy outperforms BFS strategy, but it cannot replace it. However, the framework provides the necessary means to implement multiple strategies, therefore, in the future we will be able to explore more complicated heuristic search algorithms such as Best-First-Search or A.
Chapter 6

Evaluation

6.1 Base performance

The aim of the project was to improve the execution time and reduce the memory on the Tacny tool. In this section we will analyse how the optimisations affected the performance. All test data in this section was collected by running the tool with all optimisations enabled.

6.1.1 Execution time analysis

Figure 6.1 displays execution time comparison of both tools. Because the data ranges from 7 to 127 seconds, the graph is given in logarithmic base 5 scale, to ensure the all the data points can be clearly seen. The ■ shape refers to the unoptimized tool, and the ● shape indicates the execution time of the optimized tool. We can observe that the execution time of each program has been reduced, with the biggest improvement in the BreadthFirstSearch and the Substitution.
programs. Before the optimizations due to the complexity of the the BreadthFirstSearch program, it was impossible to execute the program, however it took 53 seconds to execute it on the optimized tool. The execution time of the Substitution program, was reduced by 112 seconds, from 132 to 20 seconds. Executing the unoptimized Tacny with both of these programs yielded a very large search space. Majority of the generated search space consisted of invalid Dafny programs. By preventing these branches from being generated we significantly reduced the execution time. Furthermore, running the DFS search strategy, in some cases reduced the number of calls made to Boogie, reducing the execution time even further. The other examples show a marginal improvement in performance ranging from 2 to 4 seconds. The difference in the performance increase is due to the complexity of the program. The other 4 programs are much simpler, thus their search space is much smaller. Therefore, reducing an already small search space does not yield high improvement in execution time.

### 6.1.2 Search Space and Memory analysis

The main factor impacting the performance of the original tool is the size of the generated search space. With the exception of the case when the correct solution is at the left most branch of the tree, the size of the search space highly impacts the performance of the tool. A smaller search space takes less time to generate, the Tacny tool will make fewer calls to Dafny and Boogie, and the tool will require less resources to run. In this section we will discuss how the optimisations improved the search space, thus reducing the memory usage. Graph \ref{fig:search-space-comparison} illustrates the search space comparison of both tools. The CoqArt-InsertionSort and Streams
Tacny Program | Unoptimized | Optimized | 
--- | --- | --- | 
BreadthFirstSearch | 6812.45 | 143.54 | 100% | 
Substitution | 2377.72 | 117.37 | 95% | 
NipkowKlein-Chapter 3 | 163.45 | 135.33 | 17% | 
InductionVsCoinduction | 99.95 | 92.04 | 8% | 
Streams | 67.83 | 108.31 | -60% | 
CoqArt-InsertionSort | 63.89 | 77.29 | -21% | 

*Average: 23%*

Table 6.1: Memory Usage Comparison

programs were not affected by the optimisation. The search space of the two programs is very small, and the first final solution generated verifies, therefore the optimisation did not affect these files. However, we can observe that as the complexity of the program grows, the higher the reduction in the search space. This can be explained by the fact that the more complicated program, the more arguments a lemma takes and the more local variables are in use, the more combinations the perm atomic statement will generate, of which majority will be type incorrect.

Similarly to execution time, the biggest reduction in the generated search space occurred in the BreadthFirstSearch and Substitution examples. The search space in the Substitution example was reduced from 0.58 million nodes to 132, the BreadthFirstSearch example search space was reduced from over 33 million to 266 nodes. This change proves our assumption that majority of the generated solutions were type incorrect.

By reducing the search space it is expected that the memory usage will also be improved, however, as showed in table 6.1 it’s not the case for all examples. The right most column in the table shows the percentile improvement in the memory usage. By reducing the total search space we reduce the amount memory a program uses. The search space for the CoqArt-InsertionSort and Streams programs was not changed by the optimisation, however the memory usage of the examples has increased by 20% and 60% respectively. The reason for this is that, the yield keyword turns a regular method into an enumerable method. When the enumerable is paused, the state of the enumerable is saved in memory. The Tacny tool was converted to be lazy, thus there are multiple methods between the search framework and the atomic statement resolver that use the yield keyword, this means when a single solution is yielded, all of the methods in between have their state saved in memory, thus causing a growth in memory usage. Furthermore, after a lemma has been generated, the residual objects instantiated during the resolution may not get cleaned up by the Garbage Collector. The garbage collection is initialized when one of the three conditions is satisfied [22]:

- The system is low on physical memory.
The memory used by the allocated objects surpass the accepted threshold.

- The Garbage Collector is called directly from the code.

However, because the system is not low on physical memory, the program ran for a short amount of time, and no direct calls to the collector are made in the code, the memory is not released until the process terminates.

### 6.2 Optimisation Analysis

The optimisations were implemented in a way which allows us to turn each of them on and off, therefore we can analyse how individual optimisation affected the performance of the tool. In this section we will analyse how each optimisation affected the search space, memory usage, and the execution time.

#### 6.2.1 Type Lookup

The examples in this section were executed using only the type lookup table, this will provide us insight on how many type-incorrect solutions were generated by the original implementation.

**Memory And Search Space**

Figure 6.3 depicts the generated search space comparison between the original Tacny version and version running only the type lookup. Due to the great difference in the generated search space, the graph is given in a logarithmic scale. Table 6.2 shows how type lookup affected
Table 6.2: Memory Usage Comparison

<table>
<thead>
<tr>
<th>Tacny Program</th>
<th>Unoptimized</th>
<th>Type Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreadthFirstSearch</td>
<td>6812.45</td>
<td>204.41</td>
</tr>
<tr>
<td>Substitution</td>
<td>2377.72</td>
<td>113.79</td>
</tr>
<tr>
<td>NipkowKlein-Chapter 3</td>
<td>163.45</td>
<td>145.74</td>
</tr>
<tr>
<td>InductionVsCoinduction</td>
<td>99.95</td>
<td>86.62</td>
</tr>
<tr>
<td>Streams</td>
<td>67.83</td>
<td>69.06</td>
</tr>
<tr>
<td>CoqArt-InsertionSort</td>
<td>63.89</td>
<td>64.4</td>
</tr>
</tbody>
</table>

The memory usage of the tool. The second column refers to the memory space of the original tool, the third column refers to the memory space of the updated tool, and the last column shows the percentile improvement. As expected, the first two programs were not affected by the optimisation, this is also reflected in the memory table, the difference in memory usage is very small 1% and 2%, thus it can be safely dismissed as general memory fluctuation. However, lookup table greatly reduced the search space of the more complicated examples, we can observe that the greater the original search space was, the greater impact type lookup had on it. The original search space for the InductionVsCoinduction example was 1523, after optimisation the search space was reduced to 75 nodes, the change is also reflected in the memory usage, it was reduced by 13%, from 99.95MB to 86.62MB. The NipkowKlein example generated 6404 nodes, the type lookup reduced the number to 185 nodes, and reduced the memory usage by 11%. The optimisation is best reflected in the Substitution and the BreadthFirstSearch examples. The search space of the first program was reduced from 0.58 million nodes to 232 nodes, this also reduced the memory usage by 95% from 2377MB to 113MB. The node count for the second program was reduced from 33 million to 409 nodes, as the result, the memory usage has been reduced by 97%, from 6812MB to 204MB. This shows that the majority of solutions generated by the perm atom were type incorrect, by eliminating them we greatly reduced the search space and improve the memory usage.

Execution Time

In this section we will discuss how the lookup table affected the execution time of the program. Graph 6.4 shows the execution time comparison of both versions of the tool. Type lookup has greatly improved the execution time of the programs that had large search space. The execution time of the Substitution program was reduced from 132 seconds to 22 seconds, and the BreadthFirstSearch program was successfully resolved in 70 seconds. The reduction in execution time was to be expected, by reducing the total search space we reduce the time
spent generating the nodes, and reduce the number of times Dafny has to be called to type-check the programs. However, the optimisation had decreasingly smaller affect on the simpler programs. The execution time of the NipkowKlein example was reduced from 50 seconds to 45 seconds, let us take a look why the improvement is only 5 seconds. The execution time breakdown shows that the unoptimized tool spent 42 seconds waiting for the verifier. The optimized tool has also spent 42 seconds waiting for the verifier, this means that the reduced search space did not affect the number of valid nodes, therefore, only time taken to generate the nodes was reduced. The execution time of the InductionVsCoinduction was reduced to 10 seconds, of which 7 seconds were spent waiting for the verifier. Finally, the time taken to resolve the Streams and CoqArt-InsertionSort examples was reduced by 2 seconds. We can see that the execution time was directly influenced by the search space reduction. Therefore, the smaller decrease in the search space the smaller improvement in execution time.

The aim of the type lookup table was to prevent the generation of type-incorrect solutions, thus reducing the search space, improving the memory usage and execution time. The test data comparison shows that, indeed, the majority of generated branches in the original tool were type-incorrect, by preventing them from being generated, we greatly reduced the memory usage and improved the execution time. The greater the original search space, the greater affect the optimisation had, this is due to complexity of the program. A more complicated program will generate a bigger search space than a simpler program, this is illustrated by the Substitution and the BreathFirstSearch examples. Both programs call the same tactics, however, Substitution example is not as rich as BreadthFirstSearch, therefore, it generates smaller search space.

6.2.2 Lazy Evaluation

Memory And Search Space

By disabling the lookup table and using the default BFS search strategy, we can observe how lazy evaluation works with high search space. The objective of lazy evaluation is to postpone solution generation as long as possible, thus reducing the generated search space. Figure 6.5 depicts the memory comparison between the original version of the tool, and the tool running only lazy evaluation. We can observe that the memory usage of the original tool was directly affected by the generated search space, where as the memory usage of the Dafny tool with lazy evaluation was uniform for all programs. Furthermore, from table 6.3 we can see that the total
CHAPTER 6. EVALUATION

Figure 6.4: Type Lookup Execution Time

Figure 6.5: Lazy Evaluation Memory Usage

Figure 6.6: Execution Time
Table 6.3: Search Space Comparison

<table>
<thead>
<tr>
<th>Tacny Program</th>
<th>Unoptimized</th>
<th>Lazy Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreadthFirstSearch</td>
<td>33.3m</td>
<td>3.6m</td>
</tr>
<tr>
<td>Substitution</td>
<td>580035</td>
<td>13562</td>
</tr>
<tr>
<td>NipkowKlein-Chapter 3</td>
<td>6404</td>
<td>4747</td>
</tr>
<tr>
<td>InductionVsCoinduction</td>
<td>1523</td>
<td>643</td>
</tr>
<tr>
<td>Streams</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>CoqArt-InsertionSort</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

generated search space has been substantially reduced. The original Substitution example used 2377 MB of memory and generated 0.58m nodes. Where as the lazy version used only 113 MB of memory and generated 13.5 thousand branches. A similar change has happened in the BreadthFirstSearch example, the original Tacny implementation used 6812 MB of memory and generated over 33m modes, in comparison, the updated version generated 3.6m branches and used 145 MB of memory. As mentioned in section 5.2 lazy evaluation generates the search tree one node at a time, once a single node is generated it can be resolved, verified and dismissed. From this, two properties follow:

Firstly, lazy evaluation will generate the smallest search space required to find the valid node. Because solutions are generated one at a time we can resolve and verify them immediately, thus once the valid solution is found, tactic resolution is terminated and no unnecessary nodes are generated. That is why the generated search space was reduced.

Secondly, with lazy evaluation, depending on the search strategy, only few solutions are kept in memory. For example, using the BFS search strategy only one layer of solutions are persistent in memory, furthermore, if a solution is final and it fails to verify we dismiss it. The DFS strategy uses a stack to hold the atomic statement resolvers, thus when the resolver reaches the final statement in the tactic body, the size of the stack will be equal to the number of statements inside the body. Once DFS strategy comes across a final solution, if it fails to verify it is dismissed, thus at most only a single solution is loaded. For this reason running the tool with lazy evaluation produces uniform memory usage despite varying complexity.

Both of the properties can be observed if we compare the BreadthFirstSearch sample running lazy evaluation and the performance of the original Substitution example. The Substitution example generated 0.58m nodes, 6 times less than the BreadthFirstSearch example. However, the Substitution example used 2377 MB of memory, whilst the lazy BreadthFirstSearch example only used 145 MB. This proves that lazy evaluation reduces the size of the search space, but more importantly it greatly reduces memory usage.
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>Tacny Program</th>
<th>DFS</th>
<th>BFS</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CoqArt-InsertionSort</em></td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><em>Streams</em></td>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td><em>InductionVsCoinduction</em></td>
<td>643</td>
<td>647</td>
<td>4</td>
</tr>
<tr>
<td><em>NipkowKlein-Chapter 3</em></td>
<td>4747</td>
<td>4283</td>
<td>464</td>
</tr>
<tr>
<td><em>Substitution</em></td>
<td>13562</td>
<td>16517</td>
<td>2955</td>
</tr>
<tr>
<td><em>BreadthFirstSearch</em></td>
<td>4116877</td>
<td>3479048</td>
<td>637829</td>
</tr>
</tbody>
</table>

Table 6.4: DFS and BFS Branch Count Comparison

Execution Time

Lazy evaluation has drastically improved the memory space, however as we will discuss in this section its’ impact on the execution time was marginal. Figure 6.6 refers to the execution time comparison of the two versions of the Tacny tool. The search space of the simpler programs, such as the *Streams* or *InductionArt-InsertionSort*, was only slightly affected by the optimisation, therefore the execution time has not been greatly affected. The biggest change is observable in the *BreadthFirstSearch* program, it was impossible to execute it with the original implementation, however, due to lazy optimisation, the tool no longer exhausts all machine resources, therefore it was possible to execute the tool. Tacny took 18252 seconds, or just over 5 hours, to generate and verify 3.6m branches. The breakdown of the execution time reveals, that the tool spent 11593 seconds, 3 hours and 10 minutes, just waiting for Dafny to resolve the solutions, and it waited 145 seconds for verification results. Thus it took 6513 seconds to generate the solutions one at a time. The execution time of the *Substitution* example was reduced by 52 seconds, from 132 to 70. The time breakdown shows that Dafny resolution took 24 seconds, and 18 seconds were spend waiting for the verification results. Therefore, Tacny took 28 seconds to generate 13848 solutions.

Lazy evaluation has greatly reduced the memory requirements for the tool. Resolving even the largest programs required just a fraction of the original memory. The optimisation also reduces the generated search space to a minimum, however, the impact on the execution time was minimal. This is because the majority of generated solutions were invalid.

6.2.3 Search Strategy Comparison

In this section we will compare the DFS and BFS search strategies. The search framework greatly depends on lazy evaluation, thus it will not be turned off. The table lookup has been disabled for these tests to provide a better sense of scale.
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>Tacny Program</th>
<th>Lemma</th>
<th>DFS</th>
<th>BFS</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreadthFirstSearch</td>
<td>Lemma_IsPath_Closure</td>
<td>688717</td>
<td>1667</td>
<td>687050</td>
</tr>
<tr>
<td></td>
<td>Lemma_IsPath_R</td>
<td>3428160</td>
<td>3477381</td>
<td>49221</td>
</tr>
<tr>
<td>NipkowKlein-Chapter 3</td>
<td>AsimpConst</td>
<td>1329</td>
<td>1399</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>BsimpCorrect</td>
<td>3418</td>
<td>2884</td>
<td>534</td>
</tr>
</tbody>
</table>

Table 6.5: Search Space Breakdown

Search Space Analysis

Because the strategies are using lazy evaluation the difference in memory usage is very small, therefore we will only analyse the difference in generated search space. Table 6.4 refers to the node count generated by each search strategy. The right most column holds the difference in the generated search space, underlined results indicate which search strategy generated more solutions. We can see that DFS strategy outperformed BFS strategy in the Induction and the Substitution examples, generating 4 and 2955 fewer nodes respectively. However, it generated 464 nodes more for the NipkowKlein and 0.63m nodes more for the BreadthFirstSearch example. Earlier in the paper, we speculated that DFS strategy will not be outperformed by BFS strategy, which seems not to be the case, in the remainder of this section we will analyse what caused this.

Table 6.5 depicts the search space breakdown for the two examples, the underlined number refers to the strategy which generated larger search space, and the right most column shows the difference in the search spaces.

```
lemma Lemma_IsPath_Closure(source: Vertex, dest: Vertex, p: List<Vertex>, AllVertices: set<Vertex>)
{
    match p {
        case Nil ⇒
        case Cons(v, tail) ⇒
            Lemma_IsPath_Closure(source, v, tail, AllVertices);
    }
}
```

Code 6.1: Original Lemma_IsPath_Closure lemma body

Let’s look at the Lemma_IsPath_Closure body again[1] seen in figure 6.1. The valid solution for the program has to contain a single recursive lemma application for the Cons constructor. To generate the lemma, we apply the CasePerm tactic, listed in figure 6.2. The tactic calls the perm atom inside a while loop to generate up to two lemma applications.

[1]Detail analysis of the lemma is given in section 3.1
**tactic** CasePerm(b: Element)
{
    cases b {
        tvar v := merge(variables(), params());
        tvar l | l in lemmas();
        tvar i := 0;
        while (i < 2)
        {
            perm(l, v);
            i := i + 1;
        }
    }
}

Figure 6.7: DFS Strategy Generation Order

The reason why the DFS strategy generates a large search space in comparison to BFS, lies in how the while statement is evaluated. To understand how the statement is evaluated using DFS strategy, consider the partial search tree given in figure 6.7. Assume there are two variables \(a, b\) in the context, and \(tvar l\) is some lemma \(lem\). The numbers near the nodes indicate generation order, and the final solution is marked as \([\text{valid}]\). To reach the valid node \#4, DFS strategy has to generate node \#1 and all of its’ children. Where as the BFS strategy would only have to generate node \#1 before generating the valid node. This extra work is what caused the great difference in search space for the Lemma_IsPath_Closure example. That is the same reason why the BsimpCorrect lemma took 534 solutions more using the DFS strategy.

On the other hand, for the Lemma_IsPath_R lemma DFS strategy generated 49221 nodes less than BFS strategy. This is because, the valid solution was located at the very bottom of the tree, DFS strategy can reach such solutions without fully exploring the width of the tree, where as the BFS strategy has to fully explore each layer before moving down to the next. To conclude, if the valid solution is at the very bottom on the search tree, DFS strategy will outperform the BFS strategy. However, if the valid solutions does not require reaching the bottom of the search tree, for example, to generate the final solution we only need a single while loop iteration, BFS
strategy is guaranteed to generate a smaller search space.

**Execution Time**

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Tactic</th>
<th>#Nodes</th>
<th>Exe Time</th>
<th>Boogie Wait</th>
<th>Dafny Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>CasePerm</td>
<td>68871</td>
<td>1201</td>
<td>102</td>
<td>752</td>
</tr>
<tr>
<td>DFS</td>
<td>CasePerm2</td>
<td>342816</td>
<td>17051</td>
<td>43</td>
<td>10841</td>
</tr>
<tr>
<td>BFS</td>
<td>CasePerm</td>
<td>1667</td>
<td>15</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>BFS</td>
<td>CasePerm2</td>
<td>347738</td>
<td>18081</td>
<td>40</td>
<td>10845</td>
</tr>
</tbody>
</table>

Table 6.6: *BreadthFirstSearch* Execution Time Analysis

In the previous section we showed that the only significant difference in the search space is seen in the *BreadthFirstSearch* example, this is also reflected in the execution time, therefore, in this section we will only analyse how this particular example was affected. Table 6.6 depicts the execution time breakdown for the program, *#Nodes* column refers to the size of the search space, *Exe Time* refers to the total time taken to resolve the tactic, and the *Boogie* and *Dafny Wait* columns show the time it took to verify and resolve the solutions.

The affect the search space has on the execution time is clearly showed by the *CasePerm* tactic. The DFS strategy generated 0.68m nodes in 1201 seconds, of which 752 seconds were spent waiting for Dafny and 102 seconds were spent waiting for verification results. On the other hand, BFS strategy took 15 seconds to generate 1667 solutions, of which 11 seconds were spent waiting for the verifier, and 4 seconds were spent waiting for Dafny. This further proves our point made earlier in the paper, that the greater the search space the more time will be spent waiting for Dafny to verify the programs.

The DFS strategy was faster to resolve the *CasePerm2* tactic. It took 17051 seconds to find the valid solution, of which 10841 seconds were spent waiting for Dafny resolver and 43 seconds were spent for verification results. BFS example took 18081 seconds in total to resolve the program, which is 1030 seconds more than DFS strategy. In addition, it took 11745 seconds for the Dafny verifier to type-check each solution, however, this is expected as the BFS strategy generated 49221 more nodes than the DFS. The verification time was similar to that of DFS, 40 seconds, this suggests that the number of valid solutions for both strategies was the same.
Chapter 7

Conclusion and Future Work

The aim of the project was to identify and provide optimisation solutions for performance bottlenecks in the Tacny tool. We identified four issues and implemented three:

Firstly, the generated method (including lemma and function) applications were type-incorrect, that is, incorrect type arguments were passed. Therefore, majority of the solutions were failing to type-check during resolution. This issue heavily affected the time and space complexity of the program, in cases, due to their complexity, rendering the tactics unresolvable. To overcome this issue, we implemented a type lookup table. The optimisation takes the Dafny program with tactics and puts it through the Dafny resolver, where the variable types are resolved. Using the resolved types the lookup table is filled, which is then used to generate type correct permutations. This optimisation greatly reduced the number of solutions that are generated, thus reducing search space and improving the execution time.

The second bottleneck was eager solution generation. The issue was that the tool would first generate the entire search space and only then search for the solution. However, in some cases the whole search space will exceed the available memory, thus preventing the program from running smoothly. Furthermore, tests show that the solution tree never requires full exploration to find the final solution. However, the unexplored were still generated, and consume valuable memory space and execution time. To generate the tree one solution at a time the evaluation strategy was changed from eager to lazy. With lazy evaluation we generate the smallest search space required to find the correct solution. Secondly, the most important property of lazy evaluation is its’ ability to keep memory usage uniform. That is, depending on the search strategy in use, only a small number of solutions are loaded in memory. A program which generated over 50 thousand solutions used just slightly more memory than a program which
generated 600 branches.

The final implemented optimisation was a framework, which supports multiple search strategies. We speculated, that using DFS will offer improved execution time than BFS. We implemented a framework which holds the search strategies and offers a single point entry to execute the strategies. The framework supports two strategies, DFS and BFS, the test showed that in some cases DFS offers reduced execution time, however, certain tactics will be resolved faster using the BFS strategy.

In the project we only explored DFS and BFS strategies, however there are many more strategies that may increase the execution speed. One such strategy is the Best-First-Search, which would use the total number of siblings a node may have as the heuristic. It is possible to calculate the total number of nodes a single tactic statement can generate. For example, we know that expression such as `lemmas()`, `variables()` etc. will generate a single branch. Furthermore, based on the program context, we can calculate how many branches an atom statement may generate, e.g. calling the `tvar lm : | lm in lemmas();` atom in a program, that contains 3 lemmas, will generate a search space of 3 nodes, a node for each lemma. Using the search space a branch may generate as the heuristic will ensure that we first explore least branching nodes, thus possibly exploring less search space before finding the valid solution.

Some of the designed features were not implemented. The initial parallel tactic resolutions showed that even though it is possible to resolve multiple tactic applications at the same time, Boogie static library is not safe, thus therefore, generated solutions have to be verified sequentially. More research has to be carried out on how to verify multiple solutions at the same time. Due to time constraints the `set_strategy` expression, which is used to set a search strategy from inside the tactic body, was not implemented.

The optimisations have significantly improved the performance of the tool, and revealed opportunities for further work. To performance of the tool was tested using a single type of tactic which uses a brute force method to generate method applications. To further measure measure the performance of the tool we need to develop a richer tactic with control structures and nested `cases` application. The tool should also be tested with different tactics, such as method variant and loop invariant generation.

The programs used to test the tool use the same tactic, which had to be copy-pasted multiple times. If any change had to be made to the tactic, it had to be repeated for each program. We propose an extension to the tool, which will allow users to reuse a single tactic across mul-
multiple programs. The implementation would reuse the existing Dafny import keyword, `import <file_name.tactic>`. The statement is placed at the top of the file, before tactic evaluation is started, the tactics are loaded from the file, to the program context. To ensure there is no confusion between Dafny files and tactic files, the tactic files use `.tactic` extension.

The `BreadthFirstSearch.dfy` program, even with lazy evaluation, to a long time to execute, thus it would be useful to specify a tactic resolution timeout. If a solution is not found in the specified time, the resolution would be terminated. This extension, would be particularly useful for the future Tacny integration with Dafny, which was mentioned in the original Tacny paper [13]. The timeout mechanism would be crucial to ensure the IDE is not left unresponsive.
Bibliography


