An Investigation of Combined Domain Bi-Abductive Inference

Christopher William Curry

A thesis submitted in partial fulfilment of the requirements of Teesside University for the degree of Doctor of Philosophy

June 2022

Acknowledgements

I would first like to thank all the members of my supervisory team throughout the duration of my study, Dr. Chunyan Mu, Prof. Shengchao Qin, Dr. Mengda He and Dr. Quang Loc Le. Your support, guidance and assistance has been invaluable throughout my research, and I could not have completed this project and thesis without your knowledge and patience.

I would also like to thank Teesside University, for giving me the opportunity to undertake this PhD and for all the support and training offered. I must also thank the staff of the School of Computing, Engineering and Digital Technologies in particular for their support.

Last, but by no means least, I must thank all my friends and family for their support and faith in me throughout the duration of my PhD. I could not have done it without you.

Abstract

Bi-abductive inference is a well-established and effective technique for the enhancement of separation logic entailment provers. Capable of identifying and correcting deficiencies in an entailment through the inference of frames and anti-frames – representing the missing and extra information from the entailment – bi-abduction has seen a number of successful applications in verification systems.

While effective, bi-abductive inference has a number of notable limitations, with one of the most significant being a restriction to the inference of spatial information only. Though uncommon, non-spatial properties may have an impact on the memory-safety properties bi-abductive techniques are typically applied to establish, making the inference of such constraints a notable improvement. Additionally, support for non-spatial properties such as ordering and contents would also open the possibility of applying bi-abductive inference to establish correctness properties of programs as well.

Though a small number of bi-abductive techniques with support for the combined domain are known, these systems are dependant upon additional analysis phases, introducing overheads and potentially degrading the results produced by the system as a result.

In order to more fully investigate the feasibility and potential benefits such a system could bring, a pair of novel single-step bi-abductive inference systems operating in the combined spatial and pure domain are developed and presented: a initial system for potentially ordered singly-linked lists and trees, and a second generalised system for a more complex combined domain with user-defined predicates and support for bag, arithmetic and ordering properties. These systems were subsequently implemented as automated bi-abductive entailment provers based upon the Cyclist framework and tested over several divisions taken from the SL-COMP competition. While these implementations would not match the effectiveness of existing systems, the overall results still indicate the approach has a great deal of promise, with the implementations able to resolve the majority of examples in modest times with reasonable inferred corrections, where needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems have become ubiquitous in modern life. Where software used to be a rarity, such systems now have applications ranging from use in controlling and monitoring industrial processes, in supporting human operators of vehicles such as cars and aircraft, and more novel applications in complex lightweight systems such as the Internet of Things. While such systems can provide significant benefits, this increasing prevalence and reliance on software systems has introduced even greater costs should those systems fail.

While software failures are harmless and tolerated in some settings, such as the annoyance of a mobile app crashing during use, software faults can be disastrous in the worst cases, with costs ranging from monetary loss in the millions to catastrophic loss of life. There are numerous high-profile failures in the literature, such as the Therac-25 radiotherapy system, whose faulty software caused patients to receive significantly higher doses of radiation than was expected or safe [Leveson and Turner, 1993], the Arianne-5 rocket telemetry system, whose failure directly led to the complete destruction of the rocket and its payload [Dowson, 1997], or the troubled deployment of the London Ambulance service dispatch system [Finkelstein and Dowell, 1996]. More recently, attention was drawn to the design of avionics systems following two instances where the Boeing 737 Max crashed due to issues with the control software [Cusumano, 2020]. Each of these cases, though distinct in their causes and impacts, serve to highlight the importance of

high-quality software, and the costs that may be incurred should that software fail.

Formal methods is a leading approach to addressing this need for high-quality software. An approach proposed in the early days of computer science by pioneers of the time, formal methods aims to apply mathematical proof techniques to programs in order to reason about their operation and behaviour. The advantages of this approach are obvious: formal methods allows for a level of confidence that far exceeds more typical approaches such as program testing, able to state with certainty that a program will behave as it should, identifying faults before they can cause failures. Though many variations of formal methods are known, the specific approach of most relevance here is that of program verification.

1.1 Background

Program verification is an analysis technique that aims to formally prove a program exhibits some key properties, with respect to some known specification. A broad and highly active field, with numerous different approaches and tools, program verification has been applied to the analysis of correctness, termination and timing properties of programs. While each of these properties are of great use, the property that is the primary focus of this work is the *memory safety* of a program.

Memory safety is a property of programs that ensures that operations such as reading from or writing to program memory - sometimes called the *heap* - are valid, correct, and defined, with respect to the language design. All programs interact with memory during the course of their execution, though the degree of control a programmer has over those interactions is determined by the programming language itself. Languages such as C and C++ allow for the direct manipulation of program memory, utilising explicit pointers to access specific addresses, where others, such as Java, abstract away much of the functionality through the use of garbage collectors and other mechanisms. Regardless of

the language used, however, the identification of memory faults is an important task; "unsafe" accesses, such as attempting to access `null` or invalid memory locations, amongst other dangerous operations, are possible in all languages and are a common source of faults in software. Such faults have long been recognised as a significant threat and continue to be today; several of the entries in the *2020 CWE Top 25 Most Dangerous Software Weaknesses* [CVE, 2020] are vulnerabilities enabled or caused by violations of memory safety properties.

Unfortunately, the analysis of program memory access is often more difficult than it would initially seem. Variables targeting locations in heap exhibit a behaviour known as *Shared Mutable Memory*: all locations in memory are both accessible from multiple points in the program (Shared) and modifiable by any of those access operations (Mutable). This property can introduce difficulties when attempting to reason about the values or properties of a value in memory, as there is no longer any guarantee that a given block of program memory is in the same state as it was when it was last updated. This issue is further exacerbated by the program behaviour known as "aliasing": the ability to create multiple distinct references to the same location in memory. While this behaviour is perfectly normal in software development, this aliasing means that a seemingly harmless update applied to one variable may in fact be a potentially dangerous operation on a critical value in memory. These difficulties make the analysis of programs that manipulate the heap a complex task, with programs that take advantage of heap-based data structures such as singly or doubly-linked lists particularly affected, as the integrity of the data structure itself can be undermined by an errant memory operation.

A number of techniques were developed to address this problem, ranging from more typical analyses over the program code to the development of so-called *safety-critical* language that enforced specific safety constraints over such operations. Though effective, these approaches were not always ideal. Early approaches towards memory safety verification under shared mutable memory conditions were often problematic, typically scaling poorly or lacking precision [Yang et al., 2008], preventing the techniques from being effectively applied to

larger programs and blocks of code. Safety-critical languages such as MISRA-C or Rust [Balasubramanian et al., 2017] are powerful tools, but required programmers to retrain in order to use them, and could not address potential faults in existing codebases, heavily limiting the application of such languages.

One area which showed a great deal of promise was Shape Analysis. Shape analysis is an analysis technique which focusses on the verification of a programs memory safety from the perspective of its "shape", the general layout, connectivity and structure of the assigned blocks of memory [Sagiv et al., 1998]. Such analyses could be applied to existing code and were able to reason about a more complex set of properties, such as relational properties between the various blocks of assigned memory. Though the initial attempts at shape analysis proved to be reasonably effective, able to identify memory faults in a range of programs, these attempts were not without issue. Scalability remained a significant concern, with early techniques required to maintain records of all areas of the program heap in order to prevent aliased variables from being updated without notice.

A key advancement was made with the development of separation logic. Developed by Reynolds, O'Hearn and Istiaq, based on the early works of Burnstall [Ishtiaq and O'Hearn, 2001, O'Hearn et al., 2001, Reynolds, 2002], separation logic is an extension of Hoare Logic [Hoare, 1969] in which the program heap may be logically partitioned into discrete sub-heaps, or heaplets, *separate* from others in the formula. The core concept underpinning this new logic was the principle of *local reasoning*: the understanding that a program can only affect the parts of the heap it accesses during the course of its execution, with all other locations unaffected. This principle allows for analyses over programs affecting the heap to track only the subset of the heap that is accessed by the program, dramatically reducing the scale of the problem. This advance was used to underpin the development of a new generation of effective and scalable techniques, including tools such as Smallfoot [Berdine et al., 2005b], SpaceInvader [Yang et al., 2008], and others [Lee et al., 2005, Distefano et al., 2006a].

This ability to focus the analysis on only the areas of memory affected by some program further enabled compositional analyses, an analysis approach in which

the final output is generated by breaking down the larger analysis problem into a series of smaller, more manageable problems and combining the solutions. This compositional approach is highly beneficial, enabling an incremental approach to analyses, graceful failure in analysis results, and parallelisation of the analysis process, as well as enabling greater scalability in tools [Calcagno et al., 2009].

Bi-Abductive inference is one mechanism by which compositional analysis based on separation logic can be supported. Essentially a combination of the frame inference and abductive inference techniques, bi-abduction produces corrections to entailments that would otherwise fail by identifying both missing and extra fragments of program state. These corrections, if introduced back into the entailment, will address the deficiencies present in the original and ensure that the new version of the entailment will hold. These inferred fragments serve two purposes for compositional analyses: the extra state information, or the frame, details aspects of program state that are not affected by the procedure and should persist beyond the return, being propagated forwards into the rest of the program, eventually forming part of the postcondition. The missing information, or the anti-frame, outlines deficiencies in the program state at the point of procedure call, or more specifically, aspects of program state or constraints that are necessary for correct or safe execution that are not enforced by the current form of the program state and must be part of the program precondition as a result.

Bi-abductive inference is a reasonably adaptable tool, with a number of benefits and possible applications. In the original implementation of the Infer static analysis tool [Calcagno and Distefano, 2011], bi-abductive inference was used to identify potential memory faults, such as use-after-free and null references, in addition to specification inference. Applications for program repair are also known [Bach et al., 2016], and more recently, the Gillian tool applies bi-abduction for bug identification and test-case generation [Fragoso Santos et al., 2020, Maksimović et al., 2021].

1.2 Motivation

Despite the effectiveness of bi-abductive inference, there are still a small number of key limitations to the technique. One of the most notable is that very few known bi-abductive inference techniques have the capability to process so-called *pure* information, such as size, ordering or contents (bag) properties over data structures. Indeed, most implementations of the technique are unable to apply bi-abduction to pure constraints at all, relying on an additional analysis mechanism to process such properties, as in Infer [Calcagno et al., 2011]. Additionally, pure constraints, though rarely, may have a direct impact on the memory safety of the program, as in a program which accesses a specific number of nodes in a data structure [Magill et al., 2008], meaning that the analysis of such properties may be necessary to establish memory safety. Aside from the potential necessity of such analyses, another benefit to the introduction of these capabilities is that the inference of such properties may allow bi-abductive techniques to be applied to verify correctness properties, asserting that methods introduce or preserve specific constraints over the structure. As a result, bi-abductive inference techniques with extended support for pure properties could prove to be more effective than their contemporaries, potentially inferring more expressive specifications, identifying a wider range of program faults or deficiencies, and enabling more thorough analyses.

While a small number of bi-abductive techniques with support for the combined domain are known, they typically have limitations around the inference of pure constraints. The earliest known instance of bi-abductive inference with support for shape and pure constraints is that of Trinh *et. al.* [Trinh et al., 2013]. In this work, the problem of pure inference is resolved by the use of a secondary analysis stage, gathering relational assumptions and refining them to the final constraint set. However, this approach requires additional variables to be introduced into the initial shape analysis results, either done as part of the initial analysis or introduced based on guidance from predicate definitions. Though reasonably effective, this approach has completely separated the shape and pure domains, introducing a potential source of imprecision into the analysis.

One of the other notable implementations of combined-domain bi-abductive inference, the HIP/SLEEK verification system makes use of bi-abduction as part of its specification inference [Qin et al., 2017]. Applied during a forwards analysis, the HIP/SLEEK system gathers constraints over a program iteratively, updating and refining the system through abstraction, until a fixpoint is reached. While the tool is effective overall, the bi-abduction used throughout is insufficient to handle pure constraints in a single pass, instead relying on a supporting secondary abductive inference mechanism to aid and refine the inference process. Again, while the system is effective, the shape and pure domains are separate, and much of the design of the bi-abduction system relies on the production and discharge of additional entailments that ensure the validity of specific operations, incurring a reasonable efficiency cost and overhead.

As can be seen, there does not appear to be a "true" combined domain bi-abductive inference technique. The approaches that have been discussed above, while effective, have not fully integrated the shape and pure domains and are reliant upon multiple phases in order to infer the combined properties of the entailment. This additional analysis step is a possible source of inefficiency, as well as a possible weakening of the precision of the technique. Addressing these limitations may enable a more effective approach to bi-abductive inference than is currently available, enhancing any system that aims to utilise the technique.

1.3 Objectives

Based upon the limitations discussed in the previous section, this project aims to explore and evaluate the potential benefits of a novel bi-abductive inference technique with support for both the shape and pure domains which can process both domains *simultaneously*. This project aims to answer the research question:

How can the full integration of inference for pure constraints to bi-abductive inference techniques improve the expressiveness and capabilities of the approach?

This project aims to address this question through the development and subsequent experimental evaluation of a novel bi-abductive inference system with these fully-integrated capabilities. Both the development and subsequent results gathered from the eventual system is anticipated to provide key information about the potential costs and weaknesses of the existing split approach, as well as the potential advantages and limitations of the fully-integrated approach. This project can be broken down into three key objectives:

A Fully Integrated Combined Domain Bi-Abduction Technique

As discussed in the motivation and background sections, bi-abductive inference is an effective technique with a wide range of applications, but with few instances of the techniques featuring support for the more general pure domains. Those few instances that do typically feature a separation between the shape and pure domains, which is suspected to be a point of inefficiency in the system. In order to investigate this potential impact, as well as examine the feasibility of a bi-abductive inference system in which both domains are processed together, a fully integrated combined domain bi-abductive inference system will be designed and built. This system will be subsequently validated through by-hand experimentation over example entailments with known results in order to ensure the system is both sound and valid.

Automated Combined-Domain Bi-Abductive Tool

Once the system designed to address the previous objective has been completed, the next key objective will be to automate that system. While most systems developed in the literature are applicable by-hand, such attempts are often complex and time consuming. Instead, the majority of such techniques are utilised by means of an automated implementation, usually developed alongside the technique. Indeed, the inclusion of such automated tools is commonplace in the field. As a result of these aspects, as well as the potential benefits to the evaluation of the eventual system, an automated prototype implementation will be developed

for the bi-abductive inference system.

Experimental Evaluation

A key aspect of the eventual bi-abductive system planned as part of this project is how well the tool performs, especially against other similar tools in the area. In order to gain a greater understanding of the strengths, weaknesses and performance of the system - the automated system in particular - an experimental evaluation against a selection of similar tools in the area will be undertaken, aiming to gain concrete quantitative data as to how this novel approach will compare to more established ones of similar capabilities. This evaluation will be performed over a robust and trusted set of examples sourced from the wider field, aiming to ensure the results are reliable. The resulting quantitative data will be subsequently analysed in order to identify both strengths and weaknesses of the system, with such limitations providing valuable information as to the potential benefits and costs of the fully-integrated approach.

The completion of all the above objectives should provide some valuable insight into the effects of separating the processing of properties in tools targeting otherwise combined domains. However, these objectives will need to address and overcome several key challenges over the course of this project.

1.4 Challenges

During the course of this project, there are a small number of key challenges that must be addressed and overcome in order to achieve the objectives presented in the previous section (Section 1.3). Though some of these challenges already have mitigations and techniques to address them in the literature, they remain a key consideration.

Infinite Data Structures

A key aspect of many data structures is that their definitions are recursive, indicating that these predicates may be expanded infinitely. As a result, basic operations such as unfolding may cause the system to reach an impasse, repeating some basic operation without meaningfully progressing, preventing the system from terminating. As a result, it is imperative that the resulting systems be designed in a manner which is not vulnerable to this issue, making termination of the eventual bi-abductive system a key - if difficult - property.

Inference in Combined Domains

Though existing systems have demonstrated bi-abductive inference in the combined domain is possible [Trinh et al., 2013, Qin et al., 2017], each of these approaches has relied on secondary analysis stages in order to succeed. In order to achieve a single-step bi-abductive technique, the techniques developed in this project must be able to identify pure constraints without some prior shape analysis or some supporting abstraction, a capability that has not been explored, to the author's knowledge.

System Automation

The automation of systems that make use of separation logic is not a straightforward task. While the logic is constructed as an extension to Hoare Logic, it is still fundamentally a non-standard logic, and as such, requires specialised symbolic execution engines and proof engines in order to fully automate and examine the formulae produced. Bi-abduction is typically constructed in such a manner that the manipulation of such formulae is a core aspect of the technique, causing bi-abduction to inherit much of this complexity. Though these specialised engines have been developed since Separation Logic was first introduced and adopted, there still remains complexities in deciding which approach to adopt and what mitigations or considerations need to be made to make effective use of them.

There may also be some additional complexity to automation depending on the approach taken by the bi-abduction mechanism itself. Should the eventual approach relies upon an approach in which there may be many possible advances from a given program state, only some of which may be valid, there will most likely be a need to develop a proof search algorithm, one capable of either looking beyond a single application or of backtracking to an earlier state before taking a different path. These algorithms must be designed carefully such that dead-ends are identified quickly and effectively, and that the new paths are explored without a heavy cost to overall performance.

1.5 Contributions

With the challenges encountered during this project overcome, and the targeted objectives achieved, the key contribution of this work is presented as follows: the primary contribution presented by this work are novel one-phase bi-abductive inference techniques for the combined shape and pure domain, with corresponding prototype implementations. The experimental results gathered throughout the experimentation undertaken during this project have demonstrated that a single-phase combined domain bi-abductive technique is not only possible, but effective over a wide range of examples, though with some limitations in the current form. The performance overall, while not on par with other leading systems [Le et al., 2018, Ta et al., 2017, Qin et al., 2017], is still comparable, despite the more complex problem solved by the tools we compared against. Each of the contributions are explained in greater depth below.

Novel Bi-Abduction Inference Systems

A deviation from the initial aim of the project, two separate bi-abductive systems were developed during the course of this research. These two systems, one an initial proof of concept, and one a more refined generalised variant, are based around the use of a set of inductive rules, following in the style of Smallfoot

[Berdine et al., 2005b], and a specialised search algorithm used to guide the construction of a proof tree. These inductive rules act as both proof rules and inference rules, identifying and correcting shortcomings in the entailment during the process of proving its validity.

The first system, detailed in Chapter 4, was a novel proof of concept system designed to show the feasibility of the single-phase approach, operated in a restricted shape and order domain for potentially sorted singly-linked lists and trees. While initially intended to be further refined to achieve a more capable tool, the initial capabilities over the restricted fragment it operated over were of sufficient interest that the system was preserved. The system is sound, though not complete, and is also shown to terminate.

This second system was a novel combined-domain bi-abductive entailment prover, capable of handling entailments with near-arbitrary user-defined predicates with ordering, arithmetic and bag properties, a level of expressiveness that can be seen as matching some of the leading approaches [Qin et al., 2017]. This system, described in Chapter 5, was an extended and refined version of the initial proof of concept technique, and followed the same style and approach. While some of the work done in the creation of the first system could be transferred to the second, there were still a number of challenges and considerations that needed to be addressed to achieve this goal, with a number of additional rules and refinements developed as part of the creation of this system.

Both the systems designed over the course of this project were also automated as an instantiation of a modified version of the Cyclist library [Brotherston et al., 2011, Gorogiannis, nd]. These tools, written in OCaml, act as a compound entailment prover and bi-abductive inference tool and are able to support a range of examples. The implementations do not require any other programs in order to operate effectively at present. The implementations were evaluated over a wide range of example entailments, primarily sourced from the benchmark sets of the SL-COMP competition [Sighireanu et al., 2019b, Sighireanu et al., 2019a]. Though the set of examples supported by each tool varied, the results showed promise, with the vast majority of the examples being solved within 30s for each

tool. These results were also compared against a range of other similar tools from the literature, as well as each other.

1.6 Organisation

The remainder of the thesis is arranged into a number of chapters. Chapter 2 will act as a primer for several key concepts underpinning our work and will provide some key definitions used in subsequent chapters. Chapter 3 will present and discuss a number of key works related to our contributions and the field. Chapter 4 focusses on the development and design of the initial exploration of Combined Domain Bi-abduction, which is later used to guide the development of the generalised system as described in Chapter 5. The implementations of the systems developed during this project are presented and described in Chapter 6. Chapter 7 presents and discusses the experimental results gathered around the system implementations before we summarise and conclude the thesis in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we will present the operational details of several key techniques intrinsic to bi-abductive inference, as well as presenting the core language and semantics utilised in our contributions. The chapter will begin by discussing the core principles of Memory Safety (Section 2.1) before continuing to discuss Hoare Logic (Section 2.2) and Separation Logic (Section 2.3). The chapter will then present an overview of Bi-abductive Inference (Section 2.4) before concluding by presenting and discussing the notation and language fragments utilised throughout this document and our work (Section 2.5).

2.1 Memory Safety

Memory safety is property that attempts to establish a targeted programs interactions with memory as predictable and safe, usually following language specifications. Unlike more general correctness properties, memory safety does not typically consider whether a program achieves its intended purpose, simply focussing on the interactions of that program with memory. While violations of memory safety are frequently root causes of program faults and crashes, these flaws can also be more significant, introducing potentially catastrophic security issues into an otherwise fully functional piece of software [CVE, 2020]. As a result, identifying potential memory-unsafe operations is an important step in ensuring

the quality and effectiveness of software.

A number of techniques and approaches have been developed to address this need to identify or prevent memory faults, with the primary focus of this work - bi-abductive inference - being one example. While bi-abductive inference has a number of potential applications, the technique is most closely linked to the establishment of memory safety properties of programs. However, as will be discussed in Section 2.4, bi-abductive inference is not typically sufficient to establish memory safety alone, and is instead used to support other analysis techniques through the identification of deficiencies in program states that may indicate or lead to memory safety-violations.

In order to more effectively place this work within the wider field of program analysis, some time will be spent on introducing and explaining a number of key memory-safety principles, including core terms and concepts, as well as the source and impact of a small number of common memory faults.

2.1.1 Program Memory

While all programs interact with system memory - alternatively referred to as the program heap - in some way, the nature of that interaction depends on the developer and the language used. In languages such as the C family, the language permits total control over the manipulation of memory: a developer can assign and free specific locations of memory, and in some cases, use addresses produced by arithmetic operations to access the program heap. In other languages, such as Java, interactions with memory are indirect and constrained: a developer has little to no control over the contents, assignment or release of specific memory locations, operating instead through language constructs that obfuscate the concrete interactions, such as constructors and garbage collectors.

Regardless of the mechanism used by a language, it is a possibility that the developer may include a block of code - either a contiguous block or a complex sequence of disparate commands - that can cause an unsafe interaction with pro-

gram memory[1]. In such cases, a program is no longer considered memory safe; there exists the possibility that this fault is realised during execution and the program will attempt to read from or write to a potentially hazardous location in memory. While this interaction may simply cause the program to crash, execution may in some cases continue beyond that fault without visible issue, with unpredictable results. Such erroneous states can cause the production of incorrect output, or in the worst cases, induce security risks that may be exploited by malicious actors [Szekeres et al., 2013]. Memory faults are a common cause of security patches as a result of these impacts [Microsoft, 2019].

Memory Faults

As memory safety can be argued to cover a wide range of potential behaviours, a large number of potential faults can be considered to fall within that category. For the purposes of this work, the focus will be primarily on pointer-based memory faults, though some sources consider buffer-related faults to also fall under the category of memory safety [CVE, 2006].

The category of pointer-based faults include some of the most commonly encountered and significant issues, with `null`-pointer dereferencing being one such example. `null`pointer dereferencing is a memory fault in which a program attempts to read or write data from an invalid or `null` location in program memory. While this fault typically causes the program to crash due the invalid access, the erroneous operation may allow for invalid read or write operations to occur, potentially corrupting memory or allowing code execution attacks. `null`pointer dereferences are significant enough that the CWE Top 20 includes these faults in both the 2020 and 2021 versions of the list [CVE, 2020, CVE, 2021].

Another well-known memory fault is the *use-after-free* fault, in which a pro-

[1]While languages that utilise garbage collectors and similar mechanisms are typically considered memory-safe languages, it is still possible for some memory faults such as `null`-pointer dereferences to occur. These faults are typically caught via mechanisms such as exception handling, but analysis approaches may still be used to detect the root cause in the code.

gram attempts to read a memory location which has been "freed": an operation which notifies the memory management system that the location is no longer in use by the program and can be reassigned freely. This causes the pointer in question to become *dangling*: a previously valid pointer that now points to an invalid location. When attempting to use this dangling pointer after the free operation, there is no longer a guarantee about the data stored in that location; the location may have been validly updated by another process on the system. As a result of this, use-after-free faults may read "corrupted" data, triggering undefined behaviours or program crashes or leak the data in that location though its access. The fault may also trigger a write operation to that location, corrupting the data and triggering faults elsewhere in the program, or potentially in other processes. Use-after-free faults are another entry on the CWE Top 20 for both 2020 and 2021 [CVE, 2020, CVE, 2021].

A similar fault is the *double-free* fault, in which the program will free a location in memory multiple times, without reclaiming ownership over that block. As with the use-after-free fault, this operation is hazardous due to the fact that the location has been released for use elsewhere on the system, allowing for this free operation to potentially corrupt memory in use by another process, or elsewhere within the program. In the worst cases, this corruption may subsequently lead to a malicious actor gaining control of the data stored in the location, leading to a number of potential attacks.

A small number of additional faults do exist, but ultimately follow similar patterns as those outlined above: read or write operations targeting invalid locations in memory. For a more detailed overview of potential causes and effects of memory faults, [Szekeres et al., 2013] provides a detailed summary.

Regardless of the source of the memory fault, the effects of an unsafe memory operation can be significant, both to the execution and the system overall. As a result of the potential impact of these faults, a number of techniques have been developed to address or mitigate such faults when they arise, or to prevent them from occurring altogether.

2.1.2 Memory Protections

While ensuring a program is memory safe is clearly an important task, there are a number of techniques that can be used to either enforce memory safety or prevent memory-unsafe operations.

One potential approach to preventing memory faults is through the use of *memory-safe languages*. Memory-safe languages prevent unsafe operations at the level of the language design: commands that interact with memory are controlled through mechanisms that enforce specific restrictions designed to prevent memory faults from being introduced.

Languages that control memory accesses and maintain strong type checks, such as Java and Python, are usually considered memory-safe but do not fully prevent certain errors from occurring, though the impacts are typically mitigated. These additional control mechanisms typically introduce a cost to efficiency due to the necessary overhead needed to operate these controls. In some cases, these memory-safe languages are created from restricted subsets of existing programming languages, with one of the most well-known being MISRA-C, a subset of the C language designed for use in safety-critical applications. The advantages of this approach to safety-focussed language is obvious: a developer does not need to learn an entirely new language in order to develop a more reliable application, instead only needing to adapt to new procedures and the lack of certain functionalities. Alternatively, entirely new languages may be developed to address these new requirements. The Rust language [Matsakis and Klock, 2014] is one such example, in which memory accesses are carefully controlled and examined during compilation, with invalid memory operations detected by the compiler itself. This approach adopts some of the key advantages of memory management constructs such as garbage collectors while preserving some of the advantages of fine-grained control of memory. Memory-safe language principles have also been applied to system architectures, with systems such as CHERI introducing additional protections for the creation and use of program pointer variables [Woodruff et al., 2014] at the level of the underlying instruction set. Though such protections typ-

ically introduce a degree of storage overhead, the additional controls make unsafe modifications of memory significantly less likely.

While the above protections aim to prevent memory faults from being realised, alternative approaches aim to limit the potential damage caused by such faults. These mitigation systems are typically enforced at the lowest-levels of memory management, aiming to limit any potential damage caused through memory safety violations. Many of these approaches to protecting memory rely on obfuscating or randomising the arrangement of memory in order to limit the impact of malicious acts. Mechanisms such as Address Space Layout Randomisation (ASLR) aim to protect memory from malicious actions that rely on targeted locations being adjacent to an attacker-controlled location by randomising the assignment of memory blocks, distributing data around memory. Stack protection mechanisms such as [Cowan et al., 1998] aim to add additional data to stacks within memory, aiming to prevent attacks such as buffer overflow from succeeding by detecting attempts at corrupting that memory block. Though these techniques can limit the impact of security issues caused by memory safety violations, such approaches do not typically prevent these faults from being realised, and as such, should be seen as a final line of defence.

Though these potential protection mechanisms are quite effective, there are a few obvious limitations. One of the most notable is that language-based solutions can only easily be applied to future developments. While such approaches are obviously effective, preventing memory-unsafe operations through their design, such languages cannot easily be applied retroactively to existing systems without rebuilding the implementation entirely, a costly and time consuming effort. Further, the use of such languages require the developers to have familiarity with the languages to be used, which may require further training and expertise that is not available. For lower-level protections, the need to enable the mechanisms on the host system introduces both a degree of overhead as well as restrictions on the host systems that can be utilised by the program.

To further support the protections outlined above, verification techniques may be deployed. While these systems do not prevent memory faults in programs,

such techniques are able to identify them, allowing for a developer to be alerted to those faults and repair them before they can be realised, either during development or retroactively. Though prevention would be preferable, verification techniques allowing for memory safety to be established for a wider range of programs, without the additional costs of adopting the alternative protections. The refinement of a tool used as part of these verification strategies is the focus of this work.

2.2 Hoare Logic

One of the most foundational techniques developed in the early days of program verification was Hoare Logic. Developed by Hoare [Hoare, 1969], and influenced heavily by the preceeding works of Floyd [Floyd, 1993], Hoare Logic was one of the earliest approaches to representing programs in a formal manner, independent of hardware. The foundation of Hoare Logic is a representation of programs in an axiomatic fashion, representing each program command as a transformation of the program state. Each of these axioms can be viewed as an operation that, should relevant conditions be met, will accomplish some goal and satisfy some conditions. In other words, should some precondition be met, executing a program will ensure some postcondition. This is usually represented represented symbolically as

$$\{P\}\ C\ \{Q\}$$

where P refers to the precondition, Q represents the postcondition and C represents a command or program. The Hoare Triple is a commonly used construction for the representation of program specifications and additional constraints.

2.2.1 Operational Axioms

Hoare logic is based upon number of axiomatic rules, several focussed on the operation of specific key program operations, such as assignment and branching, and others focussed on the interactions between specifications.

The initial operational rules were designed to formally describe two key program commands, assignment and loops. Though these initial axioms permit only a relatively basic programming language, subsequent works extended this set of rules to cover a wider range of program commands, including if statements. Together, these operational axioms lay the foundation for many program verification tools and techniques. This accurate representation of the effects of a command on the state of a program allows for reasoning about the program state at any given point in the program. Such state representations may be utilised to reason about the potentially dangerous states, possible undefined behaviours or to simply consider the correctness of a state at a given program point.

[**Axiom of Assignment**] {P[f/x]} x := f {P}

The first rule to be presented here is the *axiom of assignment*. As the name suggests, the rule formalises the effects of an assignment of a value to a known variable.

$$[\textbf{Conditional Rule}]\frac{\{P \wedge S\}\ C_1\ \{Q\} \qquad \{P \wedge \neg S\}\ C_2\ Q}{\{P\}\ \text{IF}\ S\ \text{THEN}\ C_1\ \text{ELSE}\ C_2\ \{Q\}}$$

The Conditional Rule axiomatically describes conditional statements within programs. The core of this axiom is the need to branch the proof, creating two new conditions that must be proven: one where the condition holds, and the "true" branch is executed, and one where the condition does not hold, causing the "false" branch to be executed. For each of these cases, the command must end in a shared state Q, as whatever branch would be followed by the program must still result in a state that can be continued by the rest of the program. Nevertheless, this axiom is used alongside the "original" axioms without issue.

[**Rule of Iteration**] if $\{P \wedge B\}\ C\ \{P\}$ then $\{P\}$ **while** B **do** $C\ \{\neg B \wedge P\}$

The *rule of iteration* formalises the various outcomes of running a loop in a program. This axiom assumes loops of the `while(B)` form and explores the potential effects of the loop on the initial state in the cases where

2.2.2 Structural Axioms

Alongside the operational rules presented by Hoare, there are a small number of structural axioms. Unlike the operational rules, the structural axioms primarily describe the interactions between pre-and-post conditions and other logical formulae. Where the operation rules model the requirements and effect of commands over the program state, the structural rules primarily focus on the refinement and combination of specifications.

The Rule of Composition is one of the most powerful of the rules presented and discussed here. The Composition rule describes how two programs where the postcondition of one is equivalent to the precondition of the other may be composed together into a single larger program, constructing a new valid specification in the process. Formally, this rule is described as

$$[\textbf{Composition Rule}]\frac{\{P\}\ C;C'\ \{R\}}{\{P\}\ C\ \{Q\} \quad \{Q\}\ C'\ \{R\}}$$

and through the use of this rule, it is possible to compose many programs and commands together, producing a single specification for an entire block of code.

By utilising the composition rule, Hoare logic may be extended from the verification of single program commands into the verification of entire programs. This underlying approach remains to this day, with many analysis techniques taking advantage of the capabilities enabled by the rule.

The second structural axiom discussed here is the Rule of Consequence. This rule indicates that the validity of the specification would hold if the precondition was replaced with any formula that implies that precondition. Similar reasoning extends this transformation to postconditions, with any satisfied postcondition being replaceable by any expression implied by that postcondition. This natural extension can be expressed formally as the *Rule of Consequence*, detailed below.

$$[\textbf{Rule of Consequence}]\frac{\{R\}\ C\ \{S\}}{R \implies P \quad \{P\}\ C\ \{Q\} \quad Q \implies S}$$

While the rule of consequence describes the refinement of both the precondition and postcondition, the single rule can also be divided into two complimentary

rules, one focussing on the precondition and one focussing on the postcondition, as shown below:

$$[\textbf{Rules of Consequence}]\frac{\{R\}\ C\ \{Q\}}{R \implies P \quad \{P\}\ C\ \{Q\}} \quad \frac{\{P\}\ C\ \{R\}}{\{P\}\ C\ \{Q\} \quad Q \implies R}$$

By utilising these rules of consequence, it is possible to refine the specification of a command or program towards the most effective representation.

By allowing for the precondition of a program to be weakened, the specification can be progressed towards the *weakest liberal precondition*, the precondition that most minimally represents the necessary and sufficient conditions that will ensure the program operates correctly [Dijkstra, 1975]. Similarly, applying the complementary rules allows for the postcondition to be strengthened, refining the constraints to most accurately describe the outputs produced by the program. Through the combination of these two properties, a specification will be produced that will accept the widest range of inputs and results in an accurate and precise set of outputs.

2.3 Separation Logic

Separation logic is a logical system designed to allow for reasoning about the program heap. Separation logic is a well-established logical foundation and is used frequently during the analysis and verification of memory properties of programs. The key strength of separation logic lies in its ability to logically separate the heap into distinct regions, allowing for reasoning to be focussed on only the relevant regions. This process allows for analysis and verification methods to be made more scalable and more effective when reasoning about the program heap.

Separation logic shares much of the notation and operators of the first-order predicate logic of Hoare Logic, with a small number of additional operators and relations to support reasoning about memory. For the purposes of this work, we will focus on the "symbolic heap" fragment of separation logic, first outlined in [Berdine et al., 2005a]. The symbolic heap fragment is notable due to the

fact that it is a *decidable* fragment of separation logic, maintaining much of the capabilities of the full logic while also enabling reasoning about complex heap constructions such as linked lists. This fragment is utilised in a number of works in the literature [Berdine et al., 2007, Yang et al., 2008] and is also the basis of the work presented here (Section 2.5).

2.3.1 Operators and Relations

As an extension of Hoare Logic, Separation Logic makes use of many of the standard first-order predicate logic operators such as conjunction. However, in order to support more effective reasoning about the arrangement and contents of heap, a number of additional operations and relations were introduced.

Points-to Relation

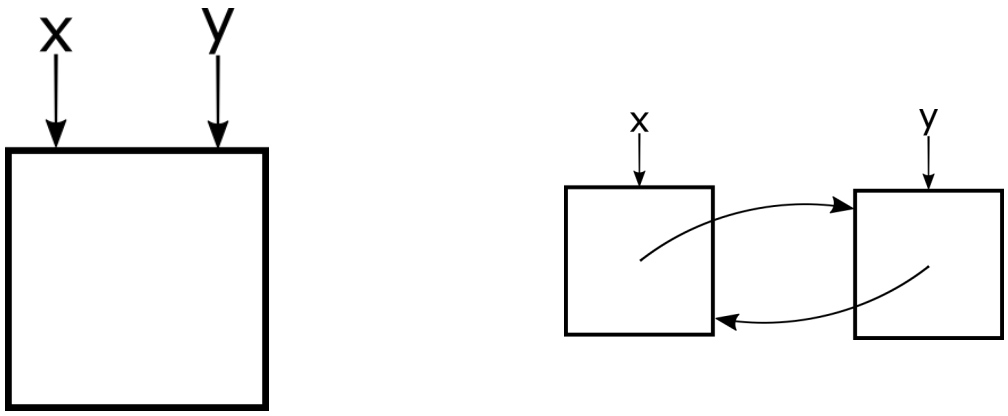
The first key addition over Hoare Logic is a points-to predicate, $\mapsto$, which represents the link between a memory address and the data stored at the referenced location. In the most direct applications, the address is a known value, as in the term

$$10 \mapsto [42]$$

which will hold in a scenario where the data value 42 is stored at the memory address 10, provided the rest of the heap is empty. This predicate also extends to variables, allowing for both the address and its contents to be represented as such. As a basic example, the formula

$$x \mapsto [5]$$

represents a simple scenario where a pointer variable x stores a location in memory in which the numeric data 5 is assigned. This notation is commonly used to relate assigned blocks of memory back to the pointer containing their location, simplifying the process of representing locations in memory and the variables that reference them. The relation can also be used to represent more complex



(a) $x{\mapsto}[y] \wedge y{\mapsto}[x]$ (b) $x{\mapsto}[y] * y{\mapsto}[x]$

Figure 2.1: Graphical representation of heap formulae

arrangements of stored data in a similar manner. As an example, consider the following separation logic term

$$y{\mapsto}[data : 1, next : z]$$

In this example, the pointer y does not point to a basic type, but instead a collection of data fields, grouped together into a singular cohesive block. These distinct fields are themselves referenced by a particular field name, in this case *data* and *next*, representing some offset from the start of the data block. Such terms are particularly common when dealing with data structures such as trees or lists, which are commonly found in software. While this representation is not a completely accurate translation of the arrangement of data in memory, it is an effective abstraction, like many of the techniques detailed here.

Separating Conjunction

The most significant operator introduced by separation logic is the separating conjunction operator, $*$. As discussed previously in this thesis, separation logic is primarily based around the notion of "separate" regions of heap, a logical division of the program heap into a number of discrete, non-overlapping and non-interfering sub-heaps. These separate regions are denoted through the of the *separating conjunction* operator, represented by the $*$ symbol. This binary operator acts in much of the same manner as classical conjunction, evaluating to

true should both arguments also evaluate to true. However, there are a number of key differences between the classical and separation logic conjunctions, with one of the most significant being that classical logic requires only that the two arguments evaluate to true, whereas separating conjunction places the additional constraint that the arguments must hold in *different* regions of the heap.

While this distinction may seem minor, it actually has a large influence on the heap states that can be represented by the use of these operators. Consider the two similar formulae:

$$x \mapsto [y] \wedge y \mapsto [x]$$

$$x \mapsto [y] * y \mapsto [x]$$

While both formulae seem to represent the same configurations of a heap, there is a significant difference introduced due to the use of the separating conjunction in the second formula; for the second formula to hold, there must be at least two distinct assigned nodes. Under standard conjunction, the formula only requires that some node x contains a pointer to location y, and that some node y contains a pointer to x. Under these constraints, a heap configuration where x and y are equal would satisfy these constraints, describing a heap state similar to that detailed in Figure 2.1a. With separating conjunction, however, this heap state would not be valid: as there would only be a single node under the condition where $x = y$, it would not be possible to partition $x \mapsto [y]$ and $y \mapsto [x]$ into separate heaps, thus invalidating the *separation* constraint of the operator. Figure 2.1b provides a corresponding illustration of the differences between the two formulae[2].

Separating Implication

The separating implication, $-\!\!*$, is a highly specialised binary relation in separation logic. Unlike the separating conjunction operator, the separating implication operator does not map as clearly to its classical counterpart, instead describing a heap state that will be satisfied if some existing heap is extended in some fixed

[2] x and y referring to differing nodes would also be a valid solution for $x \mapsto y \wedge y \mapsto x$, though only the "simplified" solution is presented here for clarity.

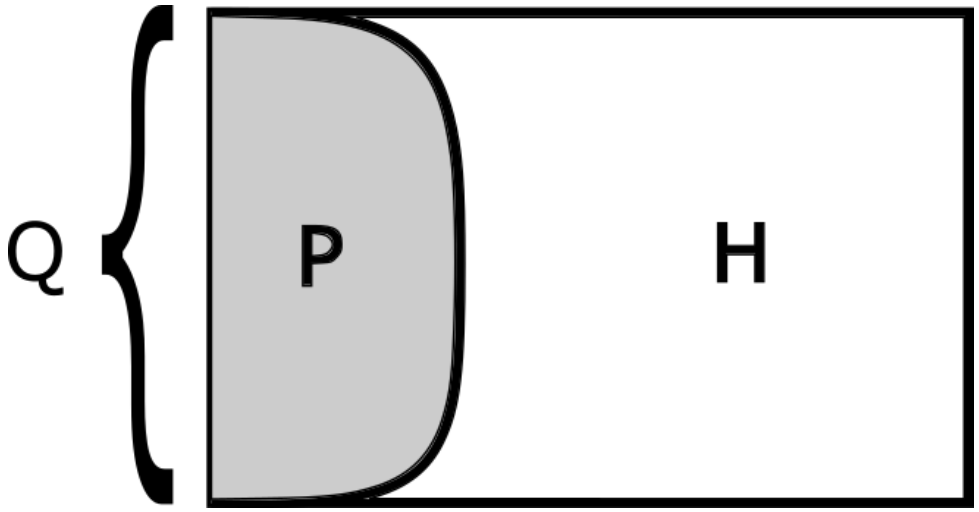


Figure 2.2: A graphical representation of Separating Implication

manner.

As an example, consider the following separation logic formula

$$\Delta * P \twoheadrightarrow Q$$

This formula (read as $\Delta * (P \twoheadrightarrow Q)$) states that if there is some discrete heap region that satisfies P, extending Δ with that sub-heap will produce a heap configuration that satisfies Q. While such a relation does not seem immediately helpful, the relation can be useful when attempting to prove correctness and could be particularly helpful when dealing with shape predicates (Section 2.3b).

Unfortunately, the complexity of determining the satisfiability of a separation logic formula with separating implication is quite high, with many of the used language fragments excluding the relation in order to maintain decidability, with the commonly-used symbolic heap fragment being one such example [Berdine et al., 2005a]. Nevertheless, the operator is an interesting, and potentially powerful, feature of separation logic.

Frame Rule

While not an operator or relation, one of the most powerful aspects of separation logic is an inference rule known as the *Frame Rule*.

$$[\textbf{Frame Rule}]\frac{\{\text{P}\}\ \text{C}\ \{\text{Q}\}}{\{\text{R}\ *\ \text{P}\}\ \text{C}\ \{\text{Q}\ *\ \text{R}\}}$$

(where R is not affected by C)

Introduced in [O'Hearn et al., 2001], the Frame Rule is an inference rule in the style of the Hoare Logic Rules of Consequence, enabling the "free" extension of an existing program specification, provided some key conditions are met. In essence, the frame rule outlines the fact that, given some specification $\{P\}C\{Q\}$, and some separation logic term or formula R that lies outside the memory footprint or effects of C, R may be introduced into the specification without affecting the validity or correctness of the specification.

This capability is of critical importance when attempting to analyse procedural heap-manipulating programs, allowing an analysis to operate over a given program fragment without requiring the analysis to consider the construction of other procedures or programs, only its specification. This functionality allows for far greater scalability due to this constrained area of interest, and is heavily related to another research area, frame inference, which is discussed in greater detail below (Section 2.4.1) due to its relevance to bi-abductive inference. In essence, however, the use of the frame rule allows for the specification of the invoked procedure to be extended to include the areas of program state that are not accessed by the procedure, without affecting the validity of the specification. This allows for the analysis to assume the procedure operates correctly - provided the precondition is satisfied - and adopt the now extended postcondition as the new state of the program following the return of the procedure, eliminating the analysis of the invoked procedure entirely without affecting the accuracy of the analysis.

2.3.2 Shape Predicates

One of the most powerful additions to separation logic following its development was the introduction of *shape predicates*. Many heap-manipulating programs take advantage of heap-based data structures such as linked lists and trees. These data structures typically follow a general structure in which each node within the structure contains some data as well as a link to the next node or nodes in the structure. This recursive design is common across these data structures, but is also the source of difficulties in analyses. More specifically, due to the (typically) unbounded size and abstract contents of the data structure, it can be difficult to reason about exact contents and layout of the heap, especially when dealing with external references to nodes in the middle of structures.

One of the best solutions to this approach was the inclusion of shape predicates, a formal description presenting the data structure as an recursively defined formula. The most basic example, and one commonly used when reasoning about programs making use of such data structures, is the singly-linked list segment, ls. The shape predicate for such structures relatively straightforward:

$$ls(x, y) \quad \stackrel{def}{=} \quad x = y \wedge \texttt{emp} \quad \vee \quad \exists x'.x{\mapsto}[x'] * ls(x', y)$$

This definition provides two distinct cases for the shape predicate, a *base* case in which the list fragment is empty between the two nodes and a *recursive* case, where there is an known assigned "head" node at x that points to some continuing segment of the list.

This recursive definition is a common one for shape predicates, outlining some base case allowing for the elimination of the predicate via unfolding operations, as well as a continuing recursive case which allows for the production of arbitrarily sized data structures. By utilising shape predicates of this style, it is possible for analyses to produce and examine formulae representing data structures of arbitrary size, provided an appropriate shape predicate is available.

2.4 Bi-Abductive Inference

The key technique explored in this work, bi-abductive inference, is a static analysis technique that aims to support the proof of entailments, usually those produced by verification tools [Calcagno et al., 2009]. Typically applied to establish memory-safety properties, bi-abductive inference provides a number of significant advantages to the analysis systems that it supports. First, bi-abduction is a very low-requirement technique, able to infer specifications without any additional input from the user, provided definitions for the data structures encountered are known. This in turn means that analysis tools making use of bi-abductive inference are able to operate with lesser burdens on the end user, making such tools more attractive for wider industry. Additionally, bi-abductive inference enables compositional analyses, analysis techniques in which the final result is a composite of a number of sub-problems.

Compositional analyses provide a number of significant advantages over the more typical monolithic approaches. Compositional analyses tend towards greater scalability, as the overall complexity of analysing a large program is reduced to a number of smaller and more manageable problems. Compositional analyses can also support parallelisation in the analysis, with the sub-problems divided across multiple threads, usually representing non-interacting call hierarchies, and gathering the desperate results together upon completion. However, perhaps the greatest strength of compositional analyses - particularly those enabled by bi-abductive inference - is the ability to allow for both graceful failure and incremental analyses.

As the overall analysis problem is split into a number of smaller sub-problems, with the results of these sub-problems being fed forwards where necessary, the failure of a specific sub-problem does not have an impact on unrelated sub-problems. Instead, the sub-problems which are not dependant on the failed sub-problem can be analysed and completed as normal, with only the aspects, either directly or indirectly, dependant upon the failed result affected. This partial success is typically far preferable to a total failure, as even when a portion of the program has

not been fully analysed, there remains information and results regarding the rest of the program, enabling some partial use of the results immediately. Indeed, depending on the nature of the approach, some partial or placeholder result from the failed sub-problem may still be used by dependant sub-problems, producing a still-useable result. This separation of problems enables a further enhancement: the possibility of incremental analyses.

Taking advantage of the independence of many of the sub-problems, the analysis may be restricted to specific aspects of the overall program, such as focussing only on parts of the code which are new or have been modified. In this way, any aspects of previously analysed code that has not been changed and is not reliant upon the results of the modified code does not need to be analysed again. Instead, the previously identified results are preserved and used immediately for the new analysis task, eliminating the need to re-examine the sub-problem code. This reduction of the scope of the overall analysis problem is an obvious advantage, allowing for the analysis to be made less computationally expensive, and perhaps increasing the overall speed of the process. These factors together show bi-abduction as a highly promising technique, and is may be part of the reason for the adoption of systems taking advantage of the technique in industry.

The core design of Bi-abductive inference can be viewed as a composite of two sub-problems, Frame Inference and Abductive Inference. In order to more clearly illustrate how the advances in these related fields can provide an advantage to bi-abductive techniques, as well as illustrate the source of some of the considerations that must be made to create an effective bi-abductive technique, we will first discuss these sub-problems in isolation, starting with frame inference.

2.4.1 Frame Inference

For verification systems that operate over procedural programs, in particular those that utilise entailment proving for the evaluation, frame inference is a critical task. Frame Inference is a technique that, given an entailment $\Delta_A \vdash \Delta_C$,

aims to identify some formula $?F$ such that the extended entailment

$$\Delta_A \vdash \Delta_C * [?F]$$

is satisfied.

This $?F$ fragment, referred to as the *frame*, consists of the elements of Δ_A that are not needed to satisfy the constraints in Δ_C, and are thus "extra" in the antecedent. Such operations are critical during the analysis of program procedure calls, as it is typical for a program to have a more complex state than the footprint of the invoked procedure. In such a scenario, the entailment would not be satisfied due to the additional state information, which is a less than ideal situation. Frame Inference allows for this to be addressed, as by identifying the additional information in the antecedent, it is possible to leverage the separation logic frame rule in order to carry the information forwards beyond the point of procedure call.

As discussed in the prior sections, verification efforts simply adopt the post-condition of the invoked procedure as the state following the return of that procedure, and by extending that postcondition with the frame of the entailment at the procedure call, all of the relevant state information is preserved.

2.4.2 Abductive Inference

Abductive inference is an analysis technique that aims to identify the "missing" portions of a program state required to satisfy some entailment. Initially inspired by the work of Peirce [Peirce, 1965] ([Calcagno et al., 2011]), abductive inference aims to find some *anti-frame* $?M$ such that the entailment:

$$\Delta_A \wedge [?M] \vdash \Delta_C$$

is satisfied. In essence, the technique makes use of the initial antecedent and consequent and infers some aspects absent from the entailment that may be required for that entailment to hold. This process also extends towards abductive inference in separation logic, which operates with a variation of the general problem which utilises the separating conjunction operator instead of classical conjunction, as

seen here:

$$\Delta_A * [?M] \vdash \Delta_C$$

Abductive Inference, when utilised as part of shape analysis, allows for the identification of so-called small specifications. As with frame inference, small specifications are specifications for program procedures that are functionally restricted to the safe operation of that procedure only, with all other state information ignored. In other words, the specification consists only of state that lies within the memory footprint of the procedure and its own invoked procedures, as well as the constraints over that state.

Abductive Inference is essentially an educated guess based upon the information available, much like in Pierce's original approach towards inference in scientific theory [Peirce, 1965]. However, for any given entailment, there is typically many valid solutions for the anti-frame, with the possibility for these solutions to vary greatly. While some of these solutions will be suitable for use, many of the solutions will be inefficient, describing a larger state than is necessary, or imprecise, describing a state that includes unnecessary information. In order to produce a useful output, most implementations of abductive inference will make use of a specialised ranking function in order to select the "best" output for a given set of target properties.

Ranking Functions

Ranking functions are a commonly-used method to filter the output of an abductive inference, reducing the full set of results to a smaller collection of "higher quality" anti-frames. The definition of high-quality used may vary across systems, but typically require that the solution is both valid and sufficient at a minimum, ensuring that the inferred fragments would correct the initial deficiencies in the antecedent. This minimum quality is commonly extended through the prioritisation of "smaller" solutions over other possible outputs. In this context, the size of a solution is typically a function of the memory footprint of the program state, either focussing on the inferred anti-frame alone or integrated into the initial en-

tailment, and aims to estimate the amount of assigned memory consumed by a given solution. The solutions that have the lower amount of memory required are deemed *smaller* from this measure.

In addition to these foundational measures of quality, further constraints may be enforced over the possible solutions. As one straightforward example, trivial solutions such as $?M = \texttt{false}$ are prohibited in order to ensure a useful solution is identified.

Another curious aspect of abductive inference is that, for a given set of conditions, an optimal solution is guaranteed to exist [Calcagno et al., 2011]. This optimal solution would obviously be the preferable output for the abductive inference, but identifying that optimal solution is often prohibitively expensive. However, higher quality non-optimal solutions are usually as effective as the optimal solution, and are far more quickly identified. Indeed, the complexity of generating the optimal solution is far more significant than the potential improvement in quality of the result; as with many aspects of bi-abductive inference, a "good enough" result is often preferable to the "perfect" result.

2.4.3 Overview of Bi-Abductive Inference

Bi-abductive inference combines the abductive and frame inference techniques such that, when given an entailment $\Delta_A \vdash \Delta_C$, a pair of formulae $?M$ and $?F$ is identified, aiming to ensure that the resulting entailment

$$\Delta_A \ * \ [?M] \vdash \Delta_C \ * \ [?F]$$

is satisfied. The entailments produced by this process essentially represent a refinement of the specification of a program, identifying constraints and state information that is absent from the precondition as well as ensuring any state information that is not affected by the various commands and procedures is carried forwards into the postcondition. This foundational use is one of the greatest strengths of bi-abduction: when suitably supported, the technique is able to synthesise full memory-safety specifications for a program from scratch, typically

with little to no input from the user.

For this work, we consider a slightly restricted variation of the bi-abductive problem, presented formally as follows. For the purposes of this definition, $FV(\Delta)$ is used to represent the free variables of the symbolic heap Δ.

PROBLEM:	`QF_BIABD`.
INPUT:	Δ_A and Δ_C where $FV(\Delta_C) \subseteq FV(\Delta_A) \cup \{\texttt{null}\}$.
QUESTION:	Does there exist $?M$ and $?F$ *such that* $\Delta_A * ?M \models \Delta_C * ?F$ holds?

This variation of Bi-abductive Inference essentially restricts the inference that can take place to fragments that consist only of fresh variables or concrete variables already present in Δ_C, simplifying the scope of the problem by preventing the inference of fragments that are essentially disconnected from the original state.

2.4.4 Bi-Abduction in Practice

Bi-abductive inference systems are typically utilised in support of a program analysis and are usually invoked when an entailment is generated. The most simple version of this arrangement is a Hoare-style forwards verification that analyses a target program line-by-line, building up the program state as it progresses. When these systems encounter a procedure call, an entailment is generated, utilising the current program state as the antecedent of the entailment and the precondition of the encountered procedure as the consequent.

Once the entailment has been identified, the bi-abductive inference system will attempt to identify appropriate frames and anti-frames in order to ensure the resulting extended entailment holds. These absent constraints are indications of potentially unsafe operations, representing faults such as uninitialised variables, hazardous aliasing or potential violations of shape properties of data structures, each of which must be addressed in order for the program to continue operation safely. These outputs are returned to the program analysis system,

with the anti-frame being propagated back into the precondition of the current procedure, refining the specification and ensuring that any inputs into the analysed procedure are appropriate for the code encountered. The frame is combined with the postcondition of the invoked procedure, with the result being adopted as the new program state before the program analysis continues. In this manner, areas of program state outside the footprint of the invoked procedure are preserved beyond the return of the procedure call.

However, these identified frames and anti-frames may not be appropriate for the program in question. Current bi-abductive systems are typically purely syntactic; they do not consider concrete memory configurations, instead aiming to establish validity or identify corrections for the generalised footprint represented by the symbolic states encountered. Further, as bi-abductive systems only have awareness of the supplied entailment, it is possible for the resulting frames and anti-frames to be contradictory to the aim or the program, or potentially unworkable when extended to the program as a whole. While some outputs may introduce obvious contradictions, triggering the verification system to attempt to utilise one of the other proposed results, it is possible for a valid solution to not be viable for the intended purpose of the analysed program. As a result, it is fundamentally up to the developers discretion as to how the output of bi-abductive inference systems is practically addressed in the analysed program.

In some cases, the refinement of specifications resulting from the propagation of the returned frames and anti-frames is sufficient, but even this can introduce the need for changes in systems that make use of that procedure or program, essentially deferring the modifications to another developer. Consider updating an algorithm specification to only accept sorted lists; systems that previously made use of that system must now also be updated to ensure that constraint is enforced, provided it is not already. This potential for the frames and anti-frames to introduce no changes is another potential outcome: if the analysed procedure has its specification strengthened in a particular manner, the developer may simply elect to *not* change the system, reasoning that the constraints are already enforced in practice, if not in theory. Ultimately, however, a limitation identified

by bi-abductive inference tends to indicate a flaw in the analysed program that can only practically be addressed via refinement of the program code.

2.5 Language and Semantics

In this section, we will present the semantics and language utilised in our main contributions, detailed in Chapters 4 and 5. These language fragments are an extended version of the symbolic heap fragment, with some of the key notation kept for consistency. This extended form of the symbolic heap fragment was necessary to develop, as the majority of separation logic fragments do not support the pure constraints that are one of the key focusses of the work. While a few fragments with these properties do exist, these rarely supported the full range of properties targeted by this project, leading to the development of the fragments outlined below.

2.5.1 Initial Language

The language used in our initial proof of concept system (Chapter 4) is presented in Figure 2.3a. In this language, program variables are defined as *italic* characters, and are used to refer to variables originating from within the program itself. Logical variables are indicated with upper-case letters, such as X, and refer to variables that appear in the analysis only. Values, where not directly numerical, are represented with the same *italic* characters as program variables, though are typically easily identified by context.

In order to improve the readability of this document, a small number of short-hand representations may be used: where unambiguous, the fields of a record may be omitted, with $x{\mapsto}[n{:}y, v{:}z]$ being shortened to $x{\mapsto}[y, z]$, as an example. We may additionally omit the square brackets around single-field records, as in $x{\mapsto}y$. Finally, variables in the formulae included here are typically represented in a generic fashion, using names such as E_1 to represent some arbitrary variable and

$x, y, ... \in Var$		Variables
$f, f_i, ... \in Fields$		Fields
$E ::=$	$\texttt{null} \mid x$	Expressions
$V ::=$	i, j, ... $\in Values$	Values
$\pi ::=$	$E{=}E \mid E{<}E$	Simple pure formulae
$\Pi ::=$	$\texttt{true} \mid \pi \mid \neg\Pi \mid \exists v.\Pi \mid \Pi \wedge \Pi$	Pure formulae
$\rho ::=$	$f_1 : E_1, \ ... \ , f_k : E_k$	Record expressions
	$\texttt{emp} \mid E{\mapsto}[\rho] \mid \Sigma * \Sigma$	
$\Sigma ::=$	$\mid ls(E, E) \mid sls(E, x, y, E)$	spatial formulae
	$\mid tree(E) \mid stree(E, x, y)$	
$\Delta ::=$	$\Pi \wedge \Sigma$	qf symbolic heaps
$H ::=$	$\exists \overrightarrow{X}.\Delta$	symbolic heaps

(a) Grammar Definitions

$$ls(E_1, E_2) \quad \stackrel{def}{=} \quad E_1 = E_2 \wedge \texttt{emp} \ \vee \ \exists E'.E_1 \neq E_2 \wedge E_1{\mapsto}[E'] * ls(E', E_2)$$

$$sls(E_1, V_1, V_2, E_2) \quad \stackrel{def}{=} \quad E_1{=}E_2 \wedge V_1{=}V_2 \wedge \texttt{emp} \ \vee \ \exists E', V'.V_1{\leq}V' \wedge E_1{\neq}E_2 \wedge E_1{\mapsto}[E', V_1] * sls(E', V', V_2, E_2)$$

$$tree(E) \quad \stackrel{def}{=} \quad E = null \wedge \texttt{emp} \ \vee \ E{\mapsto}[l, r] * tree(l) * tree(r)$$

$$stree(E, V_1, V_2) \quad \stackrel{def}{=} \quad E = null \wedge V_1 = V_2 \wedge \texttt{emp} \ \vee \ \exists V_3, V_4.V_3 \leq V' \wedge V' \leq V_4 \ \wedge \ E{\mapsto}[l, r, V'] * stree(l, V_1, V_3) * stree(r, V_4, V_2)$$

(b) Predicate Definitions

Figure 2.3: Language Fragment

V_1 to represent some arbitrary value.

The language includes inductive definitions describing fundamental list struc-

tures: ls, representing a simple singly-linked list, and sls, representing a *sorted* singly-linked list. We have also included definitions for a binary tree *tree* and a binary search tree *stree*, a subsequent addition to the work detailed in [Curry et al., 2019]. The full inductive definitions of these shape predicates may be found in Figure 2.3b, though a brief overview of each will be provided below.

A singly-linked list $ls(E_1, E_2)$ consists of a series of nodes linked via a pointer field, starting with some E_1 and ending at some E_2 (non-inclusive). Each of the list predicates included in our system can be used to define a list segment or a full null-terminated list, dependant only on the value of E_2. A *sorted* singly-linked list $sls(E_1, V_1, V_2, E_2)$ is a sequence of singly-linked nodes, beginning with some node E_1 and ending at some node E_2 (non-inclusive), with all values stored in those nodes obeying an ascending order. In order to simplify the checking of these structures, the minimum and maximum values of the list are also tracked inside the predicate, with the minimum value being represented as V_1 and the maximum V_2. Note that V_2 *is not* the value of node E_2; rather, V_2 would refer to the value of the final node in the list, which points to E_2. Both $tree(E)$ and $stree(E, V_1, V_2)$ describe a binary tree: a single node pointing to two known subtrees, or a `null` root describing an empty tree. As with ls and sls, the primary difference between *tree* and *stree* is the consistent ordering constraints present in *stree*, which ensures that all values in the left subtree of an *stree* are lower than the value of the root node, and all values in the right subtree are greater than the value of the root. Minimum and maximum values are recorded in *stree* predicates in a similar manner to sls predicates.

These predicates were selected for two primary reasons: simplicity and popularity. Singly-linked lists and binary trees are highly common data structures in modern programming, yet are also some of the most simplistic [Cormen et al., 2009]. Many initial tools and prototype techniques in the area of memory safety initially focussed on these simpler predicates in order to demonstrate and explore the effectiveness of the approach [Sagiv et al., 1998, Berdine et al., 2006]. Indeed, the SL-COMP benchmark set includes an entire division dedicated to singly-linked list examples. As the technique I was aiming to develop was largely

untested and only lightly explored, it was decided that a more simplistic fragment would be used initially, with subsequent explorations focussing on a more complex language.

Semantics

The semantics of this fragment are quite standard, following from the semantics of separation logic with general inductive definitions and arithmetic presented in [Le et al., 2017]. These semantics are given by a relation $s,h \models H$ that forces the stack s and heap h to satisfy the constraint H, where $h \in \mathit{Heaps}$, $s \in \mathit{Stacks}$, and H is a formula. Stack and heap abstractions are defined as:

$$\begin{array}{lll} \mathit{Heaps} & \stackrel{\text{def}}{=} & \mathit{Values} \rightharpoonup_{fin} (\mathit{Fields} \times \mathit{Values})^N \\ \mathit{Stacks} & \stackrel{\text{def}}{=} & \mathit{Var} \rightarrow \mathit{Values} \end{array}$$

where N is the maximum number of fields.

The semantics of our fragment are defined as follows:

$$\begin{array}{lll} s,h \models \texttt{emp} & \mathit{iff} & \mathit{dom}(h)=\{\} \\ s,h \models E \mapsto [f_i{:}v_i] & \mathit{iff} & \mathit{dom}(h)=\{s(E)\}\ h(s(E))=(\ (f_1,s(E_1)),...,(f_N,s(E_N))\ ) \\ s,h \models \Sigma_1 * \Sigma_2 & \mathit{iff} & \exists h_1,h_2 \cdot\ h_1 \# h_2 \text{ and } h=h_1 \cdot h_2, s,h_1 \models \Sigma_1 \text{ and } s,h_2 \models \Sigma_2 \\ s,h \models \texttt{true} & \mathit{iff} & \mathit{always} \\ s,h \models \Pi \wedge \Sigma & \mathit{iff} & s,h \models \Sigma \text{ and } s \models \Pi \\ s,h \models \exists v.\Delta & \mathit{iff} & \exists \alpha \cdot s[v \mapsto \alpha], h \models \Delta \\ s,h \models H_1 \vee H_2 & \mathit{iff} & s,h \models H_1 \text{ or } s,h \models H_2 \end{array}$$

where $\mathit{dom}(f)$ is the domain of function f, $h_1 \# h_2$ denotes disjoint heaps h_1 and h_2 i.e., $\mathit{dom}(h_1) \cap \mathit{dom}(h_2)=\emptyset$, and $h_1 \cdot h_2$ denotes the union of two disjoint heaps. If s is a stack, $v \in \mathit{Var}$, $v \notin \mathit{dom}(s)$ and $\alpha \in \mathit{Values} \cup \mathit{Loc}$, we write $s[v \mapsto \alpha] \equiv s \cup \{(v,\alpha)\}$. Note that in a concrete memory model such as the RAM model, the field names of points-to predicates are transformed into pointer offsets. The pair $(f_i, s(E_i)$ (for all $i \in \{1...N\}$) would then be interpreted as $s(E) + \mathit{off}_{f_i} = s(E_i)$, where off_{f_i} is the corresponding offset of field f_i. The entailment $H \models H'$ holds iff for

all s and h, we have if $s, h \models H$ then $s, h \models H'$. Note that we also preserve $s(\texttt{null})$ as a special value such that it is not in any domain of heaps.

2.5.2 Generalised Language

$x, y, ... \in Var$		Variables
$f, f_i, ... \in Fields$		Fields
$V ::=$	i, j, ... $\in Values$ | 1, 2, ... $\in Values$	Values
$E ::=$	$\texttt{null} \mid x \mid V$	Expressions
$\pi ::=$	$E{=}E \mid E{<}E$	Simple pure formulae
$\alpha ::=$	$V + V \mid V - V \mid V \star V \mid V \div V \mid$ $\texttt{MAX}(V,V) \mid \texttt{MIN}(V,V) \mid \texttt{ABS}(V)$	Arithmetic formulae
$\beta ::=$	$\{\ \} \mid \{E_0, E_1, ...\} \mid \{V_0, V_1, ...\} \mid$ $\beta \cup \beta \mid \beta \cap \beta \mid \beta \subset \beta \mid \beta - \beta$	Bag formulae
$\Pi ::=$	$\texttt{true} \mid \pi \mid \alpha \mid \beta \mid \neg\Pi \mid \exists v.\Pi \mid \Pi \wedge \Pi$	Pure formulae
$\rho ::=$	$f_1 : E_1, \ ... \ , f_k : E_k$	Record expressions
$\Sigma ::=$	$\texttt{emp} \mid E{\mapsto}[\rho] \mid \Sigma * \Sigma \mid P(\overrightarrow{E})$	Spatial formulae
$\Delta ::=$	$\Pi \wedge \Sigma$	qf symbolic heaps
$H ::=$	$\exists \overrightarrow{X}.\Delta$	Symbolic heaps

(a) Generalised Grammar Definitions

$$P(\overrightarrow{E}) ::= (\exists \overrightarrow{X_1}.\ \Pi_1 \wedge \Sigma_1) \vee ... \vee (\exists \overrightarrow{X_n}.\ \Pi_n \wedge \Sigma_n)$$

(b) Shape Predicate Definition

Figure 2.4: Generalised Language Fragment

The generalised language fragment used in the second developed system (Chapter 5) is based on the language fragment utilised by the proof-of-concept system, adapted to include a larger range of pure properties and generalised shape predicates. As such, the conventions around the presentation of variables are preserved, with *italic* variables indicating program variables or numeric data and upper-case

characters representing logical variables. Similarly, the various shorthand notations used in the first language are extended to this fragment, with record fields simplified where possible. One key addition is a notation to represent a vector of variables, $\overrightarrow{X}$. While this notation simply represents an ordered collection of variables, as may be seen in the parameters supplied to a shape predicate, one key condition is enforced: the first item of the collection is the variable used to denote the collection as a whole. As an example, the lead variable of $\overrightarrow{X}$ would simply be X itself.

One of the key differences between the two languages is the additional pure constraints permissible in the second language, namely arithmetic constraints, α, and bag (or multiset) constraints, β. These additional pure constraints are found in a number of cutting edge systems in the literature [Le et al., 2018, Qin et al., 2017] and are of significant benefit when reasoning about non-relational properties of data structures such as contents and size or length. Further, the generalised language fragment makes use of a general shape predicate, $P(\overrightarrow{E})$, instead of the predefined shape predicates for lists and trees utilised in the language used for System 1. Presented in Figure 2.4b, this general shape predicate enables the resulting systems to reason over a far wider range of shape predicates than was possible in the initial system which was restricted to only ls, sls, $tree$ and $stree$ predicates.

This general shape predicate also has a number of additional constraints that must be met for the predicate to be considered well-formed. These constraints are designed in order to ensure a reliable and productive set of behaviours under key operations such as unfolding, simplifying the design of the proof rules of System 2.

Chapter 3

Literature Review

In this chapter, a number of key techniques from the literature that are related to the work developed during this project will be presented and discussed. Additionally, this chapter aims to establish the position of our work against the existing literature. These works presented here are divided into three categories: works that are fundamental to the inference systems that this project aimed to develop, alternate approaches to accomplish the verification of programs with dynamic memory and finally, a number of works and systems directly related to shape analysis and bi-abductive inference in particular. SMT solvers will be briefly discussed in Section 3.1 before presenting symbolic execution in Section 3.2. Following this, we will present the works directly related to this project, with works focussing on shape analysis presented in Section 3.3 and works focussing on bi-abductive inference systems presented in 3.4.

To preserve some consistency throughout these discussions, the conventions around the display of formulae presented in the Preliminaries chapter (Chapter 2) will be continued here.

3.1 SMT Solvers

Another key technique used by many verification systems are satisfiability solvers. Satisfiability solvers, or SAT solvers for short, are automated systems that inspect a logical formula and attempt to identify whether that formula has a valid solution. Such techniques are of great use to verification systems as it allows for such tools to inspect program constraints and identify whether such states have a valid solution, indicating whether a particular program path may be followed or not, and what the effect of such a path may be. Such logic can also be reversed: if a set of constraints leading to an erroneous state can be identified, the existence of a solution to those constraints indicates the presence of a bug in the program.

However, as SAT solvers can support boolean formulae only, such systems are naturally limited in their applications. To address this limitation, an improved approach to satisfiability problems was developed: *Satisfiability Modulo Theory* solvers, or SMT solvers. SMT solvers, like SAT solvers, inspect a set of constraints and attempt to identify whether a valid solution exists for that formula. Unlike SAT solvers, however, SMT solvers can support a larger range of underlying theories beyond boolean satisfiability, allowing for the satisfiability problem to be solved for formulae containing a wide range of constraints, including arithmetic formulae, bit vectors and strings.

Though these solvers are effective over more typical formulae, separation logic formulae are rarely supported. Satisfiability solvers for separation logic are typically specialised, with a number of verification and proof systems taking advantage of such techniques [Navarro Pérez and Rybalchenko, 2013, Le et al., 2016, Gu et al., 2016a, Le et al., 2018].

The literature also describes a range of techniques and solvers that aim to convert separation logic formulae into an an equivalent first-order representation, allowing for more typical SMT solvers to be applied. Some systems, such as GrassHOPPER [Piskac et al., 2013, Piskac et al., 2014b, Piskac et al., 2014a] describe a first-order decidable logic to which separation logic formulae can be reduced. Once that transformation has been completed, the translated formula is

then passed into a typical SMT solver in order to confirm or disprove the validity of that formula. The Dryad logic [Madhusudan et al., 2012, Qiu et al., 2013] is another approach to addressing the satisfiability of separation logic formulae. This "dialect" of separation logic is a system which permits the inclusion of both integer and spatial constraints that can be translated into an equivalent formula in the logic of integers and sets. This translated formula, being within typical first-order theories, can subsequently be discharged via typical SMT solvers.

3.2 Symbolic Execution

Symbolic execution is a static analysis technique where instead of executing a program over concrete inputs, such as numerical data, the program is applied to *symbolic* input. By running the program in this manner, it is possible to create symbolic states, a general formula representing the state of the program in a more generic manner. Through the analysis of these symbolic states, it is possible to identify potential faults in a program through the identification of inputs that would lead to invalid symbolic states or to establish key properties of a program, such as correctness with respect to some known specification.

While symbolic execution has been implemented in a wide range of systems, we focus on a version of symbolic execution that features support for separation logic [Berdine et al., 2005b]. Utilising this new form of symbolic execution, new verification systems could be created with straightforward support for dynamic memory. Tools such as Smallfoot [Berdine et al., 2005b], JStar [Distefano and Parkinson J, 2008] and later VeriFast [Jacobs and Piessens, 2008, Jacobs et al., 2011] would all take advantage of this innovation to enable the analysis of programs that manipulated the heap.

Symbolic execution remains a key technique in modern systems, with many tools utilising the approach to establish and maintain symbolic states for use in other settings, including shape analysis (Section 3.3), testing [Fragoso Santos et al., 2019, Fragoso Santos et al., 2020] and as a key support structure for bi-

abductive systems [Calcagno et al., 2009, Qin et al., 2017, Fragoso Santos et al., 2020].

3.3 Shape Analysis

Shape analysis is a well-established method for the verification of memory safety properties of programs. While the approach existed prior to 1998, the technique did not appear to receive much attention until the work [Sagiv et al., 1998]. After this work and those that soon followed, shape analysis became established as an effective approach towards the analysis of heap-manipulating programs and remains a commonly investigated approach today. Shape analysis operates by examining the "shape" of assigned blocks of memory - a model representing the relative interconnectedness and contents of the program heap - and using that information to reason about potential memory-safety violations in the program, including faults such as null-reads, use-after-free and memory leaks.

The earliest versions of shape analysis often suffered from the same issues affecting other analysis techniques targeting heap-manipulating programs. As has been briefly discussed in the context of separation logic (Section 2.3), these initial systems were often limited due to the need to track all assigned areas of memory simultaneously (as discussed in [Yang et al., 2008]), introducing significant scalability issues and restricting the effective capabilities of the systems to only small programs. As the field continued to develop, more efficient techniques were developed in order improve the capabilities of tools for the verification of heap-manipulating programs. Numerous shape analysis techniques have been developed based upon a variety of underlying representations of the program memory, such as abstract interpretation (Section 3.3.2), memory graphs (Section 3.3.1) and separation logic (Section 3.3.3). These underlying techniques are one of the key distinguishing features of the various shape analysis techniques, and the works will be grouped and discussed accordingly.

3.3.1 Graph-Based Shape Analysis

One of the earliest representations of memory used in shape analysis are *memory graphs* [Sagiv et al., 1998]. These graphs model the arrangement and connectedness of the blocks of memory by representing the assigned memory as the nodes of the graph and representing the relations and connections between those memory locations as the edges. Though the specific mechanisms vary between works, this approach remains a commonly used foundation in shape analysis techniques.

One of the earliest shape analysis techniques was the work of Sagiv *et. al.* [Sagiv et al., 1998], which utilised a graph-based representation to model the arrangement and interconnectedness of the heap.

Though reasonably effective, its successor, TVLA system [Sagiv et al., 2002] is perhaps the most well-known. The TVLA system was one of the first techniques to achieve a capacity for effective and relatively scalable shape analysis for heap-manipulating programs, able to establish safety properties as well as process basic numeric constraints. The key advance of the TVLA system was the use of a novel three-value logic, allowing for the memory graph to model *possible* relations alongside known relations and known non-relations. This "maybe" relation was particularly effective when applied to recursive data structures, where reasoning is typically restricted to only the "root" of the structure, with any subsequent nodes only a possibility. However, as is typical of early shape analysis techniques, TVLA was still required to record and track all assigned areas of memory in order to ensure the model was properly updated through aliasing, negatively impacting the overall scalability.

Graph-based structures continued to form the basis of many techniques [Dudka et al., 2011] and continue to be utilised today, though such foundations are often supported with separation logic in more recent systems (Section 2.3).

Arising from the same area of graph-based shape analysis techniques was the development of techniques based around the use of *forest automata* to validate memory. Forest automata are tuples of tree automata that may include links

from leaves of one tree to the roots of other trees and will accept any tuple of trees that satisfies one of the automata that comprise the forest. As noted in [Habermehl et al., 2011], memory graphs may be directly decomposed into sets of trees by splitting the graph at *cut points*, in this case defined as any node pointed to by many possible transitions or a program variable. By splitting a heap graph into appropriate trees and passing the resulting tuple into an appropriate forest automata, it is possible for these techniques to establish the memory safety of that program. At present, this approach to memory representation is only utilised by a small number of tools [Habermehl et al., 2011, Abdulla et al., 2013, Holík et al., 2013], though these approaches appear to be effective and competitive with shape analysis systems based on other techniques.

3.3.2 Shape Analysis via Abstract Interpretation

Initially proposed by Cousot and Cousot [Cousot and Cousot, 1977], Abstract Interpretation is a foundational technique utilised in program analysis. The technique is based upon an abstracting function, which aims to convert some concrete program state into an equivalent abstract representation. This abstract state is behaviourally equivalent to the initial program, though with a number of the complexities of the concrete program eliminated. This "simpler" over-approximation of the program can subsequently be analysed in greater detail, aiming to identify points where erroneous states may arise.

Though the results produced by an analysis of the abstract state do not guarantee the same results over the concrete program due to the abstraction process, such analyses can be used to establish sound upper bounds instead. These bounds allow for key properties of a program to be established, as if the upper bound lies within a "safe" range, no corresponding concrete states will lie outside that range. This is often enough to establish the absence of key errors, as demonstrated by the Astré system, one of the first verification systems to be applied to safety-critical software, namely the avionics software of Boeing aircraft [Cousot et al., 2005].

A number of shape analysis techniques operate by producing and analysing such an abstract model of the assigned memory. Reasoning over this abstract model allows for the systems to examine key properties of shape and memory safety for a general set of memory configurations, establishing safety properties for the program under analysis.

[Lee et al., 2005] outlines a graph-and-grammar based shape analysis technique that operates by collecting and converting state information into an abstract representation of the state, formalised as a graph. The technique also makes use of separation logic to aid partitioning.

[Hackett and Rugina, 2005] presents a compositional shape analysis technique based upon an abstract representation of the program heap. This technique is of interest due to the fact that it exploits local reasoning to aid in the simplification of the analysis, using specialised decomposition functions to divide the abstract state into a number of independent "configurations". While viable, few modern techniques make use of this approach, with separation logic typically used instead.

[Berdine et al., 2007] proposes a shape analysis technique aimed at complex composite data structures typical of those found within device drivers, such as lists of cyclic lists with backlinks. The system is based around the use of abstract interpretation over program states produced via symbolic execution to construct a higher-order shape predicate describing the data structure. This system was one of the first to explore atypical shape predicates such as the "cyclic doubly-linked lists with backlinks" described in the work.

Some shape analysis techniques based on abstract interpretation also have the capacity to operate in combined shape and pure domains, able to effectively analyse a wider range of programs. [Chang and Rival, 2008] presents a parametric abstract domain for relational inductive shape analysis for shape and numeric data. This approach modelled properties dependant upon relations between adjacent nodes by linking the relation with the spatial connection on the edge of the memory graph itself. This system built upon a previous work [Chang et al., 2007], extending the previously non-relational system to allow for the handling of more

complex properties, though the shape predicates must once again be supplied by the user.

Another abstract interpretation based parametric shape analysis for heap, numeric and array constraints of programs is described in [McCloskey et al., 2010]. Named `DESKCHECK`, this system aimed to establish memory safety properties for complex shape predicates similar to those targeted in [Berdine et al., 2007], though with additional numerical constraints. The heap and numeric domains utilised in this tool are also separated, with predicate definitions restricted to only one domain, though references to predicates defined (acyclically) in another are allowed.

3.3.3 Shape Analysis with Separation Logic

With the development of separation logic [O'Hearn et al., 2001, Reynolds, 2002], shape analysis techniques were able to exploit the capabilities of local reasoning to achieve levels of scalability that were not previously achievable with the techniques available at the time [Yang et al., 2008].

Smallfoot [Berdine et al., 2006, Berdine et al., 2007] appears to be the first application of separation logic to the analysis of programs, developing and subsequently deploying a symbolic execution system with support for separation logic under a simplistic procedural language. This system aimed to take advantage of the scalability separation logic can enable through the use of the principle of local reasoning and the frame rule and appeared to be an effective tool when compared to others of the time. However, the Smallfoot system was restricted to only a simple symbolic heap fragment of separation logic containing only basic (in)equalities and spatial terms, along with support for singly-linked list predicates. While this is common amongst early shape analysis techniques, the overall expressiveness of the tool was heavily limited, with systems such as TVLA able to support larger ranges of properties [Sagiv et al., 2002]. Smallfoot is also of note as several aspects of its design influenced the work undertaken in this project. The SLAyer verification system [Berdine et al., 2011], capable of establishing memory

safety properties of programs of 10-30k lines of code, was also an evolution of this system.

The work of Distafano *et. al.* [Distefano et al., 2006a] describes another shape analysis technique built upon the symbolic execution system for separation logic of [Berdine et al., 2005b]. While following in the same general structure of [Berdine et al., 2006, Berdine et al., 2007], the work of Distafano additionally integrated abstract interpretation into the system, utilising the system in order to more effectively reach fixpoints when performing the analysis.

Space Invader [Yang et al., 2008] outlines a verification system for programs with dynamic memory underpinned by a separation logic proof search supported by the use of abstract interpretation to enable convergence in the proof. While utilising a similar foundation as [Distefano et al., 2006b], Space Invader is notable due to its capability to effectively analyse programs of up to 10k lines of code [Yang et al., 2008], a level of scalability that was previously unachievable in the field. This system is further of note due to an experimental *compositional* extension of the tool named Abductor, which would eventually become the basis of bi-abductive inference [Calcagno et al., 2009].

Techniques based around the use of memory graphs also began to integrate separation logic into their designs, aiming to take advantage of the strengths of both mechanisms. A notable example of this hybrid representation is the Predator shape analysis tool [Dudka et al., 2011]. This verification tool is an automated implementation with support for a range of list predicates, underpinned by both separation logic and a model of memory represented using a graph and aimed at the analysis of real-world system software, namely core Linux code. The tool was also specialised further with the Predator Hunting Party variant [Muller et al., 2015], the leading technique for heap-manipulating programs in the 2015 SV_COMP competition [Beyer, 2015].

Finally, a number of shape analysis systems based on separation logic have been built or extended to support pure properties alongside spatial constraints. [Bansal et al., 2009] utilises an extended separation logic that features enhanced

pointer relations and ordering constraints, capable enough to establish the preservation of sortedness during the merging of two ordered lists.

[Magill et al., 2007] describes a technique to combine a separation-logic based shape analysis with an arbitrary arithmetic analysis to allow for the analysis of both shape and basic numerical data via the construction of counterexample programs. This approach further makes use of abstract interpretation with separation logic in order to accomplish the shape analysis tasks required as part of this approach. The THOR analysis tool [Magill et al., 2008] builds upon this foundation, aiming to improve the precision of the approach, before [Magill et al., 2010] makes efforts to expand the numerical analyses towards a general numeric analysis, though the shape and pure constraints are still analysed separately.

[Chin et al., 2012] also introduces a verification system for shape, bag and arithmetic properties, based around forwards analysis and abstraction. This system would eventually become the basis of the SLEEK system, which in turn would be extended with bi-abductive capabilities in later works [Qin et al., 2017].

3.4 Bi-Abductive Inference

Initially developed by Calcagno *et. al.* [Calcagno et al., 2009, Calcagno et al., 2011], bi-abductive inference, or bi-abduction for short, has become a frequently used and powerful technique in the area of program verification. While briefly discussed in the corresponding preliminaries section (Section 2.4), the bi-abductive inference technique can be formulated as a pair of sub-techniques, frame inference and abductive inference. While abductive inference - a technique designed to identify anti-frames - rarely appears outside of bi-abductive tools, a small number of tools do make use of such systems to support analyses. The CABER tool of the Cyclist library is one such instance, describing a cyclic abduction system to infer shape predicates from the program state [Brotherston and Gorogiannis, 2014]. Frame inference is more commonly encountered in the literature, as the identification of frames is a significant benefit to systems that handle procedu-

ral programs. Similar to abductive inference systems, however, very few such works are developed as a stand-alone system - as in [Le et al., 2017] - limiting the applications of new frame inference techniques to bi-abductive systems. Nevertheless, advances in both abductive inference and frame inference techniques are of indirect benefit to bi-abduction.

From the first presentation of bi-abduction, the technique has seen a massive growth in interest and popularity, being integrated into a number of powerful analysis tools. The most well-known of these tools is the Infer static analysis tool developed and deployed at Facebook, arguably the first application of the technique in industry. In Infer, the strengths of the technique are highlighted, with the rapid development style of Facebook (now Meta) proving itself challenging, but not insurmountable, to the analysis approach [Calcagno and Distefano, 2011]. Bi-abduction enabled the ability to undertake compositional and iterative analyses, as well as the low burden on developers due to the low requirements helping to create an effective, fast and low-requirement analysis tool. The Infer tool has seen continuing development since its first incarnation and has in many ways moved beyond a simple verification tool with bi-abduction [Calcagno et al., 2009], with the most recent versions even taking advantage of the novel incorrectness logic to operate [O'Hearn, 2019]. Nevertheless, it remains a key and highly relevant work.

One of the more advanced bi-abductive techniques known is the work of Le *et. al.* [Le et al., 2017], which outlines a specialised second-order bi-abduction mechanism. This approach is notable as it overcomes one of main limitations of bi-abductive inference: the need for either hard-coded or user-supplied definitions for the shape predicates that are to be encountered during the analysis. In other systems, this limitation means that the tools can only operate effectively over a set of known predicates. While the set of encountered predicates are typically standard data structures, such as singly-linked lists and trees, this limitation remains a brittleness to the technique.

Finally, the most recent work in bi-abductive inference is the Gillian framework [Fragoso Santos et al., 2020, Maksimović et al., 2021]. The Gillian frame-

work is a testing and verification framework that utilises bi-abductive inference in order to correct inferences and generate test cases. While this work does not appear to advance the technique of bi-abductive inference, the technique is applied in an atypical context and in support of a more general verification and testing framework.

3.4.1 Combined Domain Bi-Abduction

Combined domain bi-abductive techniques are generally a rarity and are directly related to to the work undertaken during this project. Such variants of bi-abduction, in addition to their capabilities in the shape domain, are also able to infer more general "pure" properties such as ordering, size and length, and contents. These more expressive techniques are thus able to effectively handle more complex forms of the standard bi-abductive problem and can infer expressive and precise specifications or identify deficiencies for a far wider range of programs.

The earliest known form of combined-domain bi-abduction is described by Trinh *et. al.* [Trinh et al., 2013], outlining a framework to enhance the results of an existing bi-abductive technique via shape inference mechanisms in order to infer specifications in the combined domain. This technique in particular highlights the potential issue of the "split domain" approach: the final outputs are heavily reliant upon the initial shape analysis, with any errors propagating through to the resulting specification In addition, the shape inference is completely divorced from the shape inference, leaving open the possibility of imprecision or inaccuracies when producing the pure constraints independent of the shape information, though this remains speculative.

The next known work is presented in [He et al., 2013, Qin et al., 2017], outlining an bi-abductive mechanism in the HIP/SLEEK system. In this system, the program state is built up line-by-line in a forwards analysis until the end of the program is reached. At this point, the system applies an abstraction function and iterates, aiming to repeat the process until a fixpoint is reached. Should the analysis reach a point where it is unable to progress, the bi-abductive mechanism

would be applied, aiming to identify the frame and anti-frame necessary for the analysis to proceed.

This mechanism is a relatively thorough instance of bi-abduction, but has a number of key limitations. The first, and perhaps the most impactful, is the heavy reliance upon entailment proofs for the bi-abductive inference. The proof rules comprising the bi-abductive mechanism, while very broadly applicable and ultimately effective, are either triggered by or produce entailments ensuring sound applications. Though these entailments are handled by the SLEEK subsystem, the approach seems fairly expensive, as such entailments will still take both time and effort to be solved. The second major limitation is that the system is not able to fully handle predicates with both shape and pure constraints. While much of the context and state can be gathered from the standard forwards analysis, the system is not fully able to fully process predicates with both shape and pure constraints as part of the bi-abductive mechanism. Instead, an specialised "abductive abstraction" is applied, inferring further pure information in order to abstract the state information into a corresponding predicate.

A common theme amongst all of these techniques is that the bi-abductive mechanisms are unable to fully process inference in the combined domain without heavy reliance upon separate tools or analysis phases. As mentioned in the Motivation discussion earlier (Section 1.2), it is suspected that this limitation carries negative impact on the performance of such tools, and investigating this hypothesis is the key focus of this work.

Chapter 4

Combined Domain Bi-Abduction

In this chapter, the work undertaken to investigate and create a novel combined domain bi-abductive technique, capable of solving bi-abduction problems in the shape and ordering domain in a single analysis phase, will be described. The system, based upon an algorithmic search over a set of specialised inference rules, utilised unfold-and-match reasoning to reduce the input entailment until either a decision as to the validity of the entailment can be made, or some inference operation can permit the process to continue. This system was implemented as an automated tool and evaluated over a range of examples taken from the SL-COMP competition [Sighireanu et al., 2019b]. The work detailed in this chapter also formed the basis of an earlier conference publication [Curry et al., 2019].

The organisation of this chapter is as follows: the motivations behind this work may be found in Section 4.1, with the developed bi-abductive inference system detailed in Sections 4.2 through 4.6. Finally, we briefly summarise and conclude the chapter in Section 4.7.

For further reference, the language utilised throughout the system is presented and discussed in the preliminaries (Section 2.5). The evaluation of the system, as well as the details and results of the experimentation, may be found in Section 7.2 in the Evaluation chapter, where they will be discussed and further compared against the other systems and tools developed during this project.

4.1 Motivation

Bi-abductive Inference is a powerful analysis technique, most commonly used to support program verification techniques based on the use of entailments. First presented in [Calcagno et al., 2009], bi-abductive inference - bi-abduction for short - operates by identifying any parts of program state that are "missing" or "extra" from the entailments supplied to the system. These fragments, the *anti-frame* and *frame* respectively, describe constraints and spatial terms that are absent in the precondition, or carry forwards beyond the source of the entailment, potentially forming part of the eventual postcondition. This in turn allows for the use of so-called "small specifications" [Calcagno et al., 2011], and thus supports compositional analyses in systems that aim to make use of them.

Though bi-abduction has seen much success in the field of program verification and analysis [Calcagno and Distefano, 2011], the technique typically presents a number of key limitations, with the expressiveness of the technique being of the most interest to this work. In essence, most of the existing bi-abductive techniques known to the author do not have the capability to infer pure constraints, such as ordering, size, or contents properties, and are restricted to the inference of (dis)equalities and spatial information. While these properties have a less direct impact on the memory-safety of a program, such constraints can still have a significant impact; a commonly presented scenario in this area is a method that accesses a fixed number of nodes running over the end of a data structure due to the structure being too small for the operation. Such methods may be identified as safe from a spatial-only perspective, with the fault not being identified without consideration of the pure constraints involved. Due to the potential benefits that pure inference could bring to bi-abductive inference techniques, we began to investigate how such a technique may be developed, and what advantages such a technique may possess.

In this chapter, we present our novel technique to solve bi-abductive entailment problems with shape and ordering properties over singly-linked lists and binary trees in a single analysis phase; a so-called "combined domain" bi-abductive

technique. This technique follows in the style of Smallfoot [Berdine et al., 2005b], making use of unfold-and-match reasoning to reduce entailments and identify fragments both missing and extra from entailments. The main contributions of this work are thus:

- A bi-abductive system for the combined shape and ordering domain, featuring novel single-phase properties.
- A prototype automated implementation of the technique, evaluated over a range of benchmarks sourced from the SL-COMP competition.

4.2 System Design

The bi-abductive system consists of two key components: a set of inductive inference rules and a search algorithm operating over them. The rules follow in the design style of the Smallfoot tool [Berdine et al., 2005b], consisting of inference rules that aim to identify a fragment of the entailment that meets some condition and either transforming the fragment or removing it in order to advance the construction of a proof tree. This rule application also has a possibility of identifying some aspect of the frame or anti-frame, which will be recorded and propagated throughout the proof as appropriate. Once completed, the proof tree is used to generate the final output and also serves a useful witness for the system, allowing for validation of results once the analysis is complete.

In the current version of the system, the rule-set is organised into three distinct groups, corresponding to the intended purpose of the rules design: Normalisation, Match-and-Subtract, and Inference. Each of these groups are designed to accomplish a specific sub-task, either normalising the entailment, reducing the entailment by eliminating specific constraints, or inferring fragments of the frame and anti-frame. Though all of these rules are sound when applied correctly, there is a chance that applying a rule at the wrong time will cause a loss in precision, and subsequently cause the proof to fail. As an example, many of the rules targeting specific shape predicates will only operate in a sound manner when the

entailment is in normal form, with premature applications potentially missing key reductions or refinements. In order to avoid this, we developed the second major component of the system, the search algorithm.

The search algorithm that forms the second major component of the system is designed to minimise the loss of precision caused by the application of our inference rules. The algorithm searches over the set of rules in a specific, carefully arranged order, aiming to transform the entailment in a manner which preserves as much precision as possible, thus improving the overall quality of the final results. In essence, the algorithm searches over the rule-set, aiming to identify an inference rule that can be applied to the current state, applying the first appropriate match to the entailment. This rule application advances the construction of the proof, records the inferred fragments - if any are identified - and subsequently restarts the search, now targeting the newly-modified entailment or entailments. This process will repeat, continuing to refine and reduce the entailment - and any branched entailments produced - until such a time that a clear decision can be made to the validity of the current entailment, concluding the proof.

The individual rule-sets will be discussed in detail below: the normalisation rule set will be discussed in Section 4.3, the match-and-subtract rules will be discussed in Section 4.4 and the inference rules will be discussed in Section 4.5. The search algorithm will close the discussion in Section 4.6.

4.3 Normalisation

The Normalisation rules are the first set of rules to be searched over and applied to the entailment. These rules, applied repeatedly and exhaustively, aim to transform the entailment into the corresponding normal form, with most of the normalisation operations simply making explicit properties of the antecedent's symbolic heap. This normalisation process addresses a degree of ambiguity that may be present in the initial entailment, and allows for certain later rules to operate soundly without worry about loss of precision or restricting the search,

making it a critical and highly important stage of the proof search. It is also important to note that this normalisation process will likely be invoked at several points throughout the proof search, depending on the impact of later rules. Indeed, applying several of the later rules will produce terms that will trigger further normalisation.

4.3.1 Normal Form

To begin the discussion, the normal form utilised in this system will be introduced. For the sake of readability and conciseness, we introduce and utilise a small number of shorthand symbols to represent a range of values. $op(E)$ is used to represent the set of spatial terms used in our symbolic heap fragment, namely $E{\mapsto}[\rho]$, $ls(E,F)$, $tree(E)$, $sls(E,I,J,F)$ and $stree(E,I,J)$. This symbol is used to simplify the normal form definition, as well as provide a reference to the "root" of the term, which is necessary to define key formulae in certain conditions.

Alongside $op(E)$, there is a corresponding function, $G(op(E))$, which maps a given spatial term to a *guard* for that term. These guards are essentially pure constraints that, when considered alongside the spatial term, will ensure that the term does not reduce to `emp`. The function is defined as a fixed set of mappings as follows:

Definition 1 (Operator Guards) *The Guard, G of an operator $op(E)$, is directly defined as follows:*

$$
\begin{array}{lcl}
G(E{\mapsto}[\rho]) & \stackrel{def}{=} & \mathtt{true} \\
G(ls(E,F)) & \stackrel{def}{=} & E{\neq}F \\
G(sls(E,I,J,F)) & \stackrel{def}{=} & E{\neq}F \\
G(tree(E)) & \stackrel{def}{=} & E \neq \mathtt{null} \\
G(stree(E,I,J)) & \stackrel{def}{=} & E \neq \mathtt{null}
\end{array}
$$

The presence of a guard for a spatial term can have significant influence on the progress and direction of the proof, as the presence of a guard restricts a number of possible evaluations from occurring by invalidating any entailment

that breaches the non-empty property. Though the derivation of the guards for points-to terms is obvious, with the non-empty properties of a such a term self-evident, the derivation of guards for the shape predicates is more complicated. For lists, the guard is derived from the pure constraints defining the recursive case, as in the base case, there is neither an assigned node, nor a continuing predicate to satisfy the non-empty property. Thus, a term ensuring the inequality of the head and tail of the list is sufficient to act as the guard, both in the case of simple singly-linked lists and sorted singly linked lists, due to the presence of such terms invalidating the base cases. In the case of trees, the guard follows the same design, relying on the pure constraints of the recursive case to invalidate the base case, resulting in a guard which simply requires the root variable to be non-null.

With this in mind, the normal form is defined as follows:

Definition 2 (Normal Form) *A formula $\Pi \wedge \Sigma$ is in normal form (NF) if:*

1. $op(E) \in \Sigma \implies G(op(E)) \in \Pi$.
2. $op(E) \in \Sigma$ *and* $G(op(E)) \in \Pi \implies E{\neq}\texttt{null} \in \Pi$.
3. $op_1(E_1) * op_2(E_2) \in \Sigma$, $G(op_1(E_1)) \wedge G(op_2(E_2)) \in \Pi \implies E_1{\neq}E_2 \in \Pi$.
4. $E_1{=}E_2 \notin \Pi$.
5. $E{\neq}E \notin \Pi$.
6. Π *is satisfiable.*

If a formula Δ is in normal form, and for any s and h such that $s, h \models \Delta$, Δ is uniquely defined by s. Finally, for the purposes of our bi-abduction system, we say that a bi-abductive entailment is in normal form if its antecedent is in normal form.

4.3.2 Normalisation Rules

In order to ensure that the bi-abductive entailments are normalised for the subsequent reduction and refinement stages of the proof search, we developed a number of inductive proof rules that will transform a given entailment into its normalised form. For each condition, there are a number of possible checks and relevant rules that will further transform the supplied bi-abductive entailment into the corresponding normalised form. Important to remember is that this normalisation process will be applied repeatedly throughout the proof search, as all the subsequent rules operate under the expectation that the entailment is normalised.

Condition 1 Condition 1 aims to ensure that all spatial terms present within the normalised entailment are also guarded, ensuring that such terms cannot be reduced to `emp`. This condition also implies that any spatial term that is not guarded should be reduced to `emp` and removed. Though it would be trivial to introduce guards for all terms present within the entailment, such an operation would also eliminate otherwise valid configurations of the heap, when applied to predicates. Similarly, failing to introduce the guard for predicates would require the elimination of such terms from the entailment, which could cause otherwise valid entailments to be determined to be invalid. In order to resolve this impasse, we apply the EXCLUDE-MIDDLE rule.

$$[\mathbf{EXCLUDE-MIDDLE}]$$
$$\frac{(E_1{=}E_2 \wedge \Delta_1) * [M_1] \vdash \Delta' * [F_1] \qquad (E_1{\neq}E_2 \wedge \Delta_2) * [M_2] \vdash \Delta' * [F_2]}{\Delta * [M_1 \wedge M_2] \vdash \Delta' * [F_1 \vee F_2]}$$
$$(\text{where } FV(E_1, E_2) \subseteq (FV(\Delta_1) \cup FV(\Delta_2)))$$

The EXCLUDE-MIDDLE rule targets shape predicates that do not already have a guard present in the entailment and produces a branch in the proof tree. In one branch, the entailment is extended with the guard corresponding to the selected predicate, allowing for the continued presence of the predicate in the normalised entailment. In the other branch, the entailment is transformed in such a way that the predicate is forced into a configuration that satisfies the base case; in other words, the predicate is made to unfold to `emp`, removing it from the entailment

altogether. The predicate is not directly removed in this scenario in order to fully evaluate the constraints introduced by the base case over the remainder of the entailment. Such operations are often performed by the various PRED-LBASE rules. As all of the cases are fully and independently explored, splitting the case in this way does not introduce soundness issues, as all of the configurations are represented in the completed proof.

$$\frac{(\Delta) * [M] \vdash \Delta' * [F]}{(\Delta * ls(E,E)) * [M] \vdash \Delta' * [F]}\ [\mathbf{LS{-}LBASE}] \qquad \frac{(\Delta) * [M] \vdash \Delta' * [F]}{(\Delta * tree(\texttt{null})) * [M] \vdash \Delta' * [F]}\ [\mathbf{TREE-LBASE}]$$

$$\frac{(\Delta[V'/V]) * [M] \vdash \Delta'[V'/V] * [F]}{(\Delta * sls(E,V,V',E)) * [M \wedge V = V'] \vdash \Delta' * [F \wedge V = V']}\ [\mathbf{SLS{-}LBASE}]$$

$$\frac{(\Delta[V'/V]) * [M] \vdash \Delta'[V'/V] * [F]}{(\Delta * stree(\texttt{null},V,V')) * [M \wedge V{=}V'] \vdash \Delta' * [F \wedge V{=}V']}\ [\mathbf{STREE-LBASE}]$$

The PRED-LBASE rules are themselves relatively simple, firing when a shape predicate has been parametrised in such a way that the base case is the only possible unfolding. These rules replace the predicate with that base case, eliminating the shape predicate and replacing it with the corresponding pure constraints, simplifying and progressing the entailment.

Condition 2 Condition 2 aims to make explicit the fact that any non-empty spatial term must have a non-`null` root. In order to further transform the entailment into its normal form, the system introduces non-`null` constraints via the use of the NODE-EX rule.

$$\frac{(G(op(E)) \wedge E{\neq}\texttt{null} \wedge \Delta * op(E)) * [M] \vdash \Delta' * [F]}{(G(op(E)) \wedge \Delta * op(E)) * [M] \vdash \Delta' * [F]}\ [\mathbf{NODE{-}EX}]$$
(where $E{\neq}\texttt{null}{\notin}\Delta$)

The NODE-EX rule identifies guarded spatial terms which do not have a corresponding non-null constraint over the root variable and introduces such a term into the entailment. While this rule introduces new constraints, they do not form part of the frame or anti-frame due to the fact that such constraints are implicit under

the guarded predicate principle: if a spatial term cannot reduce down to empty, it must be assigned and thus cannot be `null`.

NODE-EX is one of the earliest rules examined in our proof search algorithm, ensuring that non-null properties are made explicit early in the proof process and ensures the information is properly considered throughout.

Condition 3 Condition 3 is a formalised pure representation of the separation properties of the separation conjunction operator used for spatial terms. More specifically, any two non-`emp` spatial terms must be distinct in order for the spatial terms to hold. In order to make these constraints explicit, the NODES-EX rule is applied.

$$\frac{(G(op_1(E_1)) \wedge G(op_2(E_2)) \wedge E_1{\neq}E_2 \wedge \Delta * op_1(E_1){*}op_2(E_2)) * [M] \vdash \Delta' * [F]}{(G(op_1(E_1)) \wedge G(op_2(E_2)) \wedge \Delta * op_1(E_1){*}op_2(E_2)) * [M] \vdash \Delta' * [F]} \; [\mathbf{NODES{-}EX}]$$
$$(\text{where } E_1{\neq}E_2{\notin}\Delta)$$

The NODES-EX rule identifies pairs of guarded spatial terms and constructs a inequality between the root variables of the two, introducing it into the entailment. As with the NODE-EX rule, this new constraint is not introduced into the inferred fragments as it is naturally implied by both the separation conjunction operator and the guards attached to the spatial terms.

Condition 4 Condition 4 seeks to ensure that variables are represented consistently throughout the entailment and proof search. In order to accomplish this, any concrete equalities are applied to the entailment through the use of the SUBSTITUTION rule.

$$\frac{(\Pi \wedge \Sigma)[E/x] * [M] \vdash \Delta[E/x] * [F]}{(\Pi \wedge x{=}E \wedge \Sigma) * [M \wedge x{=}E] \vdash \Delta * [F \wedge x{=}E]} \; [\mathbf{SUBST}]$$

The SUBSTITUTION rule fires when an equality is found in the antecedent of the target entailment. The rule selects one of the variables and applies a renaming to the both the antecedent and consequent, ensuring that all instances of the two

variables are updated to be represented by only one of them. This transformation ensures that any aliased variables, where the alias is known, are represented by a single consistent symbol, allowing for a more effective and precise proof search. The SUBSTITUTION rule also makes a record of the alias by introducing the equality into both the frame and anti-frame, ensuring that the equality is preserved throughout the inference.

In addition to the SUBSTITUTION rule, there is also a special-case rule for equalities, LEFT-IDENTITY=.

$$\frac{[\textbf{LIDENT} =]}{\dfrac{(\Delta) * [M] \vdash \Delta' * [F]}{(\Delta \wedge E{=}E) * [M] \vdash \Delta' * [F]}}$$

The LEFT-IDENTITY= rule, referred to as LIDENT= going forwards, is applied when there is a trivial equality in the antecedent, removing the equality without recording it in the inferred fragments. While there is little semantic difference between LIDENT= and SUBSTITUTION, LIDENT= is a slightly more efficient application of the process, taking advantage of the fact that no renaming would actually be necessary in such a case.

Condition 5 and 6 Conditions 5 and 6 are conditions which help to enforce the satisfiability of the pure components of the entailment. More specifically, condition 5 aims to prevent the inclusion of trivial inequalities in the entailment, seeking to ensure that the antecedent remains valid throughout the process. This condition essentially defines a failure state for the entailment, as the antecedent resolving to `false` indicates that there is no valid solution.

Condition 6 is a more general representation of this fact, aiming to enforce the satisfiability of the pure constraints. While the non-satisfiability of the initial constraints would indicate the bi-abductive entailment cannot be made to hold, even under inference, the continuing satisfiability of the constraints is difficult to enforce. Indeed, there are no rules tied to conditions 5 and 6 as a result of this, with condition 5 being checked semantically and condition 6 expected to be resolved through the use of an SMT solver such as [de Moura and Bjørner, 2008].

4.4 Match and Subtract

Following the completion of the normalisation process, the proof search begins to search for an appropriate rule in the Match and Subtract rule sequence (presented in Section 4.4). These rules, as the name suggests, aim to identify parts of the antecedent's heap that match with parts of the consequent, and either remove or reduce those fragments, progressing the search by making the target entailment "smaller". This rule-set also includes a small number of decision rules, specialised proof rules that examine the current state of the sub-goal, and if appropriate, end the search with some final inferences or transformations. These rules, alongside the normalisation rules, are restricted to working with the information directly available in the entailment; no inference is performed during the application of these rules, though some information may be made explicit where appropriate.

Due to the design of the system, these rules are expected to be applied on fully normalised entailments only, with many rules taking advantage of the design of the normal form in order to perform specific operations. One of the most common instances where this occurs is with the rules which feature unfolding, leveraging the properties of normal form to guarantee specific outcomes.

4.4.1 Rules

We will now discuss the rules in greater detail. The subtraction rules can be divided into three broad groups: **decision** rules, **subtraction** rules and **generalization** rules. Each of these groups aim to accomplish a specific subgoal of the larger rule sequence, whether that is the identification of states where a decision can be made as to whether the entailment holds or the reduction of the entailment.

Decision Rules

The group of decision rules in the subtraction sequence aim to identify scenarios where a clear decision can be made about whether the entailment holds at the current subgoal. The two rules forming this group, EMP and IDENT, will both detect heap configurations where the entailment can be seen to hold and end the proof search over that particular subgoal. The primary difference between the two rules are the scenarios in which they apply.

IDENT

$$\frac{}{\Delta * [\texttt{true} \wedge \texttt{emp}] \vdash \Delta * [\texttt{true} \wedge \texttt{emp}]} [\textbf{IDENT}]$$

The Identity rule, or IDENT rule for short, is a decision rule that fires when the antecedent and consequent are identical. In such a scenario, it is readily apparent that the entailment will hold and the IDENT rule acts as the final applied rule in that branch of the proof tree, ending the search over that subgoal.

The rule also provides a base for the anti-frame and frame fragments in the proof: each instance in which there is a fragment of the frame or anti-frame inferred, that fragment is defined in a manner following the form of $\Pi \wedge ?M \wedge \Sigma$. These individual inferred fragments are all defined as series of continuing terms, building inferred fragment by inferred fragment until the final value of the frame or anti-frame can be inferred completely. The IDENT rule and EMP rule both act in such a manner, inferring the final fragments of the anti-frame for that branch of the proof search, acting as the base value of the fragment and allowing for the complete values to be determined by propagating forward (or back) the individual fragments.

EMP

$$\frac{}{\Pi \wedge \texttt{emp} * [\texttt{true} \wedge \texttt{emp}] \vdash \texttt{true} \wedge \texttt{emp} * [\Pi \wedge \texttt{emp}]} [\textbf{EMP}]$$

The EMP rule applies in scenarios where both the consequent has been reduced to

$\texttt{true} \wedge \texttt{emp}$ and the antecedent has no remaining spatial terms. In these cases, the consequent has been reduced to a state where all constraints and terms have been satisfied, with only the pure constraints of the antecedent remaining. As with the IDENT rule, this state implies that the current state of the entailment holds, though there may remain some pure constraints. The EMP rule infers that the entailment will hold completely if the remaining pure constraints are represented in the consequent, identifying the remaining constraints as part of the frame and ending the proof search over that branch.

Subtraction

The majority of the Match and Subtract rules of the system fall into the second group, subtraction rules. The primary purpose of these rules is to reduce the entailment through the subtraction of matching terms, utilising some minor transformations in order to simplify or allow for this. These rules typically identify some explicit match between terms in the antecedent and consequent and remove one or both of those terms, reducing the size of the entailment and aiming to reduce both sides to $\texttt{true} \wedge \texttt{emp}$.

The overview of the subtraction rules are as follows: the system detects a scenario in which there is a match between terms in the antecedent and consequent, with identical terms and terms that indicate overlap between data structures and other spatial terms both being the primary source of such matches. The system then checks the side conditions of the rules in order to identify whether the rule is safe to apply for that particular match, and if so, the system applies the rule, progressing the proof search and restarting the search at the normalisation rules.

The primary difference between the rules in this group is the type of match that is detected, with the majority of the rules being crafted to specifically handle matches that feature a single type of shape predicate, though there remains a small number of rules that target more general operations. In addition to these predicate-focussed rules, a small number of rules that perform relatively elementary reductions over the consequent, such as eliminating predicates that equate

to `emp`, are also present in this group.

Right Identity

$$[\textbf{RIDENT} =]$$
$$\frac{\Delta * [M] \vdash \Delta' * [F]}{\Delta * [M] \vdash \Delta' \wedge E{=}E * [F]}$$

The RIDENT= rule is the counterpart to the LIDENT= rule present in the Normalisation rule sequence (Section 4.3) and aims to accomplish the same general goal of eliminating trivial equivalences. The mechanics of this rule are functionally identical to that of LIDENT=, and is one of the earliest rules examined in the Match and Subtract stage of the proof search.

Hypothesis

$$[\textbf{HYPOTHESIS}]$$
$$\frac{\Pi \wedge \Sigma * [M] \vdash \Pi'' \wedge \Sigma' * [F]}{\Pi \wedge \Sigma * [M] \vdash \Pi' \wedge \Pi'' \wedge \Sigma' * [F]}$$
$$(\text{where } \Pi \Rightarrow \Pi')$$

The HYPOTHESIS rule is one of the more elemental rules present in the subtraction rule sequence, aiming to reduce the consequent via the elimination of pure constraints that are already implied by the constraints of the antecedent. The rule identifies a sub-formula of the consequent that is implied by the constraints present in the antecedent and eliminates them from the consequent. In order to ensure that the constraints are preserved for future rules, the constraints in the antecedent are not eliminated and remain in the entailment going forwards.

Separation Conjunction Introduction

$$[*{-}\textbf{INTRODUCTION}]$$
$$\frac{\Sigma * [M_1] \vdash \Sigma' * [F_1] \qquad \Pi \wedge \Sigma_1 * [M_2] \vdash \Pi' \wedge \Sigma_2 * [F_2]}{\Pi \wedge \Sigma * \Sigma_1 * [M_1 * M_2] \vdash \Pi' \wedge \Sigma' * \Sigma_2 * [F_1 * F_2]}$$

Separation Conjunction Introduction (*-INTRO from this point forwards) is one of the most important rules in our proof system. This rule identifies sub-heaps from the antecedent and consequent such that the antecedent sub-heap entails the consequent sub-heap. The rule then separates those sub-heaps into a new

branch of the proof tree, creating a reduced sub-goal to continue the search. These reductions are one of the most effective ways of eliminating matches from the entailment and is also one of the most simple. $*$-INTRO can fire whenever an appropriate match is found, performs no inference and has no side conditions, ensuring it can be applied easily.

The rule does have one particular aspect of note however, which is the effect it has upon the inferred frame and anti-frame. In essence, the split in the proof causes the introduction of a split in the inferred fragments, caused by the potential for inference to occur through the proof of both new sub-goals. These split frames and anti-frames both refer to the same unified entailment and are recombined once the proof search concludes.

R-EX

$$\frac{[\mathbf{R-EX}]}{\Delta * [M] \vdash \Delta'[V'/V] * [F]}{\Delta * [M] \vdash \exists V.\ \Delta' * [F]}$$
$$(\text{where } V' \in FV(\Delta))$$

At several points throughout the proof, there may be a scenario in which existential variables are encountered in the consequent. In these cases, it may be beneficial to concretise these variables in such a way that it permits the proof search to continue more effectively than with the existential variables. In order to do this, the R-EX rule is applied.

The R-EX rule is fired whenever the consequent contains a existential variable and aims to identify an instantiation for that variable which enable the proof search to continue. This rule can replace the existential variable with any free variable present in the symbolic heap of the consequent, substituting all instances of the existential variable with the selected concrete variable taken from the consequent.

Predicate Base

$$\frac{\Delta * [M] \vdash \Delta' * [F]}{\Delta * [M] \vdash \Delta' * ls(E, E) * [F]}\ [\mathbf{LS{-}BASE}] \qquad \frac{(\Delta) * [M] \vdash \Delta' * [F]}{\Delta * [M] \vdash \Delta' * tree(\texttt{null}) * [F]}\ [\mathbf{TREE - BASE}]$$

$$\frac{\Delta * [M] \vdash \Delta'[V'/V] * [F]}{\Delta * [M] \vdash \Delta' * sls(E, V, V', E) * [F \wedge V = V']}\ [\mathbf{SLS{-}BASE}]$$

$$\frac{\Delta * [M] \vdash \Delta'[V'/V] * [F]}{\Delta * [M] \vdash \Delta' * stree(\texttt{null}, V, V') * [F \wedge V{=}V']}\ [\mathbf{STREE - BASE}]$$

The predicate base rules, LS-BASE, SLS-BASE, TREE-BASE and STREE-BASE, aim to identify and eliminate predicates in the consequent that will reduce to `emp`. Much like their normalisation equivalents, PRED-LBASE, the rules identify predicates in hte consequent whose parameters indicate that the only viable branch of the predicate is the base case. The rule subsequently eliminates the predicate and introduces the remaining pure constraints back into the consequent.

Predicate Recursion The predicate-targeting Recursive rules aim to identify and resolve scenarios in which a predicate in the consequent is found to share a root node with a spatial term in the antecedent. These matches indicate that the antecedent at least partially satisfies the predicate in the consequent and aims to eliminate that concrete state while preserving the notion that more may be necessary to fully satisfy the constraints. These Recursive rules, or REC rules, have two forms: one form, REC (Node), targets points-to terms that match with the predicate in the consequent and is common to all predicates. The second form is REC, with variants existing for lists and sorted lists only.

$$\frac{\Delta * [M] \vdash \Delta' * ls(E_2, E_3) * [F]}{\Delta * E_1{\mapsto}[E_2] * [M] \vdash \Delta' * ls(E_1, E_3) * [F]}\ [\mathbf{LS - REC(Node)}]$$
$$(\text{where } E_1{\mapsto}[E_2] \notin \Delta')$$

$$[\mathbf{TREE-REC(Node)}]$$
$$\frac{\Delta * [M] \vdash \Delta' * tree(l) * tree(r) * [F]}{\Delta * E{\mapsto}[l, r] * [M] \vdash \Delta' * tree(E) * [F]}$$
$$(\text{where } E{\mapsto}[l, r] \notin \Delta')$$

$$[\mathbf{SLS-REC(Node)}]$$
$$\frac{\Delta * [M] \vdash \exists V_2.\ V_1{\leq}V_2 \wedge \Delta' * sls(E_2, V_2, V_3, E_3) * [F]}{\Delta * E_1{\mapsto}[E_2, V_1] * [M] \vdash \Delta' * sls(E_1, V_1, V_3, E_3) * [F]}$$
$$(\text{where } E_1{\mapsto}[E_2, V_1] \notin \Delta')$$

$$[\mathbf{STREE-REC(Node)}]$$
$$\frac{\Delta * [M] \vdash V_1{\leq}V_2 \wedge V_2{\leq}V_3 \wedge \Delta' * stree(l, V_1, V_2) * stree(r, V_2, V_3) * [F]}{\Delta * E_1{\mapsto}[l, r, V_2] * [M] \vdash \Delta' * stree(E_1, V_1, V_3) * [F]}$$
$$(\text{where } E_1{\mapsto}[l, r, V_2] \notin \Delta')$$

The REC (Node) form, as mentioned, is applied when a node in the antecedent shares a root with a predicate in the consequent and the points-to node does not appear in the consequent separately. In such a case, it can be safely assumed that the predicate in the consequent is non-empty, which in turn implies that the predicate can be unfolded into the recursive branch without issue. This process is essentially summarised in the REC (Node), which unfolds a node (with matching parameters) from the predicate in the consequent, matches the two nodes and subsequently eliminates them from the entailment, leaving the recursive continuation of the shape predicate in the consequent as a remaining constraint to be satisfied, along with any pure constraints identified from the unfolding.

$$[\mathbf{LS{-}REC}]$$
$$\frac{\Delta * [M] \vdash \Delta' * ls(E_2, E_3) * [F]}{\Delta * ls(E_1, E_2) * [M] \vdash \Delta' * ls(E_1, E_3) * [F]}$$
$$(\text{where } E_3 \text{ is dangling } \&\ ls(E_1, E_2) \notin \Delta')$$

$$[\mathbf{SLS{-}REC}]$$
$$\frac{\Delta * [M] \vdash \exists V_2.\ V'{\leq}V_2 \wedge \Delta' * sls(E_2, V_2, V_3, E_3) * [F]}{\Delta * sls(E_1, V_1, V', E_2) * [M] \vdash \Delta' * sls(E_1, V_1, V_3, E_3) * [F]}$$
$$(\text{where } E_3 \text{ is dangling } \&\ sls(E_1, V_1, V', E_2) \notin \Delta')$$

The REC rules can be seen as a specialisation of the corresponding REC (Node) rules. The two REC rules, LS-REC and SLS-REC, are fired when there is a shared root node between a predicate in the consequent and a predicate of the same type in the antecedent, though as with the REC (Node) rules, there is a further side condition preventing the rules being applied if the predicate appears separately in the consequent. This scenario, while appearing relatively simple, is in actuality a complex problem. Since this rule is examined after $*$-INTRO, it can be typically assumed that the predicates are not identical, and as such, it is likely that there will be a difference in the length of the predicates. Such matches can produce three outcomes: the predicate of the antecedent is shorter in length, resulting in a remaining continuing predicate in the consequent; the predicate in the consequent in smaller, resulting in a continuing fragment in the antecedent; or both predicates are equivalent, eliminating both, though potentially requiring the identification of equivalences to unify the predicates. For the purposes of our system, we assume that the first scenario is the one that is accurate, resulting in the consequent retaining some fragment of the predicate after the rule is applied. This decision is primarily a practical one: by assuming the current state is insufficient, we produce a safer precondition, as the remaining fragment, if not resolved by existing terms in the antecedent, will eventually be identified as a fragment of the anti-frame.

These outcomes are only possible to describe when dealing with list-based predicates, and more specifically, list *segments*. The presence of a clear end-point is the key enabler of these operations, as it provides a surface from which the shape predicate can be continued. While the logic underpinning this rule is shared with other predicates, it is more difficult to provide a reasonable cut point when dealing with non-segmented predicates such as trees. As a result of this, only LS-REC and SLS-REC are supported in this version of the system.

Generalisation

$$\frac{V \leq V' \wedge \Delta * [M] \vdash \Delta' * ls(E_2, E_3) * [F]}{\Delta * sls(E_1, V, V', E_2) * [M] \vdash \Delta' * ls(E_1, E_3) * [F]} \text{ [LS – GENERALIZE]}$$
(where $sls(E, V, V', E') \notin \Delta'$)

$$\frac{V \leq V' \wedge \Delta * [M] \vdash \Delta' * [F]}{\Delta * stree(E, V, V') * [M] \vdash \Delta' * tree(E) * [F]} \text{ [TREE – GENERALIZE]}$$
(where $stree(E, V, V') \notin \Delta'$)

Due to the direct knowledge of the relation between basic singly-linked list segments and sorted singly-linked list segments (and similarly between binary trees and binary search trees), it is possible for our system to generalise sorted shape predicates into their shape-only counterpart, where needed.

This capability is a specialised operation enabled by the explicit knowledge of the relationship between the simple and sorted variations of the same general shape structure. Specifically, the design of the sorted variant of the list and tree predicates is such that the spatial terms of the sorted variant is sufficient to satisfy entailments expecting the non-sorted variant.

As an example, consider a list node $x \mapsto y$, or to show it's complete representation, $x \mapsto [next : y, \rho']$. In this more complete representation, it can be seen both that the *next* field (the field expected by our ls definition) and ρ' have been made explicit, though such aspects are always present. While the *next* field is expected, the ρ' term is of more interest here. Essentially, ρ' is the continuation of the record stored at the location pointed to by x, representing further potential field-value pairs that have been omitted for the sake of simplicity or are not yet made explicit in the reasoning. Such an continuation is necessary in the majority of these scenarios, as a list structure is used for data storage; any list comprising of nodes in which only the *next* pointer is recorded would have little practical use. This continuation carries a further implication: assigned nodes may be of different "sizes" in the entailment. In such a case, given that the larger node possesses the same root and field-value pairs, it is sufficient to match with the smaller node.

Given this relationship between nodes, it becomes possible to "reduce" a more complex node into a simpler form, abstracting away any fields that are not necessary to satisfy newer form. As a result of this, a node produced from the unfolding of a sorted list predicate (such as $x{\mapsto}[next : y, value : i, \rho']$) can be used to satisfy a node produced from a simple linked list ($x{\mapsto}[next : y, \rho'']$) as all the requisite information remains represented. It is important to note that this relation is one-way: antecedent to consequent only. This is due to the fact that a smaller node would be insufficient to satisfy a larger node, and weakening the consequent would invalidate the search as a whole.

Though this property is already represented in $*$-INTRO, that capability is limited to matching nodes only; the $*$-INTRO rule cannot directly handle entailments of the form $sls(E, V, W, F) \vdash ls(E, F)$. In order to support this behaviour for shape predicates, we created two rules LS-GENERALISE and TREE-GENERALISE. These rules detect when a sorted list or sorted tree in the antecedent match with a simple list or tree in the consequent. In such a case, the generalise rules abstract the sorted shape predicate into the corresponding simplified version, retaining some aspect of the ordering properties in order to preserve those implicit constraints, though the individual relations between nodes is lost. The rules then proceed to eliminate the now-matching predicates, though the lists are reduced in the same manner as LS-REC, taking advantage of the segment structure to identify whether some fragment of the list would remain after this operation.

Together, the normalisation and subtraction rules are sufficient to act as a basic entailment prover for the separation logic fragment supported by our system. Should the entailment require no inferred terms to be introduced in order to ensure that the entailment holds, there will be no need to progress beyond the Match and Subtract rule sequence, as the entailment will have been reduced to a point of decision by those two rule sequences alone. However, should the entailment not hold in its initial form, the search algorithm will proceed onwards to the final rule sequence, the Inference rules.

4.5 Inference

When the normalisation and subtraction rules are insufficient to prove the entailment, the system attempts to find an applicable rule in the set of inference rules. While the previous rules essentially aim to prove the entailment through the manipulation of the information already present in the entailment, the inference rules aim to progress the proof through the inference of new information. This newly-inferred fragment of the frame or anti-frame is then leveraged to further reduce the entailment and hopefully enable the normalisation and subtraction rules to continue in the process. These inference rules are where the vast majority of the inference takes place in the system, with every rule included in this set inferring some fragment of the frame or anti-frame. These fragments are inferred and recorded alongside the rule application in the constructed proof structure, and are gathered together at the conclusion of the proof.

As with the other rule groups in the system, the system will restart the search after an inference rule is applied, beginning the next search in the normalisation group. While this approach is shared with both the normalisation and subtraction rules, this "restart" has an additional benefit for the inference rules: *inference operations are reserved as a last resort.* By ensuring that the inference rules are applied when the entailment is fully normalised, and does not include a matching sub-heap to be removed, the inferred fragments are essentially necessary for the completion of the proof, with the design of each of the inference rules aiming to ensure the fragments identified are also as minimal as possible. These two design principles encourage the system to produce higher-quality outputs for the frame and anti-frame, while preserving the core functionality.

4.5.1 Rule Design

Generally, the inference rules follow the general design and style of the prior subtraction rules, aiming to identify and resolve matches between terms in the antecedent and the consequent through the manipulation and/or subtraction of

those matching terms. As mentioned above, however, the inference rules differ in that they are typically anticipated to infer some new fragment of the frame or anti-frame in order to perform these operations. For the majority of the inference rules, these inferences describe conditions that must be enforced for the relation outlined by the rule to hold: pure conditions that enforce non-emptiness of a list, or an equality that must be constant throughout the proof in order to preserve the soundness of the system, though a small number of the more powerful rules can infer more general terms.

While almost all of the rules in the system were designed to operate with a small effect, it is the inference rules where this design intent is most significant. Each of the rules in the inference group are designed to infer only a minimal amount of new information, enough to perform a relatively basic advance in the proof and no more. Though this is likely to introduce a cost to performance over inference rules that could have a more significant impact, these small advances aim to encourage the use of the subtraction rules over the inference rules, prioritising the careful manipulation and reduction of existing information over the inference of new information. As a result of these design decisions, the inference rules should infer only information that is necessary for the continuation of the proof, and only the minimal amount of information required to do so. This principle helps to ensure that the final output for the values of the frame and anti-frame are as close to minimal as possible, without further refinement.

Each of the inference rules in the system follow these design principles, though each operate in very different manners. In order to provide a more useful discussion as to the design intent and function of the rules, they will now be discussed in further detail.

Infer Pure

$$[\textbf{INF}-\textbf{PURE}]$$

$$\frac{}{(\Pi \wedge \texttt{emp}) * [\Pi' \wedge \texttt{emp}] \vdash \Pi' \wedge \texttt{emp} * [\Pi \wedge \texttt{emp}]}$$

(where $\Pi \wedge \Pi'$ is satisfiable.)

The INF-PURE rule is the final axiom of our core system, and aims to conclude

the proof search once the entailment has been reduced to pure constraints only. In such scenarios, all spatial fragments of the bi-abductive problem have been matched and removed, and any pure constraints still present in the consequent are those that are not implied by the pure constraints of the antecedent, due to the lack of applications of previous rules required for the inference rules to be considered. As a result, the remaining pure constraints of the consequent are constraints that are required, but not enforced by the antecedent, implying that they are previously unidentified fragments of the anti-frame. Similarly, any remaining constraints in the antecedent represent constraints and information that are not acknowledged in the consequent of the entailment, marking them as a fragment of the frame. The INF-PURE rule, when applied, formalises this reasoning, recording all remaining pure constraints as part of the frame or anti-frame as appropriate and eliminating them from the entailment, reducing the entailment to $\texttt{true} \wedge \texttt{emp} \vdash \texttt{true} \wedge \texttt{emp}$, concluding the search over that branch of the tree.

Infer Points-to

$$\frac{[\mathbf{INF}{-}{\mapsto}]}{(E_0{=}E_1 \wedge \Delta) * [M] \vdash \Delta' * [F]}{\Delta * E{\mapsto}[E_0] * [E_0{=}E_1 \wedge M] \vdash \Delta' * E{\mapsto}E_1 * [F]}$$

INF-POINTS-TO (INF-$\mapsto$ for short) is the first targeted inference rule examined by the proof search of our bi-abductive system, and aims to reconcile points-to terms in the antecedent and consequent by inferring equalities that will equate the values recorded in the fields. These equalities are introduced into the entailment and also recorded as part of the anti-frame, as while the immediate introduction of the equalities will allow the search to continue, such equalities must be preserved and represented throughout the entailment. The now-matching points-to terms are also eliminated from the entailment at this point.

This rule is deemed sound due to a simple reasoning: as the nodes are assigned to the same location, they must be equal for the entailment to be valid. As a result, any differences in field values indicates either that the two nodes are unequal, introducing an irrecoverable contradiction in the entailment, or that

the two nodes *are* equal, implying the fields to also be equivalent. Thus, if the nodes are in fact equivalent, the fields can be deemed to be equivalent and the constraints establishing this fact should be present in the entailment to establish this.

Predicate Inference

$$[\mathbf{INF-LS}]$$
$$\frac{E \neq E_1 \wedge \Delta * ls(X, E_1) * [M] \vdash \exists \overrightarrow{Y}.\Delta'}{\Delta * ls(E, E_1) * [E \neq E_1 \wedge M] \vdash \exists X \overrightarrow{Y}.\Delta' * E{\mapsto}[X]}$$

$$[\mathbf{INF-SLS}]$$
$$\frac{E_1 \neq E_2 \wedge V_1{\leq}V' \wedge \Delta * sls(X, V', V_2, E_2) * [M] \vdash \Delta' * [F]}{\Delta * sls(E_1, V_1, V_2, E_2) * [E_1 \neq E_2 \wedge V_1{\leq}V' \wedge M] \vdash \Delta' * E_1{\mapsto}[X, V_1] * [F]}$$

$$[\mathbf{INF-TREE}]$$
$$\frac{X{\neq}\texttt{null} \wedge \Delta * tree(E) * tree(F) \vdash \exists \overrightarrow{Y} \Delta'}{\Delta * tree(X) * [X{\neq}\texttt{null} \wedge M] \vdash \exists X \overrightarrow{Y} \Delta' X{\mapsto}[E, F]}$$

$$[\mathbf{INF-STREE}]$$
$$\frac{X{\neq}\texttt{null} \wedge V_3{\leq}V \wedge V{\leq}V_4 \wedge \Delta * stree(E, V_1, V_3) * stree(F, V_4, V_2) \vdash \exists \overrightarrow{Y} \Delta'}{\Delta * stree(X, V_1, V_2) * [X{\neq}\texttt{null} \wedge V_3{\leq}V{\leq}V_4 \wedge M] \vdash \exists X \overrightarrow{Y} \Delta' * X{\mapsto}[E, F, V]}$$

The predicate inference rules, INF-LS, INF-SLS, INF-TREE and INF-STREE, aim to match predicates in the antecedent with nodes in the consequent and produce a corresponding node to reduce the entailment. These rules all follow the same design principles, essentially acting as a targeted unfolding of the predicate, selectively substituting the selected predicate with the corresponding non-empty branch, ensuring that a matching node is produced. These nodes are subsequently removed, leaving the newly introduced pure constraints and continuing predicate fragments in the antecedent and the remainder of the consequent as the new sub-goal for that branch of the proof.

Due to the need to unfold a matching node from the predicate, an unfolding into the empty case cannot be permitted. While the rules do this mechanically, the predicate inference rules also infer a set of pure constraints which will invalidate any attempt at unfolding the predicate into the empty cases, adding them

to the anti-frame. These pure constraints are essentially the pure terms of the non-empty predicate branch, which are incompatible with the pure constraints of the empty branch of the same predicate. As a basic example, the INF-LS rule infers an inequality between the head and tail node, which would directly contradict with the empty case of the same predicate, which state that the head and tail parameters are equivalent. This ensures that the final resulting entailment will not permit the predicate to be empty, ensuring that the node can be unfolded safely, though such a configuration would be prevented by the earlier case split during the normalisation process.

Infer Missing and Extra

$$\textbf{[INF}-\textbf{MISSING]}$$
$$\frac{\Delta * [M] \vdash \Delta' * [F] \qquad \Delta * Q(E, E') \not\vdash \texttt{false}}{\Delta * [M * Q(E, E')] \vdash \Delta' * Q(E, E') * [F]}$$
(where $Q(E, E')$ is $op(E)$)

$$\textbf{[INF}-\textbf{EXTRA]}$$
$$\frac{\Delta * [M] \vdash \Delta' * [F] \qquad \Delta' * Q(E, E') \not\vdash \texttt{false}}{\Delta * Q(E, E') * [M] \vdash \Delta' * [F * Q(E, E')]}$$
(where $Q(E, E')$ is $op(E)$)

The final rules of our bi-abductive entailment proof system are the "bulk" inference rules, INF-MISSING and INF-EXTRA. These rules, unlike the prior inference rules, do not have a specific condition to activate. Instead, the rules will identify any spatial term in the entailment and move it to the inferred fragment corresponding to the opposite side of the entailment, with terms from the consequent being introduced to the anti-frame and terms from the antecedent introduced to the frame. These two inference rules are the final resort of our system, and aim to identify and resolve any spatial terms that cannot be further reduced by any of the prior rules, whether that is caused by a lack of appropriate matching terms in the other component of the entailment, or whether potential matches were prevented by side-conditions due to soundness concerns.

An important aspect of these two rules is that for many entailments that require inference, these rules are necessary; though many of the prior subtrac-

tion and inference rules can identify some of the necessary information, there is typically a need to infer the remaining fragments, which cannot be effectively achieved without these rules. As an example, the entailment $x{\mapsto}y \vdash ls(x, z)$ will reduce down to $\pi \wedge \texttt{emp} \vdash ls(y, z)$. In this case, the final remaining fragment of spatial information, $ls(y, z)$, cannot be reduced further and no other matches exist. As a result, the INF-MISSING rule is the only possible rule that can be used to advance the proof search, identifying the final spatial term, removing it and recording it as a fragment of the anti-frame.

However, these rules, while powerful, are also imprecise: they do not consider other spatial terms, nor do they consider pure constraints. As a result, these rules may eliminate important state information too early, if applied at the wrong time. In order to reduce the chances of this occurring, these rules are the final rules to be examined in the search. Moreover, the design principles of the search overall, namely the apply-then-reset search strategy, as well as the strict order in which the rules are examined, all aim to ensure that the INF-MISSING and INF-EXTRA rules infer the most minimal fragments possible throughout the search. As mentioned in the earlier chapters (4.2), this also improves the overall quality of the results.

These rules do include one restriction to their application, however. While any spatial fragment can be identified for inference operations, these fragments must not contradict the side of the entailment to which they are being inferred, i.e. a term from the consequent, if introduced to the antecedent, must not cause the antecedent to entail `false`. This check, while relatively simple in design, is simply aimed at preventing the inference of terms that will invalidate the entailment as a whole. Though these checks are performed by the prior rules in the implementation of our system (Section 4.3, 4.4), they could be undertaken by any existing entailment prover, such as the works of [Le et al., 2016, Bach et al., 2016, Gu et al., 2016b, Le et al., 2017], or [Xu et al., 2017].

4.6 Search Algorithm

While each of the rules are carefully designed, many are reliant upon key assumptions about the state of the entailment in order to operate both effectively and soundly. As an example, many of the match-and-subtract rules operate under the assumption that the entailment is in normal form, ensuring that there is little chance of the subtraction eliminating a key constraint before it has been fully explored. In order to ensure these assumptions are well-founded, a specialised search algorithm was developed to be utilised in this system.

The core design of the proposed algorithm is a repeating linear search over the rule-set, aiming to find a rule appropriate to apply to the current subgoal of the proof. Once such a rule is identified, it is applied, progressing the proof, creating new subgoals and possibly generating some new fragments of the frame or anti-frame in the process. These new details are recorded in a proof tree - constructed and maintained by the algorithm - before the search restarts, searching for the next rule application from the beginning of the sequence. This predictable order of search is a key aspect of the algorithm, as it allows for the rules to be searched in an order that will ensure the rules are applied at appropriate times.

For this system, the algorithm searches the rules in a group-by-group basis, starting with the normalisation rules. By leading the rule sequence with the normalisation rules, the search will ensure the entailment will be translated into its normal form as soon as possible in the search, with any subsequent subgoals later in the proof also being fully normalised as a first priority. Through this, it is now possible to apply the following rules under the assumption that the entailment is fully normalised, ensuring the subsequent subtraction or inference rules do not erroneously eliminate or introduce invalid constraints. The individual rules are also carefully arranged, prioritising key operations such as substitutions and the identification of non-`null` properties in order to ensure the minimum amount of ambiguity.

The subtraction rules are the next group to be explored by the algorithm. With the entailment in a fully normalised state, ambiguity in the equation will

have been minimised and several pure constraints should have been made explicit or simplified. Under these conditions, the match-and-subtract rules will operate most effectively, able to assume key properties, such as inequalities between variables and the absences of substitutions. This group of rules is typically responsible for much of the forward progress of the proof.

The inference rules are the final group of rules in the sequence, placed there due to the intended use as rules of last resort. As with the match-and-subtraction rules, this placement has a number of potential benefits to the rules application. The most simple is the assumption that if the inference rules are being checked, the entailment is fully normalised and there are no further matches or simplifications in the antecedent or consequent at this point in the proof. While this assumption could be difficult to enforce generally, by ensuring the search algorithm exhaustively searches all normalisation and match-and-subtract rules before considering the inference rules, it can be clearly seen that no other rule could have been applied; the search has reached a dead-end. As a result, any potential application of the inference rules advances a proof that would otherwise need to halt, typically identifying some new constraints in the process.

The inference rule sequence has another key constraint enforced over it: the inference rule sequence - as well as the other sequences, though to a lesser extent - is designed to prioritise simpler inferences, such as identifying potential equalities, over the inference of larger or more complex fragments such as an entire shape predicate. While there is typically an optimal solution for any valid bi-abductive problem, such efforts are typically not appropriate due to the cost of such actions [Calcagno et al., 2011] and prioritising smaller inferences tends to produce more reasonable and simpler results that are equally useful. The arrangement of the inference rules is designed to encourage this behaviour, as by prioritising the inference of smaller fragments, it is anticipated that the frames and anti-frames produced by this search will be of greater quality than alternative arrangements. This aspect will be discussed in greater detail in Section 4.6.1.

In the event that no suitable rules can be identified, the branch is left open and the search will attempt to continue over one of the other branches. If all of

the branches have reached a completed state - either closed via a decision rule, left open as no appropriate rule was identified, or closed via an inconsistency in the entailment - the search concludes, with the proof tree finalised in that state. The proof tree is subsequently processed in order to extract the final set of frames and anti-frames required to make the entailment valid (if any are produced). This final analysis also serves as the mechanism by which an invalid entailment is identified: if there are no branches where the entailment holds, the entailment is not be valid in its starting configuration and no corrections supported by this system can resolve it.

4.6.1 Quality of Output

A additional and somewhat unique property of our approach to the bi-abductive problem is that there is no application of a ranking function. As discussed earlier, abductive inference techniques can produce numerous valid anti-frames, and a ranking function is used to select the anti-frame of the greatest quality, typically the solution with the smallest memory footprint. While our system can produce multiple anti-frames, our technique does not apply such a function, and instead will return all the potential frame and anti-frame pairs produced. This design decision, though unusual, was made due to a number of factors, partly resulting from the approach itself.

Due to the design of the inference rules used in the system, the inference operations are somewhat restricted as to what they can infer. As an example, the INF-PTO rule is capable only of inferring an equality (or set of equalities) over the values in the record of the node, as opposed to the node itself. For a concrete example, consider the entailment $x{\neq}\texttt{null} * x{\mapsto}[a] \vdash y{\neq}\texttt{null} * y{\mapsto}[a]$. In this case, our system would consider x and y to be two *distinct* nodes; while an equally valid solution (for the wider abduction problem) would be $x = y$, our system always operates under the assumption that spatial terms are discrete, unless shown to be otherwise via the presence of an equality, $x = y$ in this case. While this does eliminate a range of potentially valid solutions, inferring an alias between

two potentially distinct nodes seems to be potentially dangerous operation, and leaves open the possibility of memory faults going undetected.

Despite these limitations, there are some advantages to this approach. Due to the specific order in which the rules are searched, the system has been designed to prioritise the inference of minimal corrections, rather than more considerable inferences. In essence, by carefully arranging the rules, both in terms of grouping and in terms of order within those groups, the system can be made to have a preference to infer pure constraints over spatial terms, and to prioritise smaller sets of spatial terms where possible, i.e. inferring a points-to term over a list predicate. Furthermore, the design of the normal form also reduces the need for a ranking function. As our normal form forces the antecedent to represent a singular configuration of the program heap, utilising case-splits where necessary, the system explores every possible arrangement of spatial terms. As a result of these decisions, the system typically produces a range of final entailments, each valid, and each describing a specific set of constraints over the initial predicates. While an additional stage in the system could be introduced to integrate these separate cases, it was decided that such an effort would not be in the best interest of the project, as this system was primarily a proof of concept, and further systems were not necessarily going to exhibit this property.

Ultimately, the set of final outputs describe a number of possible effective solutions to the bi-abductive entailment problem supplied, and it is up to the user as to which is the most appropriate for the scenario in which the system was applied.

4.7 Summary

In this chapter, we presented and described our novel bi-abductive system for the combined shape and ordering domain. This system, capable of handling a restricted fragment of lists and trees, follows in the design style of the Smallfoot [Berdine et al., 2005b] system and utilises unfold-and-match reasoning to reduce

a supplied entailment to a point where a decision can be made to its validity.

This system has also been implemented into an automated bi-abductive entailment proof tool based upon the Cyclist [Gorogiannis, nd] cyclic proof library, which was tested over a range of benchmarks from the SL-COMP competition. The experimental results of this tool are presented and discussed in Chapter 7.

Chapter 5

Generalised Bi-Abduction for the Combined Domain

This chapter details the design and construction of a generalised combined domain bi-abductive system, designed and built as an answer to the limitations identified in the initial proof-of-concept system detailed in Chapter 4. The motivation behind this generalised system will be presented and discussed in Section 5.1. The construction of the system will be presented in Sections 5.3, 5.4 and 5.5, before a final review in Section 5.6. As in other chapters, experimental results are presented together in Chapter 7.

5.1 Motivation

The motivation for creating a generalised form of combined domain bi-abduction is largely rooted in the limitations of the first combined domain bi-abduction system developed as part of this work (Section 4). The initial system showed that not only was a single-step bi-abduction system for shape and pure properties possible, the resulting system showed promise in both efficiency and expressiveness. Despite the immaturity of the implementation, the benchmark set examined was solved almost completely, and typically within 30s (Section 7.2). However, the

success of this tool was limited to one of the most basic benchmark sets included in the SL-COMP set, as the tool was only able to support (potentially sorted) singly-linked lists and binary trees. While trees and lists are some of the most commonly used data structures [Cormen et al., 2009], they are only a small subset of the data structures that are found in programs, especially considering the inclusion of more complex tree and list based structures such as AVL trees or Skip lists. As many programs make use of shape predicates that are unsupported by the initial technique, a significant number of examples could not be analysed, indicating that the technique would likely struggle if applied to more complex real-world examples. Indeed, even over the entailments that were already supported by the fragment, the technique was beginning to exhibit issues. Seemingly rooted in the branching behaviour of the EXCLUDE-MIDDLE rule, the initial system seemed to create a state-space explosion over the number of generated branches, likely proportional to the number of predicates in the entailment. While this only became a significant issue where a large number of predicates were present in the entailment, this increasing cost and poor scalability was a concern, and a notable flaw.

Though the initial system could be extended to support additional data structures, this would require a range of new inference rules to be introduced in order to describe how each new shape predicate would be unfolded and reduced. Though the effect of these additional rules on the performance would be minimal individually, each additional data structure supported would introduce *several* new "core" rules, and possibly a number of additional auxiliary rules in order to support or improve the overall quality of the analysis. As can be seen from the rule-sets detailed in Chapter 4, a large portion of the rules comprising the initial technique are already designed to operate over a single targeted predicate, and as the core search algorithm is designed to be exhaustive across each rule group, each additional supported data structure is anticipated to incur an increasing cost to the overall performance of the tool. When also considering that each of these new rules would need to be designed and introduced by-hand, adding a relatively large overhead to the process and increasing the demands placed on any possible users of the system, improving the expressiveness of the list-and-tree system in

this manner seems to inadvisable.

As a result of these issues, it was determined that a generalised approach to the problem was likely to be the most effective approach. In a generalised system, any predicates are handled by a small set of generalised proof rules, capable of operating over any predicate, provide sufficient information about that predicate is available. In many existing tools such as [Qin et al., 2017], this information is in the form of a set of *user-defined predicates*, an arbitrary set of predicate definitions supplied prior to the analysis that describe the data structures expected to be encountered during execution. These definitions are then referred to by the system during execution, guiding the analysis through unfolding operations.

This approach is expected to be superior to the targeted approach taken by the first system for a number of key reasons. First, by reducing the number of proof rules examined by the search, the search itself is likely to identify an appropriate rule application or fail in a shorted time-frame, without a negative impact on the expressiveness or soundness of the system. Second, extending the range of supported predicates is made significantly easier than the bespoke rules required by the list-and-tree system, reducing the introduction of a new predicate to the addition of the predicate definition only, reducing the overall effort required by an user of the system and providing opportunities for runtime definitions, such as the provision of a shape predicate alongside the entailment. Finally, if the language fragment supported by the system is sufficiently expressive, the system should be able to support a significant range of shape predicates, allowing for the bi-abductive mechanism to be applied to a much more significant range of properties than the first system and to compete with other, more established, tools in the literature [Qin et al., 2017, Le et al., 2018].

5.2 Overview

In order to position this contribution against the initial sorted list-and-tree system described in Chapter 4, a brief overview of the generalised language and

system will be presented, illustrating the additional capabilities this extended fragment includes and the advantages such inclusions may present. Additionally, the aspects of the initial list-and-tree system that were adopted to form part of the generalised system will also be identified and discussed.

5.2.1 Extended Fragment

One of the key additions over the list-and-tree system detailed in Chapter 4 is the extensions that were made to the supported symbolic heap representation. The initial motivation towards the creation of a generalised variant of the bi-abductive system presented in Chapter 5 was to enable the analysis of a wider range of shape predicates. While the introduction of generalised shape predicates and the underlying proof rules would extend the range of data structures supported by the resulting system, many shape predicates feature a more complex range of pure constraints beyond ordering. As an example, data structures that have a notion of "balance", such as AVL trees, require some level of arithmetic capabilities in order to represent the relative weights of the subtrees. Additionally, if the generalised system were to be applied to analysis problems focussing on the contents of a list, the capability to reason about collections of values would be necessary to effectively process such properties.

In order to support these potential applications, the language fragment was extended to include arithmetic and bag constraints. While the full formalisation of the language can be found in Figure 2.5.2, a brief discussion around the new capabilities will be presented here.

Bag Constraints

$$\beta ::= \quad \{\ \} \mid \{E_0, E_1, ...\} \mid \{V_0, V_1, ...\} \mid \beta \cup \beta \mid \beta \cap \beta \mid \beta \subset \beta \mid \beta - \beta$$

The first significant addition to the language fragment over the list-and-tree system was the addition of multiset or "bag" constraints. These constraints

describe collections of variables or values - potentially with repetition - and are of particular interest when used to reason about the contents of a given data structure. As an example, consider a variant of a singly-linked list, *lsb*. This predicate describes a singly-linked list with a tracked bag of the values contained by the various nodes of the list.

$$\begin{array}{rcl} lsb(E,F,B) & \stackrel{def}{=} & E = F \wedge B = \{\} \wedge \texttt{emp} \vee \\ & & E \neq F \wedge B = \{V\} \cup B' \wedge E{\mapsto}[E',V] * lsb(E',F,B') \end{array}$$

Through the introduction of support for Bag operations, program functions such as insertion and deletion from data structures may be effectively modelled. Such capabilities again advance bi-abduction from a tool for shape analysis and memory safety into a system were program correctness may also be considered, though such efforts are beyond the scope of this work.

Arithmetic Constraints

$$\alpha ::= \quad V + V \mid V - V \mid V \star V \mid V \div V \mid \texttt{MAX}(V,V) \mid \texttt{MIN}(V,V) \mid \texttt{ABS}(V)$$

Arithmetic constraints were another significant addition over the basic combined domain fragment utilised in the list-and-tree system. Though complex arithmetic is rarely encountered, a number of shape predicates, both simple and complex, make use of arithmetic formulae to describe properties such as size and balance. Two such examples would be a size-tracking variant of a singly-linked list and binary trees with a known height.

$$\begin{array}{rcl} lsn(E,F,N) & \stackrel{def}{=} & E = F \wedge N = 0 \wedge \texttt{emp} \vee \\ & & E \neq F \wedge N = N' + 1 \wedge E{\mapsto}[E',V] * lsn(E',F,N') \end{array}$$

$$\begin{array}{rcl} & & E = \texttt{null} \wedge H = 0 \wedge \texttt{emp} \vee \\ htree(E,H) & \stackrel{def}{=} & E \neq nil \wedge H = 1 + \texttt{MAX}(LH,RH) \wedge \\ & & E{\mapsto}[L,R] * htree(L,LH) * htree(R,RH) \end{array}$$

In both of these examples, the capability to handle arithmetic formulae is critical to effectively identifying and reasoning about height or depth properties.

For the singly-linked list, this can be accomplished through a simple constraint establishing the length of the list to be 1 greater than the length of the tail of the list following unfolding, a relatively straightforward calculation. For trees, however, this calculation becomes more complex. The height of a tree is not a direct calculation between the heights of the subtrees, but instead is an increment of the *larger* value only. In order to allow for these more complex relations, the generalised language fragment also includes a number of key functions - the maximum, minimum and absolute functions - ensuring that predicates that rely on such mechanisms are adequately supported.

Generalised Predicates

The most significant aspect of the new fragment is the support for generalised shape predicates. In the language fragment of the ordered list-and-tree system, shape predicates were predefined and fixed, restricted to singly-linked lists, sorted singly-linked lists, binary trees and binary search trees. In the generalised system, the intent is for any known and valid shape predicate to be supported by the bi-abductive methods. In order to achieve this, generalised shape predicates are necessary.

By introducing support for user-defined predicates into the system, alongside the addition of arithmetic and bags constraints, the eventual bi-abductive approach can be used to reason over a significantly larger and more complex set of shape predicates. Data structures such as AVL trees and Red-Black trees are particularly well-known examples of these advanced predicates and are often used as key benchmarks in the evaluation of other systems in the literature [Qin et al., 2017].

$$
\begin{array}{rcl}
 & & E = \texttt{null} \wedge B = 0 \wedge \texttt{emp} \vee \\
avl(E, H) & \stackrel{def}{=} & E \neq \texttt{null} \wedge H = 1 + \texttt{MAX}(LH, RH) \ \wedge \texttt{ABS}(RH - LH) \leq 1 \\
 & & \wedge \ E \mapsto [L, R] * avl(L, LH) * avl(R, RH)
\end{array}
$$

Well-formed Predicates Not all shape predicates may be supported easily by the bi-abductive mechanism, however. Specific arrangements and forms of shape predicate may cause difficulties in reasoning and cause some of the core operations of the system to fail to meaningfully progress.

One of the most significant restrictions applied to the definitions of shape predicates is the demand that the leading variable, or the root of the data structure, will always be unfolded as a points-to term during predicate unfolding operations. This restriction is critical for the safe and useful application of any of the unfolding rules of the system, as ensuring the root variable progresses reduces the risk of infinite unfolding cycles. As an example, consider a "reversed" singly-linked list lsr, where the shape predicate unfolds from the tail instead of the head, as in the more typical ls predicate.

$$lsr(x, y) \stackrel{def}{=} x = y \wedge \texttt{emp} \vee x \neq y \wedge x' \mapsto [y] * lsr(x, x')$$

In a scenario where an lsr predicate is matched with a points-to term, a scenario may unfold as such:

$$\texttt{Unfold}\ \frac{lsr(x, x') * x' \mapsto [y] \vdash x \mapsto [y]}{lsr(x, y) \vdash x \mapsto [y]}$$

In this situation, the unfolded points-to term is not likely to be identified as an aliased match. Because of the preference to unfold known predicates over infer new constraints in the system, it is anticipated that the search will again identify the match between the post-unfolding shape predicate in the antecedent and the initial points-to term in the consequent and repeat the unfolding indefinitely. By restricting the set of valid entailments to those at unfold from the head of the data structure, this issue is far less likely to arise.

The other key restriction is a requirement of at least one base case in the definition. While simple, this constraint is in place to support the proof rules by ensuring that several key rule applications will progress the search, either reducing the predicate in a manner which decreases "size" of the entailment or by reducing the predicate to some minimal case. By ensuring the predicate has a base case, unfolding operations will always reduce the predicate towards some

minimal case, allowing for that predicate to be removed via application of the Base rule, provided it is not eliminated through some other means.

5.2.2 Search Algorithm

As in the first system, the core of the generalised combined-domain bi-abductive technique is a search algorithm, one that aims to identify a series of inductive rule applications that will prove the validity of a given entailment, potentially with an inferred frame and anti-frame. This chain of rule applications is used to identify frames or anti-frames and construct a proof tree validating the decision, as well as a record of the sequence necessary to produce the result.

This aspect of the search is the one with the least changes between the initial bi-abductive system and the generalised variant, remaining unchanged in its structure and operation. The key differences between the two forms is the rule-sets that are explored as part of the search, with the overall approach and construction unchanged. As in the prior system, each set of rules is explored in sequence until an appropriate rule is found, at which point that rule is applied to the current sub-goal and the search restarts. The benefits and intent behind this approach have been discussed in greater detail in Section 4.6.

5.3 Normalisation

To effectively manage the extended language fragment utilised as the foundation of the generalised bi-abductive system, a refined normalisation mechanism was required. This extended mechanism was produced in order to effectively handle the introduction of generalised shape predicates, as the core mechanism of the prior normal form, predicate guards, cannot be easily identified and modelled in a generalised system. This new mechanism follows the same form as the Normalisation approach from the initial list-and-tree system, consisting of a normal form and a set of inductive proof rules to transform a given entailment into the

corresponding normalised representation.

5.3.1 Normal Form

As in the proof of concept system, a key and repeating stage of the proof search is the transformation of the entailment into its equivalent normalised representation. While the majority of these conversions are the explicit introduction of otherwise implicit state information, such as the identification of the non-`null` property of points-to terms, a select few aim to eliminate "empty" shape information from the entailment. This normal form is presented below:

Definition 3 (Generalised Normal Form) *A formula* $\Pi \wedge \Sigma$ *is in normal form (NF for short) if:*

1. $E \mapsto [\rho] \in \Sigma \implies E \neq null \in \Pi$.
2. $E_1 \mapsto [\rho_1] * E_2 \mapsto [\rho_2] \in \Sigma \implies E_1 \neq E_2 \in \Pi$.
3. $P(\overrightarrow{E}) \in \Sigma \implies \Pi \wedge Base(P(\overrightarrow{E})) \equiv false$
4. $E_1 = E_2 \notin \Pi$.
5. $E \neq E \notin \Pi$.
6. Π *is satisfiable.*

In this definition, $Base(P(\overrightarrow{X}))$ represents a disjunctive formula of all predicate cases of P under parameters $\overrightarrow{X}$ which are deemed "base cases": predicate branches in which no instance of the initial predicate, or any predicates recursively defined by that predicate, are present. A simple example would be the empty case of singly-linked list fragments

$$x = y \wedge emp$$

as no ls predicates are present in that particular configuration.

Though this normalisation is highly similar to the normal form of the list-and-tree system (Figure 2), the introduction of support for general inductive predicates requires some small modifications be made. The largest of these changes is the changes made to conditions 1 through 3 in the normal form. Where in the previous normal form conditions 1, 2 and 3 would apply to any spatial term, requiring the presence of both guards and non-null constraints over such terms, such conditions have been restricted to cover specific subsets. As an example, where condition 1 and 2 would have previously applied to shape predicates, the conditions of the generalised normal form do not, as it is no longer possible to guarantee specific properties, namely non-emptiness and thus inequalities arising from the separation of such terms.

In place of conditions 1 and 2, shape predicates are now the sole focus of condition 3. Condition 3,

$$P(\overrightarrow{E}) \in \Sigma \implies \Pi \wedge Base(P(\overrightarrow{E})) \equiv false$$

is an attempt to produce a generalised equivalent of the corresponding condition 3 in the list-and-tree system, aiming to establish that only predicates expected to be non-empty are present in the normalised representation. However, while the list-and-tree system could be assured of the non-empty properties of lists and trees, as the non-null requirement of the prior conditions enforced this property, the generalised system could not make the same guarantee. Instead, the normal form of the generalised system aims to establish the fact that the predicate cannot be unfolded to a base case under the current constraints. While this condition is more complex to establish, the overall result achieves the same goal, replacing predicates with their base cases where such unfoldings are enforced.

Conditions 4 through 6 are also unchanged from the list-and-tree system. Condition 4 still aims to ensure all variables are represented consistently, with any aliasing information applied to the state and any trivial equalities eliminated. Conditions 5 and 6 remain present in order to prevent the introduction of trivial inequalities and ensure that the formula is satisfiable, identifying invalid states and preventing the preventing the inference of invalid frames and anti-frames.

5.3.2 Normalisation Rules

To transform a supplied entailment into the corresponding normal form representation, a range of normalisation rules may be applied. Each of these rules aim to identify a particular configuration of state and apply a transformation to the entailment, progressing the entailment towards its normalised representation and forwarding the search overall. In order to separate the work done towards achieving a single-step generalised combined-domain bi-abductive system from the work that was re-used from the initial list-and-tree system, the rules will be presented in two groups: rules that have been reused or adapted from the list-and-tree system and rules that have been developed for the generalised system.

Re-used Rules

During the development of the generalised system, it was quickly realised that a number of the rules of the list-and-tree system could be easily adapted to operate within the generalised language fragment. While a small number of these rules can be adopted without modifications, several required adaptation to fit within the new system. The intended usages, as well as any changes made during the adoption, are detailed below.

Empty Axiom

$$\frac{}{\Pi \wedge \texttt{emp} * [\texttt{true} \wedge \texttt{emp}] \vdash \texttt{true} \wedge \texttt{emp} * [\Pi \wedge \texttt{emp}]} [\textbf{EMP}]$$

The empty axiom EMP is one of the rules of the list-and-tree system that was fully adopted into the generalised system. The EMP rule aims to identify scenarios where the consequent has been reduced to the empty state of $\texttt{true} \wedge \texttt{emp}$ and the antecedent has no remaining spatial information. In these cases, any remaining pure constraints in the antecedent are identified as part of the frame of the bi-abductive problem and the proof search concludes over that particular sub-goal.

Substitution

$$\frac{(\Pi \wedge \Sigma)[E/x] * [M] \vdash \Delta[E/x] * [F]}{(\Pi \wedge x{=}E \wedge \Sigma) * [M \wedge x{=}E] \vdash \Delta * [F \wedge x{=}E]} [\textbf{SUBST}]$$

The substitution rule is similarly unchanged, identifying equalities in the antecedent and substituting all instances of one of the variables in the entailment with the other. The equality is preserved as a fragment of the frame and anti-frame as in the prior system.

Left Identity

$$\frac{(\Delta) * [M] \vdash \Delta' * [F]}{(\Delta \wedge E{=}E) * [M] \vdash \Delta' * [F]} [\textbf{LIDENT} =]$$

The Left Identity rule is completely unchanged from the list-and-tree version, aiming to identify and eliminate self-equalities from the antecedent of the entailment, simplifying the entailment where possible and progressing the antecedent towards $\texttt{true} \wedge \texttt{emp}$.

Node-EX

$$\frac{E{\neq}\texttt{null} \wedge \Delta * E{\mapsto}[\rho] * [M] \vdash \Delta' * [F]}{\Delta * E{\mapsto}[\rho] * [M] \vdash \Delta' * [F]} [\textbf{NODE}{-}\textbf{EX}]$$

(where $E{\neq}\texttt{null} \notin \Delta$)

The Node-EX rule is the first rule that was modified from the version detailed in the list-and-tree system, adapted to handle the generalised normal form. In the version used in the initial system, the Node-EX rule identified any spatial term in the antecedent that was guarded and introduced a non-null constraint over that spatial term into the antecedent. These constraints were utilised primarily in reasoning around the separation of spatial terms, as any spatial term with a non-null variable as the root or head cannot be empty, and thus must not overlap with any other term.

As guards are not utilised in the generalised system, the Node-EX rule was restricted in its operations to identifying and extracting non-null constraints for

points-to terms only, ensuring that vital state information was made explicit and preserved throughout the solving process.

Nodes-EX

$$[\mathbf{NODES{-}EX}]$$
$$\frac{E_1 \neq E_2 \wedge \Delta * E_1{\mapsto}[\rho_1] * E_2{\mapsto}[\rho_2]) * [M] \vdash \Delta' * [F]}{\Delta * E_1{\mapsto}[\rho_1] * E_2{\mapsto}[\rho_2] * [M] \vdash \Delta' * [F]}$$
$$(\text{where } E_1{\neq}E_2{\notin}\Delta)$$

The Nodes-EX rule aims to identify and made explicit the inequalities implied by the separation of non-empty spatial terms in separation logic. This inequality was only made explicit when the non-emptiness of the two terms is assured, either intrinsically or through the presence of constraints in the entailment that would ensure such a property. Due to the removal of guards from the generalised system, establishing the emptiness of shape predicates is not a simple task, and the Nodes-EX rule is restricted to operating over points-to terms only as a result. This change maintains the core operation of the Nodes-EX rule, though removing some of its former capabilities as a result.

The restrictions applied to the Node-EX and Nodes-EX rules do introduce some serious limitations into the generalised system, particularly around the identification and preservation of emptiness and separation between predicates. In order to effectively address these limitations, as well as support the manipulation and reasoning around general shape predicates, it was necessary to create a small number of generalised normalisation rules.

Generalised Normalisation Rules

While many of the necessary rules could be adopted from the previous list-and-tree system, a number of key behaviours required the creation of a number of new generalised normalisation rules. These rules are primarily applied to the shape predicates present in the entailment, aiming to transform these user-defined predicates into the corresponding normalised form, as well as establishing key constraints over those predicates, such as emptiness properties.

Generalised Base Rule

$$[\textbf{BASE}]$$
$$\frac{\Pi \wedge \Sigma * \mathit{Base}(P(\overrightarrow{E})) * [M] \vdash \Delta' * [F]}{\Pi \wedge \Sigma * P(\overrightarrow{E}) * [M] \vdash \Delta' * [F]}$$
$$(\text{where } \Pi \wedge P(\overrightarrow{E}) \implies \mathit{Base}(P(\overrightarrow{E})) \wedge \vdash \mathit{Rec}(P(\overrightarrow{E})))$$

One of the key conditions of the normal form is the identification and elimination of "empty" shape predicates, or more generally, the identification of shape predicates that can be unfolded without increasing the size of the symbolic heap. Almost all shape predicates known to the author feature at least one of these conditions, representing a base case in the inductive definition of the shape. The most typical example is the base case of ls predicates, describing an unfolded state that is empty in the heap when the head and tail are identical variables. By identifying state configurations that force shape predicates to unfold into this base case, it is possible to eliminate such predicates from the entailment quickly and efficiently, simplifying the entailment and advancing the proof.

Due to the wider range of predicates targeted in this generalised system, identifying such states is a much more complex problem than in the initial system. When designing the list-and-tree base rules, LS-LBASE, SLS-LBASE and the tree equivalents, the conditions enforcing the base conditions were directly identified and quantified within the rules themselves. Such an approach is not easily generalised, as the conditions enforcing a base-case unfolding - or possibly many valid base-case unfoldings - are essentially arbitrary under the user-defined predicate approach.

The BASE rule developed to address this aspect of the normalisation process aims to overcome this difficulty. The BASE rule aims to identify scenarios where a predicate, constrained by the rest of the antecedent, can only be unfolded into its base cases, effectively eliminating the predicate from the entailment. Once such a state is identified, the predicate is unfolded, reducing the predicate to its definition and making progress towards the normalisation. Depending on the predicate identified, it is possible for multiple valid base cases to be identified as an unfolding. Should this occur, the BASE rule will introduce a case split, creating

a set of new branches corresponding to each base case, with any invalid case immediately resulting in a dead-end in the search, pruning that branch out of the proof tree. By following this design, the search attempts to identify and explore as many valid configurations of a predicate as possible.

Detecting Conflicting Predicates

$$\frac{Base(P_1(\overrightarrow{X})) \implies \texttt{emp} \quad \Delta * Base(P_1(\overrightarrow{X})) * P_2(\overrightarrow{X}) * [M] \qquad\qquad Base(P_2(\overrightarrow{X})) \implies \texttt{emp} \quad \Delta * P_1(\overrightarrow{X}) * Base(P_2(\overrightarrow{X})) * [M] \quad \vdash \Delta' * [F]}{\Delta * P_1(\overrightarrow{X}) * P_2(\overrightarrow{X}) * [M] \vdash \Delta' * [F]}$$

Due to the restrictions applied to the Node-EX and Nodes-EX rules, it became necessary to establish a rule that attempted to resolve scenarios where two separate predicates in the antecedent shared a root. While such a state appears to violate the separation property of separation logic, if one or both of the conflicting predicates were equivalent to empty, there would be no conflict. The Resolve Conflict rule aims to establish a state where this solution is enforced, resolving the conflict and preserving the validity of the entailment in the process.

The Resolve Conflict rule is applied when two shape predicates are identified as sharing a root variable and aims to identify configurations of the predicate that will establish one or both predicates as empty, resolving the conflicting state. The rule operates in a straightforward manner: if there is a base condition that describes an empty heap, the rule identifies pure constraints that will enforce that condition, introducing them into the entailment and enabling the future application of the Base rule, removing one of the predicates from the entailment. In the event that both predicates have an acceptable empty configuration, the rule will produce two branches to explore both configurations, examining both possibilities for validity or differences in the frame and anti-frame.

The Resolve Conflict rule is an important addition to the normalisation rule-set due to the issues conflicting predicates would introduce when attempting to apply the rules from subsequent rule-sets. Ignoring the necessity of resolving the violation of separation properties, such a configuration would introduce a significant amount of ambiguity when attempting to resolve a match between the

two “overlapping” predicates and a points-to term in the consequent, a situation that cannot be easily resolved. By enforcing a scenario in which only one of the predicates is allowed to be in a non-empty state, the conflict is immediately resolved, eliminating the ambiguity and allowing the subsequent rules to operate as intended. While such entailments are rare, they may occasionally arise during the analysis of complex interlinking predicates, requiring that the system is able to process them effectively.

In the event that the rule cannot find an appropriate configuration, the search over that sub-goal immediately terminates with an negative result, as such a shortcoming cannot be addressed by bi-abductive means.

5.4 Subtraction

Once the entailment has been fully normalised through applications of the normalisation rules, the subtraction rules can be applied. The purpose of the subtraction rules remains the identification of aspects of the entailment that overlap or match, removing them and progressing the search towards a state where a decision can be made as to the validity of the entailment. The overall function of the subtraction rules is largely unchanged from the list-and-tree system, and many of the same expectations over the entailment, such as it being in the corresponding normal form, still hold.

5.4.1 Recurring Rules

As in the Normalisation rule-set, there was a number of rules in the list-and-tree system that could be adopted into the generalised system. As briefly discussed in the Normalisation section (Section 5.3), these rules were typically those that targeted more general sub-formulae and were not reliant on more rigid definitions, as in the rules that targeted predicates. As a result, these rules required minimal changes in order to operate in the generalised system.

Identity

$$[\textbf{IDENT}]$$
$$\frac{}{\Pi \wedge \Sigma * [\texttt{true} \wedge \texttt{emp}] \vdash \Pi \wedge \Sigma * [\texttt{true} \wedge \texttt{emp}]}$$

The identity axiom remains unchanged from the instance found in the list-and-tree system, identifying entailments where the antecedent and consequent are identical and closing that branch of the proof search.

Empty

$$[\textbf{EMP}]$$
$$\frac{}{\Pi \wedge \texttt{emp} * [\texttt{true} \wedge \texttt{emp}] \vdash \texttt{true} \wedge \texttt{emp} * [\Pi \wedge \texttt{emp}]}$$

The empty axiom is similarly unchanged, identifying entailments where the consequent is in the `emp`state, $\texttt{true} \wedge \texttt{emp}$ and no spatial terms remain in the antecedent. When such an entailment is found, the branch is closed, with any remaining pure constraints from the antecedent identified as part of the frame.

Right-Identity

$$[\textbf{RIDENT} =]$$
$$\frac{\Delta * [M] \vdash \Delta' * [F]}{\Delta * [M] \vdash \Delta' \wedge E{=}E * [F]}$$

The Right-Identity rule identifies and eliminates trivial $E{=}E$ pure constraints from the consequent, reducing the overall size of the entailment.

R-EX

$$[\textbf{R} - \textbf{EX}]$$
$$\frac{\Delta * [M] \vdash \Delta'[V'/V] * [F]}{\Delta * [M] \vdash \exists V.\ \Delta' * [F]}$$
$$(\text{where } V' \in FV(\Delta))$$

The R-EX rule, as in the list-and-tree system, identifies existential variables in the consequent and instantiates them as a one of the free variables of the antecedent. This instantiation is applied to all instances of the existential variable,

aiming to find a configuration where a match can be produced. In both implementation, this rule is constrained to only be applied when the renaming enables a subsequent reduction of the entailment, which must follow directly.

Hypothesis

$$\text{[HYPOTHESIS]}$$
$$\frac{\Pi \wedge \Sigma * [M] \vdash \Pi'' \wedge \Sigma' * [F]}{\Pi \wedge \Sigma * [M] \vdash \Pi' \wedge \Pi'' \wedge \Sigma' * [F]}$$
$$(\text{where } \Pi \Rightarrow \Pi')$$

The Hypothesis rule, as in its previous version, identifies pure constraints present in the consequent that are implied by the pure constraints of the antecedent. In such a case, the implied constraints are eliminated, progressing the search by further reducing the consequent towards the minimal state. This rule is unchanged when compared with the variant found in the list-and-tree system.

Separating Conjunction Introduction

$$\text{[}*\text{−INTRODUCTION]}$$
$$\frac{\Sigma * [M_1] \vdash \Sigma' * [F_1] \qquad \Pi \wedge \Sigma_1 * [M_2] \vdash \Pi' \wedge \Sigma_2 * [F_2]}{\Pi \wedge \Sigma * \Sigma_1 * [M_1 * M_2] \vdash \Pi' \wedge \Sigma' * \Sigma_2 * [F_1 * F_2]}$$

Another key rule brought forwards into the generalised system is the $*$-Intro rule. This rule is applied when a sub-formula of spatial terms in the antecedent is sufficient to satisfy a sub-formula of spatial terms from the consequent, potentially with some supporting fragment of frame or anti-frame. This rule identifies and separates out these fragments into a separate branch to confirm this relation, with the search continuing from the remaining entailment. Any fragments of the frame and anti-frame identified as part of the sub-proof are integrated into the final results once the overall proof is completed.

5.4.2 Generalised Rules

As with the other rule groups, a number of generalised rules were necessary in order for the system to operate effectively in the extended language fragment. As

generalised shape information can be handled by a number of the rules above, the generalised rules are aimed at the handling of shape predicates, and more specifically, operations that include unfoldings.

Right-Base Rule

$$[\textbf{RBASE}]$$
$$\frac{\Delta * [M] \vdash \Pi' \wedge \Sigma' * Base(P(\overrightarrow{E})) * [F]}{\Delta * [M] \vdash \Pi' \wedge \Sigma' * P(\overrightarrow{E}) * [F]}$$
$$(\text{where } \Pi' \wedge P(\overrightarrow{E}) \implies Base(P(\overrightarrow{E})) \wedge \vdash Rec(P(\overrightarrow{E})))$$

The Right-Base rule is the consequent-equivalent form of the Base rule applied during the Normalisation stage of the proof search. As in the normalisation, it identifies shape predicates present in the consequent that are in a state where only base-case unfoldings will be valid in the entailment. When such a state is identified, the relevant shape predicate is unfolded, eliminating that predicate from the entailment and replacing it with its valid base cases. Where multiple base-cases are valid, the rule will also introduce a split in the proof, with one branch for each valid case.

Unfold Match Rule

$$[\textbf{UNFOLD} - \textbf{MATCH}]$$
$$\frac{\Delta * unfold(P(\overrightarrow{E})) * [M] \vdash \Delta' * [F])}{\Delta * P(\overrightarrow{E}) * [M] \vdash \Delta' * E \mapsto [\rho] * [F]}$$
$$(\text{where } unfold(P(\overrightarrow{E})) \text{ produces some } E \mapsto [\rho])$$

Unfolding shape predicates in the antecedent of an entailment is a key aspect of the unfold-and-match paradigm followed by our bi-abductive system. While in the list-and-tree system this unfolding was carefully managed and targeted through the design of the REC and REC (Node) rules, such curated unfolding mechanisms were not possible in the generalised rules. In the place of these tailored rules, the Unfold-Match is used.

The Unfold-Match rule operates in the same spirit as the REC rules, identifying a match between the nominal head of a predicate in the antecedent and a points-to term in the consequent and attempting to generate a matching points-to term through unfolding. This unfolding process is the first potential hurdle, however.

As the rule must be able to operate over the general case of shape predicates, the unfolding process must be curated to ensure that a match will be produced. Consider the case of basic singly-linked lists, making use of a general unfolding and case-splitting rules:

$$\texttt{Split}\,\frac{x = y \wedge \Delta \vdash \Delta' * x{\mapsto}[w] \qquad x{\neq}y \wedge \Delta * x{\mapsto}[x'] * ls(x', y) \vdash \Delta' * x{\mapsto}[w]}{\texttt{Unfold}\,\dfrac{\Delta * (x = y \vee x{\neq}y \wedge x{\mapsto}[x'] * ls(x', y)) \vdash \Delta' * x{\mapsto}[w]}{\Delta * ls(x, y) \vdash \Delta' * x{\mapsto}[w]}}$$

In this unfolding, two branches are created, each corresponding to one of the two cases of the predicate. However, it is immediately apparent that one of these branches is a practical dead-end. In the empty case of the ls predicate, no points-to term is generated, consuming the predicate and only identifying a new alias between x and y in the process, leaving no spatial terms to satisfy the points-to term in the consequent. As a result of this behaviour, the Unfold-Match rule is restricted, only permitting cases producing a matching points-to term to be produced from the unfolding operation. By introducing this restriction, it becomes significantly more unlikely for impractical unfoldings to be produced as part of the search, reducing the search-space of the analysis and accelerating the process as a result.

5.5 Inference

Once the entailment has been normalised and any immediate subtractions have been undertaken, the entailment may be left in a state where the proof search would be unable to continue. In such a situation, the search begins examining the set of inference rules to attempt to find a match that will allow for the search to continue. The inference rules remain the stage in which the bulk of the inference operations take place and are still the only rules capable of identifying spatial information as part of the frame and anti-frame. While the inference rules have

seen some modifications from the versions utilised in the list-and-tree system, they retain one of the key properties: the inference rules are treated as rules of last resort, and the identification of new information is only performed when the existing information is insufficient to progress the proof search.

5.5.1 Reused Rules

As with the other rule-sets presented throughout this chapter, the inference rule-set of the generalised system includes a number of the inference rules of the list-and-tree system. The inference rules in particular were designed to operate in a highly generalised manner, each designed to target broad categories of formulae and conditions in order to identify unsatisfied sub-formulae in the current entailment. As a result, these rules were able to be imported into the generalised system with almost no modifications necessary.

Infer Pure

$$\frac{}{(\Pi \wedge \texttt{emp}) * [\Pi' \wedge \texttt{emp}] \vdash \Pi' \wedge \texttt{emp} * [\Pi \wedge \texttt{emp}]} [\textbf{INF}{-}\textbf{PURE}]$$
(where $\Pi \wedge \Pi'$ is satisfiable.)

Infer Pure is the last axiom of the generalised system and retains its behaviour from the previous system. The rule identifies entailments where both the antecedent and consequent have been reduced to pure constraints only. In such a case, the remaining pure constraints are identified as fragments of the frame and anti-frame, reducing the entailment to $\texttt{true} \wedge \texttt{emp} \vdash \texttt{true} \wedge \texttt{emp}$ and closing the branch.

Infer Point-to

$$\frac{(E_0{=}E_1 \wedge \Delta) * [M] \vdash \Delta' * [F]}{\Delta * E{\mapsto}[E_0] * [E_0{=}E_1 \wedge M] \vdash \Delta' * E{\mapsto}E_1 * [F]} [\textbf{INF}{-}{\mapsto}]$$

The Infer Point-to rule is unchanged from the existing variant, identifying a points-to term in the antecedent and consequent that shares a root variable, but has different values within its node. In such a scenario, the rule identifies a set of substitutions that will equate the points-to terms and introduces them into the entailment, eliminating the points-to terms in the process.

Infer Missing and Infer Extra

$$\frac{\Delta * [M] \vdash \Delta' * [F] \qquad \Delta * \sigma \not\vdash \texttt{false}}{\Delta * [M * \sigma] \vdash \Delta' * \sigma * [F]} \; [\textbf{INF}-\textbf{MISSING}] \qquad \frac{\Delta * [M] \vdash \Delta' * [F] \qquad \Delta' * \sigma \not\vdash \texttt{false}}{\Delta * \sigma * [M] \vdash \Delta' * [F * \sigma]} \; [\textbf{INF}-\textbf{EXTRA}]$$

The Infer Missing and Infer Extra rules are the two most significant inference rules of the generalised system, identifying spatial terms in the entailment that cannot be otherwise reduced or eliminated from and introducing them into the frame or anti-frame as appropriate. As in the list-and-tree system, these two rules are the only inference rules that can identify spatial fragments of the frame and anti-frame, capable of inferring both points-to terms and whole shape predicates, where other inference rules may only infer pure constraints. As a result of this property, the INF-MISSING and INF-EXTRA rules are the last two rules examined by the search.

5.5.2 Generalised Rules

The generalised rules of the inference rules, like the other rule-sets, are primarily targeted at shape predicates. For the generalised inference rules, the focus is detecting and resolving matches between points-to terms present in the antecedent and shape predicates in the consequent. Unlike the comparable rules in the subtraction rule set (Section 5.4), the proof rules of the inference rule set must take additional steps in order to effectively handle these operations. These rules are described below.

Infer Unfold

$$[\textbf{INF} - \textbf{UNFOLD}]$$
$$\frac{E{\mapsto}[F] * (R_1 \vee ... \vee R_n) \vdash unfold(P(\overrightarrow{E})) \qquad \begin{array}{c}\Delta * [M] \vdash \Delta' * R_1 * [F] \\ \cdots \\ \Delta * [M] \vdash \Delta' * R_n * [F]\end{array}}{\Delta * E{\mapsto}[F] * [M] \vdash \Delta' * P(\overrightarrow{E}) * [F]}$$

Infer Unfold is the inference counterpart of the Unfold Match rule, aiming to identify and resolve overlap between a points-to term in the antecedent and a shape predicate in the consequent. As with the Unfold Match rule, the Infer Unfold rule is applied when a points-to term and a predicate share a root variable, indicating an match and overlap between the two.

Unlike the Unfold Match rule, the Infer Unfold rule operates in a manner which ensures that any residual terms produced by the unfolding are fully represented in the consequent, and that this viability is checked in a separate entailment, in order to ensure all such residues are explored without conflicting with the remainder of the antecedent.

Predicate Match

$$\frac{\begin{array}{c}\Delta * Base(P(X, \overrightarrow{I}) * [M] \vdash \Delta' * P(X, \overrightarrow{J}) * [F] \\ \overrightarrow{I} = \overrightarrow{J} \wedge \Delta * P(X, \overrightarrow{I}) * [M] \vdash \overrightarrow{I} = \overrightarrow{J} \wedge \Delta' * P(X, \overrightarrow{J}) * [F] \\ \Delta * P(X, \overrightarrow{I}) * [M] \vdash \Delta' * Base(P(X, \overrightarrow{J})) * [F]\end{array}}{\Delta * P(X, \overrightarrow{I}) * [M] \vdash \Delta' * P(X, \overrightarrow{J}) * [F]}$$

The RULE rule is applied when a predicate in the antecedent and a predicate in the consequent of the entailment is found to share a type and root variable but have a number of differing parameters. The purpose of RULE is the resolution of this partial match, attempting to identify spatial overlap between the two predicates and eliminating it, potentially identifying some residual fragment in the process.

The rule is founded on the notion that predicate-predicate matches can resolve in one of three possible outcomes: the predicate of the antecedent is insufficient to satisfy the corresponding predicate in the consequent, the predicate of the

antecedent satisfies the predicate in the consequent with some additional fragment remaining, or both predicates are equal with previously unidentified aliasing.

In the case where the predicates are determined to be equal, the resolution is straightforward: the rule will identify the parameters that differ and construct equalities resolving this mismatch. These equalities, once applied through applications of the SUBST rule, should enable the now-identical predicates to be eliminated through the use of $*$-INTRO, eliminating the predicates. This outcome is the most simple of the three cases, but is ultimately insufficient to effectively handle many possible entailments and in particular entailments including *segmented* shape predicates, where the predicates can be easily partitioned and recombined.

In the other two cases, the overall behaviour is equivalent, only varying on which of the two predicates is the focus of the key operations. The other two cases aim to represent scenarios where the predicates are of differing "sizes", where one predicate does not represent sufficient state to satisfy the other. As a basic example, consider the entailment:

$$ls(x, y) \vdash ls(x, z)$$

This entailment outlines a scenario where the RULE would be applied: a pair of predicates with identical type and root, but a difference in parameters. Applying the equality logic would identify one solution, $y = z$, establishing the two list predicates as identical and enabling the subsequent elimination of the predicates. However, in the case that $ls(x, y)$ represents a list segment of differing size - or $y \neq z$ is present in the entailment - this solution would not be effective. Instead, this relation must be explored in the proof.

The logic underpinning this exploration is the notion that the predicates may be unfolded in parallel, matching and eliminating the concrete state node-by-node. This process, if not correctly managed, is an infinite one; unfolding the predicates in parallel will simply produce identical terms, the majority of which would be matched and removed.

$$\textit{Unfold Predicates}\ \frac{\dfrac{x \neq y \wedge ls(x', y) \vdash x \neq z \wedge ls(x', z)}{x \neq y \wedge x{\mapsto}x' * ls(x', y) \vdash x \neq z \wedge x{\mapsto}x' * ls(x', z)}\ *\text{-INTRO}}{ls(x, y) \vdash ls(x, z)}$$

However, if the predicates are treated as objects of finite size, a point would be reached during this process where one or both of the predicates cannot be unfolded further, reaching a base case in its definition and ending the process.

This approach forms the foundation of the RULE rule: one of the matching predicates is selected and unfolded to its base cases, effectively eliminating it from the entailment but leaving some residual state and constraints. This residual state can be handled in the standard manner, applying substitutions or eliminating identified matches. This process is demonstrated in the continuation of the example shown below.

$$\text{RULE}\ \dfrac{\text{SUBST}\ \dfrac{\texttt{true} \wedge \texttt{emp} \vdash ls(y,z)}{x = y \wedge \texttt{emp} \vdash ls(x,z)}}{ls(x,y) \vdash ls(x,z)}$$

As can be seen from this basic proof sketch, unfolding the list segment in the antecedent to its base case has resulted in predicate of the consequent being "reduced" to a smaller segment. This segment, inferred as a fragment of the anti-frame, would produce a resulting entailment:

$$ls(x,y) * ls(y,z) \vdash ls(x,z)$$

This logic - which also forms the basis of the LS-REC and SLS-REC rules from the list-and-tree system (Section 4.4) - allows the generalised system to identify predicate segments as a fragment of the frame or anti-frame, permitting a greater degree of precision when reasoning over partitioned and overlapping shape predicates.

As there is no way to fully establish the which of the three scenarios is the best choice in a general case, the rule instead produces a split in the proof, producing a branch corresponding to each general scenario and additional branches for each valid base unfolding within those scenarios. While this does improve the overall scope of our technique, allowing for greater exploration of the range of possible solutions, the branching reintroduces one of the more notable limitations of the prior list-and-tree technique, the production and exploration of large amounts of branches in the proof. While this approach will have a negative impact on the overall performance, the cost is necessary; limiting the handling of partial predicate matches to direct equalities only will incur an significant cost

to the effectiveness and expressiveness of the technique, as well as degrade the performance over examples that exploit segment interactions unacceptably.

However, while the introduction of greater numbers of branches will have a negative impact, it is anticipated to have a less significant impact than in the prior list-and-tree system due to the later application of rules introducing case splits, as well the more precise applications of these splits. Though the risk of time-outs is still present, it is anticipated to be less severe due to these additional mitigations.

5.6 Summary

In this chapter, the generalised combined-domain bi-abductive inference technique was presented. An extension and refinement of the combined domain bi-abductive system for the restricted shape-and-ordering domain over lists and trees, the generalised system is capable of reasoning over a more expressive combined domain of shape, bag, ordering and more general arithmetic constraints over general user-defined shape predicates. In the same manner as the list-and-tree system, the general system utilises a search algorithm that aims to identify a sequence of rule applications based on unfold-and-match reasoning, reducing the entailment to a point where a decision can be made about the validity of the entailment, possibly with an inferred frame and anti-frame.

An automated implementation of this system was also developed, itself based upon the implementation of System 1, and tested over a number of divisions of the SL-COMP competition in order to compare the two systems over a consistent set of benchmarks. The results of these evaluations are detailed in Chapter 7.

Chapter 6

Automated Implementations

In order to effectively study and evaluate the techniques developed during the project, as detailed in Chapter 4 and Chapter 5, the techniques were implemented as automated bi-abductive solving tools. The implementation of the techniques into an automated tool was a key step in the evaluation of single-step combined domain bi-abduction, as while the techniques can theoretically be applied by-hand, it is both typical and expected that many examples would be impractical to solve in such an approach. As a result, an automated implementation was a key supporting artefact for the system, allowing for the practical investigation of the effectiveness of the technique over such entailments. Additionally, and perhaps more importantly, the majority of the current leading systems and techniques in the field [Calcagno and Distefano, 2011, Qin et al., 2017, Fragoso Santos et al., 2019, Fragoso Santos et al., 2020] each present a corresponding automated implementation, along with a set of experimental results; in order to effectively compare my system with these approaches, an automated implementation was necessary.

In this section, the two tools developed during this project will be presented, highlighting key design considerations, as well as drawing attention to the changes made to permit the techniques to operate effectively in an automated setting. We will begin by presenting a brief discussion on the foundations of both tools in Section 6.1, before presenting each separately in Section 6.2 and Section 6.3.

6.1 Foundations

In order to both simplify the implementation process, as well as reduce the amount of work required to construct an effective implementation, it was decided to base the implementation of the proof-of-concept system upon an existing automated tool or framework. A number of possible foundations were considered for this purpose, with potential candidates being selected for the expressiveness of the supported language, the effectiveness of the tools overall, and where possible, existing inference capabilities. Though a number of options were identified, including Infer [Calcagno and Distefano, 2011], HIP/SLEEK [Qin et al., 2017] and variants of the S2 tool [Le et al., 2016], it was eventually decided to use the Cyclist framework as the starting point [Gorogiannis, nd].

The Cyclist framework is a verification framework designed to produce so-called *cyclic* proofs of program correctness and termination of heap-manipulating programs. Written in the OCaml functional language, the framework is based around the construction of a proof tree via a search over a set of inductive inference rules, aiming to reduce an entailment to a point where a decision about its validity can be made. The design of this framework shares a number of qualities with the systems developed during this project, described in Chapters 4 and 5, though the framework was not selected for these points alone.

The Cyclist framework had a number of additional qualities that made it a preferable choice over the other considered tools. First, the framework was designed to be easily adapted and extended for new instantiations. The developers of the framework had already created a number of instantiations targeting a range of different analysis problems, with existing tools for program verification over toy examples with loops or procedures and a number of tools handling separation logic formulae or entailments. This seeming ease of instantiation was obviously of great interest, as creating a new instantiation of a framework would be far preferable to a complete adaptation of a singular tool.

The supporting mechanisms of the framework are also of great interest. While the main focus of the implementation is creating an automated version of the bi-

abductive systems we created during this project, a number of key sub-systems are necessary in order to use such implementations effectively. One of the best examples of this is the parsing system. An implementation of the monadic parser described in [Hutton and Meijer, 1996], the parsing systems of the cyclist framework are extremely modular, allowing for the creation and application of specialised parsers designed to identify specific types of input. These parsers, used throughout the framework, can also be composed together, creating highly complex and adaptable parsers capable of identifying a range of patterns from a single input. This approach allows for relatively complex parsers to be created in an efficient and straightforward manner, a significant benefit when attempting to parse entailments with a wide range of constraints, properties and predicates.

Another point of interest was one of the existing instantiations of the framework, `sl_abduce`. This tool, an implementation of the system described in [Brotherston and Gorogiannis, 2014], was a program verification technique that was capable of the abduction of key shape information. This, alongside basic framing capabilities required for the construction of the backlinks used to create cyclic proofs [Brotherston, 2005] suggested the framework already had some of the necessary structures and capabilities required to introduce full bi-abductive inference. Unfortunately, further examination would eventually lead to the conclusion that such mechanisms were not easily translated into a form useable by our bi-abductive techniques, leading to the eventual construction of alternative mechanisms. Nevertheless, the capability of the tool to construct shape predicates from program code is of significant interest and offers a potential avenue for further research in this area.

Another area of particular note was the Cyclist frameworks representation of separation logic formulae. The Cyclist framework is based around the symbolic heap fragment of separation logic, a similar fragment as the one used in our bi-abductive systems. Though restricted to (dis)equalities and basic spatial terms, the representation appeared to be efficient and effective, as well as being simple in its design, with its symbolic heaps constructed as an OCaml record with fields representing each form of constraint in the supported language, namely equalities,

disequalities, points-to terms and shape predicates. Such a record structure would be easily extended, simplifying the introduction of new sets of constraints, a critical stage of any planned expansion to the expressiveness of the supported fragment.

Because of the properties and advantages identified, the Cyclist framework appeared to be the best choice of the potential bases, and would form the foundation of the systems implemented during this project.

6.2 Initial System Implementation

The first system to be implemented was the initial bi-abductive inference technique for the ordered list-and-tree combined domain. This first prototype implementation was designed to be both an exploration of the potential strengths of the base system in an automated setting, and if of sufficient quality, the basis of further implementations.

The implementation process had a number of considerations that needed to be made in order to produce an effective tool, including restrictions over the base system, as well as taking advantage of existing code to accelerate the process. These aspects will be discussed below.

6.2.1 Key Additions

While the Cyclist framework provided a strong foundation for the implementation of the proof-of-concept technique, a number of key extensions and modifications were necessary in order to support the extended language fragment, as well as introducing mechanisms to perform the critical inference operations necessary for a bi-abductive analysis.

Ordering Constraints

The introduction of ordering constraints into the core language of cyclist was one of the first extensions to be made to the framework. While the Cyclist framework is based around the use of the symbolic heap fragment of separation logic, it is the standard space-and-equality form of the fragment and has no support for numeric data at all. In order to introduce support for numerical data, as well as ordering constraints, a number of key modifications were needed.

The first, relatively minor, adjustment to the system was modifying the definition of a "Term", the core representation of variables and special values in the Cyclist framework, to additionally include numerical values. These changes were relatively straightforward, expanding the corresponding parser to detect numerical terms and ensuring that such terms are recognised and marked as such, necessary for basic simplifications and checks.

The rest of the required changes were more involved. The internal model of a symbolic heap in the Cyclist framework is a single record containing a number of fields. These fields represent a particular form of constraint in the symbolic heap, initially consisting of equalities, disequalities, points-to terms and shape predicates. Each of these fields contains a set of the constraints, represented as a module to store and manipulate them and a module modelling the key attributes and behaviours of a singular instance. In order to add support for ordering constraints while preserving the overall structure and behaviours of the Cyclist heap representation, two new modules were created: one for a singular ordering constraint and one representing a set of such constraints. These modules were constructed following the structure of comparable modules already constructed as part of Cyclist, following the general naming convention and style. Indeed, many of the changes made as part of the implementation have no effect on the various existing instantiations of the framework.

The core of the representation is a pair of terms, either variables or numerical values. This pair is passed to an OCaml constructor, attaching a "less-than" or a "less-than-or-equal-to" tag to the pair in order to identify between the two

relations. “Greater-than” and “greater-or-equal-to” relations are represented using the less-than equivalent representation, simply reversing the relative order of the parameters during the record creation. This representation keeps the construction of ordering constraints relatively simplistic and easy to perform while preserving the key functionality required to manipulate and use them.

Bi-Abductive Mechanisms

The most significant addition to the Cyclist framework was the mechanisms and related support systems for bi-abductive inference. Though the extension to the language fragment was a highly important task for the exploration of the combined ordering and shape domain, the bi-abduction mechanisms were anticipated to be of equal significance and greater complexity.

Delegating the inference process to the implementation of the rules was quickly determined to be the best approach due to the integration of inference into the proof rules in the underlying technique. This approach was decided due to the belief that separating that process from the rules could cause issues in the translation from the theoretical basis into automated tool, as well as necessitating a more complex general inference technique in order to achieve the expected result from a supplied state; in the base rule-set, the inferred fragments are clearly defined in the rules themselves, a far simpler approach. Instead, it was determined that the corresponding support mechanisms for inference would be designed in a manner that would permit the proof rules to identify and record the fragments during the application of the selected rule, as in the core technique.

This design proved to be relatively straightforward to implement. The core proof mechanisms of the cyclist framework were designed to take a partial proof tree and a set of proof rules and identify an appropriate rule to apply to the current entailment. These rules would be subsequently “applied” to the current sub-goal and generate the appropriate result. This new entailment would be recorded as a new leaf node in the growing proof tree, recording not only the starting or previous entailment, but the resulting entailment(s) and the rule that was

applied. In this manner, the cyclist framework would not only identify whether a given entailment was satisfiable, but also generate a proof tree that supported this result. By augmenting this proof tree structure to record the inferred frame and anti-frame fragments alongside the rule application, it is possible to make a record of all inferred fragments and assemble the completed frame and anti-frame from these partial records once the proof has been fully constructed.

Proof Rules

The implementation of the proof rules of our first bi-abductive technique was one of the key aspects of the development of the automated system. Once again, the approach of the Cyclist framework aligned somewhat closely with the approach of our system, and as a result, the design was adopted for our automated tool.

The cyclist framework operated via a breadth-first search over a set of proof rules. As briefly mentioned above, these proof rules are implemented as a function that takes a proof tree and an entailment and, should the relevant conditions be met, produces a new entailment or set of entailments representing the result of the rule application. These new results are subsequently added into the relevant location in the constructing proof tree, with the search continuing from that new point, backtracking if needed.

This mechanism was easily adapted into an implementation of our own search algorithm, updating the breadth-first algorithm into a depth-first search and ensuring that the rules are searched in a specific order. As the rules are stored in an array, which is checked sequentially by the algorithm, this aspect of the search was simply ensured by arranging the rules in the sequence necessary and by ensuring the search ends on the first valid rule application.

The rules themselves were implemented following the function design of Cyclist, taking an entailment and constructing a new node or nodes for the constructing proof tree. The rule functions followed a consistent design pattern, examining the entailment to see if the entailment matches the state expected by

the rule application and checking any side conditions the rule may require. If all these conditions are met, the function "applies" the rule it represents, calculating either a set of changes or generating a new entailment representing the result of the rule application. This resulting entailment, or entailments for rules which cause a branch in the proof, are transformed into nodes for the proof tree and added into the tree.

6.2.2 Limitations

The implementation overall is a fairly accurate representation of the initial combined-domain bi-abductive system. While the search algorithm is a direct implementation of the technique, there were a number of restrictions and limitations that needed to be introduced in order to ensure the automated tool could operate in an effective manner. In addition to the restrictions introduced, there were a number of naturally arising limitations - particularly in terms of the operation of the system - that will also be discussed below.

Operational Limitations

As the tool was developed first and foremost as a proof-of-concept, the system does not include a number of features that would make it more user-friendly, hampering the overall usability of the tool. While these limitations do not affect the core mechanisms of the bi-abductive system, they do limit the overall usefulness of the tool beyond the testing and evaluation performed here.

The most notable limitation in this area is a lack of a graphical user interface, with the tool instead being operated through the command line interface of the operating system as initially implemented in the Cyclist library. While the lack of this GUI has a noticeable impact on the overall usability of the system, it was not considered a key issue; the primary focus for this implementation was on testing and evaluating the tool itself. However, this lack of GUI does have a number of notable issues: first, the control of the bi-abductive system is quite unwieldy,

with the user required to manually enter and use command-line arguments in order to operate the implementation, which includes the filepaths of the benchmark entailments that were used in the evaluation. Though this method of operation is functional, it is not ideal, making the selection of input inconvenient and placing further demands on any potential end-user. Secondly, detailed output of the results of the bi-abductive system could be extremely difficult to parse. While only a minor issue for smaller entailments, larger entailments often spanned several lines in the command line, with minimal additional structure to make the output more easily readable. As a result, it could be difficult to examine and parse the output of the system once deployed, though this could be simplified with reduced output and practice, neither of which are ideal solutions.

Rule Limitations

In addition to the user-facing limitations, there were also a small number of restrictions applied to the rule-set. While not ideal, a small number of compromises were necessary in order to effectively apply the proof rules in an automated setting, typically constraining their application for the sake of efficiency.

One of the most significant limitations introduced was the restrictions applied to the $*$-INTRO rule. In the initial technique, the $*$-INTRO was to able identify and separate not only single terms, but whole sub-formulae from the antecedent that satisfied the entailment problem for fragments from the consequent. This capability allowed for sub-entailments such as

$$x{\mapsto}y * y{\mapsto}z \vdash ls(x, z)$$

to be identified and separated out in a single application of the rule. Unfortunately, the initial performance of the automated tool utilising the unrestricted variant of the rule was quickly be found to extremely poor for even relatively small bi-abductive problems. Due to the need to check every rule in the sequence during the proof search, as well as the relatively early positioning of the $*$-INTRO rule in that sequence, the early implementation was being slowed dramatically by the cost of checking for possible applications of the $*$-INTRO rule. This dramatic

cost to the speed of the implementation was simply too significant to allow, and as there was no known way to reduce the cost of the check, the decision was made to restrict $*$-INTRO to identifying and removing only terms that were directly equivalent. While this approach does restrict the effectiveness of the $*$-INTRO rule, the rule is still able to operate effectively enough that further applications of the other rules in the sequence will accomplish the same effect.

Another restriction the rules of the technique was a restriction on the R-EX rule. This rule is designed to instantiate existential variables in the consequent of the entailment, allowing for subsequent rule applications to identify matches. When done by-hand, this rule would be applied in a targeted manner, using human intuition to identify suitable matches. However, an automated variant of the rule lacks the capacity to reason about suitable instantiations, which in the unrestricted implementation caused the system to reach dead-ends due to the rule choosing substitutions that would not introduce a match. While such applications of the rule would be both sound and valid, such applications do not advance the overall proof. In an effort to reduce these dead-ends, the R-EX rule was grouped with a selection of subtraction rules via a modification of the "tactic" mechanism of the Cyclist library. This mechanism allows for rule applications to be grouped into a form of chained application, ensuring that a particular sub-sequence will only produce a result if all the rules in that group succeed. In this manner, the R-EX rule was placed in sequence with the selection of subtraction rules, ensuring that R-EX can only be applied in scenarios where the substitution allows for a subsequent subtraction. As with the other rule restrictions, this limitation does not effect the overall capabilities of the system in a meaningfully negative manner, but does have a positive effect on the overall performance.

Solution Validation

Outside of the restrictions outlined above, the implementation matches the base technique closely. Further limitations are present, however. The most significant limitation present in the implementation is the relatively limited and ad-hoc validation of the results produced from the application of the list and tree technique.

The list-and-tree technique, due to the case split rule applied early in the proof process, produces a number of potential solutions, with each solution mapping to a specific configuration of the shape predicates present in the entailment. While the majority of these solutions are valid, there is a small chance that one of the solutions requires some heap component or constraint that is simply unsatisfiable[1]. In an attempt to identify and filter out these results from the set, a final validation check was implemented. This validation check, it its most basic form, was a re-application of the system, though one restricted to rules without inference capabilities. By restricting the rules in this manner, this secondary proof is reduced to what is essentially a basic separation logic entailment prover. By passing the original entailment - extended with the candidate frame and anti-frame - into this restricted variant of the search, unsatisfiable solutions may be identified. Unfortunately, despite the relative efficiency of the core proof search, this approach can scale poorly. As will be discussed in the following experimental results section (Chapter 7), this has the unfortunate effect of introducing large costs to the overall execution time, with examples containing a large amount of shape predicates suffering the greatest impact.

A number of alternate approaches and potential solutions were identified during the course of this project, such as making use of an existing SAT or SMT solver alongside methods to translate a separation logic formula into a first-order equivalent, as in [Qiu et al., 2013, Piskac et al., 2014b, Piskac et al., 2014a], but such modifications were eventually decided against due to the prototype nature of the tool. Some of these techniques were eventually integrated into the subsequent implementation, which will be presented below.

[1]One of the most notable examples is the `qf_ls_entl/...` example of the SL-COMP benchmark set. This entailment is ultimately invalid, but the system does not detect the violation of separation constraints during the inference, though it is readily apparent when viewed upon return.

6.3 Generalised System Implementation

The generalised system, much like the technique that underpins it (Chapter 5), can be seen as a generalisation of the first system and the generalised implementation follows in that style. The basis of the generalised implementation is the list-and-tree combined domain bi-abductive prover built for the proof-of-concept technique, adapting and extending a number of the underlying modules in order to permit and support the wider range of properties necessary for the implementation of the generalised technique.

As before, the core of these changes were made to the modules responsible for the representation of separation logic symbolic heaps, introducing further fields for Arithmetic equations and Bags, as well as modifying and extending a number of functions and mechanisms in order permit full and effective use of these new records of constraints.

6.3.1 Extensions

The designs of the supporting Bag and Arithmetic modules were more complex than the other supporting modules, operating in a restricted yet effective manner. Both of these constraints were designed to operate as a set of definitions, permitting the presence of bags or arithmetic equations only when they form part of an assignment. In this representation, it is not possible for a "raw" equation or bag to be present in the entailment, which is to be expected in the context of bi-abductive inference.

Bags

A Bag constraint in this context consists of a destination variable for the assignment, followed by either an atomic definition of a Bag (including the empty set) or an formula describing a bag as a result of a bag operation such as union or intersection. Though an atomic definition could also be folded into the equali-

ties set of the heap record, it was initially separated in order to group the bag constraints together and to simplify the creation process. Another addition to the bag module was an simplification method applied at the creation of a bag object. In the cases where a bag is defined as a product of an operation between two atomic bags, the module will solve the operation and create an atomic definition instead of the compound definition that was initially attempted, simplifying the representation. While such a simplification is not necessary, it improves the readability of the output overall.

At present, there are three bag operations supported by the implementation, Union, Intersection, Subtraction, with an additional representation for subset relations. These operations were selected in order to ensure that the majority of possible shape definitions and constraints could be represented within the implementations.

Arithmetic

The arithmetic module follows the general structure of the Bags module, constructing an arithmetic constraint as a definition only. As with the bags, this limitation is unlikely to have an effect on the overall effectiveness of the technique or the implementation, as an equation in isolation should not typically appear in a bi-abductive entailment problem.

The greatest complexity in the arithmetic module was the need to include mathematical functions beyond simple binary operations. While the size property of lists can be described and solved by a simple fragment with only addition, representing the depth of trees requires a more complex range of capabilities. In particular, the equation to determine the depth of a tree requires a MAX function to determine the depth of the largest subtree. This equation also includes three arguments and two "operators", necessitating a more complex representation for arithmetic formulae.

A representation based around a *Reverse Polish Notation*, or *Postfix* notation

was eventually selected. This representation was chosen for a number of advantages, with one of the most significant being the ability to eliminate parenthesis while preserving the correct order of operations. By basing the systems internal representation of the arithmetic equation on this format, many of the difficulties around operator precedence is resolved, as the format naturally models the order in which operations must be carried out. Mechanically, an arithmetic equation is modelled as the combination of an operator (or function) and a list of equations representing the arguments, either atomic arguments or another equation object. For a given equation, the "top-level" arguments and operation is the final operation that would be carried out during an evaluation, with all "child" equations being implicitly higher in the order of operations.

The construction of these Arithmetic records is achieved by an adapted version of the Shunting-Yard algorithm, first developed by Dijkstra. The arithmetic equation is parsed through the same monadic parsers used throughout the project, producing a list of Tokens. These tokens are essentially a top-level wrapper for the terms and operators, allowing for both tokens to exist in the same collection and solving the issue of handling operators and values simultaneously. These tokens are then passed into the shunting-yard algorithm, passing arguments into the "output" stack and pushing operations and functions alike into the operation stack, producing the final output. This process also allows for structural issues in the equation to be detected, preventing the tool from attempting to solve invalid entailments.

A key difference between the canonical version of the algorithm and this adapted versions occurs during the operation to output operators. Instead of simply outputting the operation, the modified variant instead "applies" the operation, constructing an Equation record from the operator and arguments already output and replacing them in the output stack. This construction can consume both atomic arguments and equations, emulating the effect of the order of operations. Once the algorithm completes, the system should have consumed all of the parsed input, chaining Equation record constructions and resulting in a single "top-level" Equation record representing the entire arithmetic equation.

Though this representation permits a great deal of freedom in representing arithmetic equations, the various functions designed to interact with this structure must be able to operate recursively, descending into the arguments to continue the process. Though the designs for these functions follow naturally, there is naturally a potential cost to efficiency.

The design of the arithmetic representation does have a number of significant restrictions, however. Due to the underlying structure of the symbolic heap utilised throughout the Cyclist framework and the implementations developed during this project, it was not possible to effectively represent equalities or ordering constraints that utilised formulae as an argument. While such constraints may still be constructed through the use of an intermediary variable representing the arithmetic formula, this does introduce some overhead and additional efforts required. This additional requirement would become an issue when attempting to parse a number of the examples utilised in the evaluation of the implementation (Section 7.3), though an mechanism to automatically introduce the intermediary was constructed to resolve this.

Generalised Proof Rules

The implementation of the Generalised proof rules was the most important aspect of the implementation, and was the area in which the most time was spent. The rules were designed in the same manner as the previous system implementation, as a series of functions transforming the entailment from a state matching the premiss of the rule into a state satisfying the conclusion. These rules are arranged into a predefined sequence, as described in Chapter 5, with the system attempting to find a suitable rule to apply from the rule-set.

A number of the rules of the generalised system had already been implemented as part of the automation efforts made towards implementing System 1, and as such, could largely be carried forwards into the development of System 2. These rules were typically rules that operated over sections of the symbolic heap independently of its contents and thus would not be affected by the generalisation

process and included a number of the terminating axioms of the system such as EMP and INF-PURE. However, in an effort to ensure that these imported rules would operate as effectively as possible, the implementations were reviewed, aiming to ensure both the correctness of the implementation with respect to the core bi-abductive technique, as well as the overall quality. A number of minor refinements were made as part of this process, though no significant changes were made.

The rest of the rules of the generalised technique needed to implemented from the ground up, with a number of rules requiring the introduction of a generalised unfolding mechanism to operate effectively. Again, the existing mechanisms of Cyclist were of significant use here, as the framework already has a general unfolding mechanism in place. As the shape predicates utilised by my own tool followed the design of the Cyclist predicate representation, this mechanism could be directly adopted in the generalised implementation.

As with the rules implemented in System 1, these rules were built as functions that examine an entailment to see if specific conditions were met, and if so, would continue onwards to applying the rule to the entailment, either though introducing new states or constraints or through the elimination of specific constraints. These applications are recorded as a node on a proof tree, which is returned alongside the final inferred fragments once the search is concluded.

6.3.2 Limitations

While the generalised tool was constructed to a higher overall quality than the prior list-and-tree system, there were still a number of notable limitations that hampered the overall performance. These limitations, though often relatively minor in their impact, can have a noticeable effect on the performance and capabilities of the system.

A limitation shared with the list-and-tree system is the lack of a refinement mechanism. While not directly affecting the soundness of the solutions, a number of the outputs produced by the tool may include terms or aspects that could be

simplified or removed from the output. While many of these inefficient representations are in common with the previous tool, with examples including the presence of empty-equivalent shape predicates or equality artefacts from the internal unfolding processes, a small number of additional inefficiencies are introduced as a result of the more expressive fragment used by the generalised tool. The most common example arises from the representation of arithmetic formulae, in which many arithmetic formulae could be quickly and directly refined into a more compact form. These formulae most commonly appear as sequences of smaller formulae, as in

$$\exists y, z.\ x = y + 1 \wedge y = z + 1 \wedge z = 0$$

and it can be easily seen how these formulae could be simplified to a single equation. While the absence of such refinement mechanisms does produce a less visually clear set of results, the results themselves are still suitable, making the introduction of a simplification pass a task for future works.

Another shared limitation is the validation stage of the analysis. Though both systems aim to identify invalid solutions at the point of inference, some failed inferences may not be identified until the inferred frame and anti-frame are introduced into the initial entailment. As in System 1, this process is currently handled by passing the new entailment back into the proof system, now using a restricted non-inferring rule-set. While this process is functional, the experimentation carried out over System 1 clearly indicates that the validation can be an extremely expensive stage of the overall analysis. As a result, the overall performance is notably affected by the number of potential solutions produced by the tool. These "candidate solutions", if numerous enough, can cause timeouts after the construction of the proof, resulting in an `UNKNOWN` output as a result. While this is a rare occurrence, the limitation is still present and worth observing.

A final known limitation of the generalised implementation was the (necessary) restrictions over shape predicates. While the implementation is able to function over a wide range of shape predicates, many of the proof rules operate under specific assumptions about how the predicate is structured and how it would unfold. As a result of this, certain shape predicates, such as odd-even lists or

tail-unfolding lists, cannot be accepted in the current version as specific unfolding behaviours will fail to effectively progress the analysis. Though this was not initially expected to be a significant issue, the experimental results highlight the impact this restriction has over the expressiveness of the tool and are discussed in greater detail there.

Chapter 7

Experimental Results

In this chapter, we will present and discuss the experimentation undertaken as part of this PhD project. We will first present the overall aims and approach taken to the experimentation during the project in Section 7.1 before leading into the experimental results gathered for each of the automated systems presented in Chapter 6, with results for the list-and-tree implementation presented in Section 7.2 and results for the generalised implementation presented in Section 7.3. Finally, an analysis and overview of the results will be presented in Section 7.4.

7.1 Overview

The primary goal of the experimentation performed and reported in this chapter was to establish the overall quality and effectiveness of the combined domain bi-abductive inference tools developed during this project (Chapter 6). While the preferred experimentation would be an equivalent comparison between the tools developed here and the other combined domain bi-abductive inference tools already known in the literature [Trinh et al., 2013, Qin et al., 2017, Fragoso Santos et al., 2020], the lack of program analysis capabilities in the implementations prevent this. Additionally, due to the anticipated difficulty of extracting the bi-abductive inference mechanisms from the tools in the literature, a fair and

direct comparison between the systems does not appear possible at this time; comparisons between our tool and the full verification tools of the literature are likely to be of extremely limited value due to the differences in methodologies and inputs. As a result of this, an alternate testing method was deemed necessary.

Functionally, the implementations of our technique are entailment provers with additional bi-abductive capabilities. The tools accept an entailment and attempt to prove its validity, applying bi-abductive inference should some dead-end be identified. As a result of this, the techniques developed during this project could serve as an alternate method for entailment proving within verification systems, with the embedding of the implementations into an existing verification tool - with appropriate adaptations - potentially being sufficient to enable bi-abductive reasoning within that tool. While this integration was ultimately outside of the scope of this work, such an application can still serve as a potentially useful point of evaluation. By comparing our bi-abductive techniques against other similarly capable entailment provers, it is possible to examine the potential performance impacts of the introduction of combined domain bi-abductive inference into an existing setting.

In order to perform this evaluation, it was determined that the SL-COMP competition[1] would provide a good starting point. The SL-COMP competition [Sighireanu et al., 2019a] is aimed at the evaluation of techniques and systems designed to solve satisfiability and entailment problems in separation logic. The competition itself consists of several "divisions", with each division representing a particular subset of problems. One example of this is the `qf_shls_entl` division, which consists of entailment problems based around shape-only singly-linked lists in the quantifier-free symbolic heap fragment of separation logic. These examples provide a relatively large and robust set of benchmarks for use in the evaluation of our implementations, covering a large range of shape predicates and properties.

[1]The benchmarks were initially downloaded in November 2019, though no further changes were noted.

7.1.1 Results and Metrics

Under the test conditions described above, there will be a range of metrics gathered from the experimentation. From the example itself, we will gather the name of the example tested, the anticipated result and the actual returned result. Alongside these data points, two timing data points will be recorded from within the implementation: the time taken to complete the proof tree, which in turn identifies the potential returned values, and an overall execution time, which will include the time to construct the proof as well as the time required to extract and verify the possible results. As discussed in Chapter 6, the verification of the results is an important stage of the analysis process, and as can be seen in the results, often takes a significant portion of the overall execution time. These timing statistics are recorded from within the implementation itself, utilising a function that calls and records system clock time before the start of the analysis process and immediately upon return. This timing mechanism is applied twice during an analysis, once for the construction of the initial proof tree and once for the subsequent extraction and validation of results, recording the proof time separately in order to output both the proof time and overall time separately as needed.

Another key data point gathered during the experimentation will be the frames and anti-frames produced by the implementations when applied to the entailments. As the example entailments are aimed at testing entailment provers, there are no pre-defined results for valid inferred fragments for initially invalid entailments. In order to ensure the tool is producing valid results, a number of the outputs produced by the implementations will be examined separately, with any invalid results noted alongside the rest of the experimental data. A subset of these results will also be presented alongside the entailments themselves in order to more effectively present the outputs produced by the implementations, as well as to present the results for review.

For the expected and actual results, an unusual scheme is followed: valid entailments will return `UNSAT` and invalid entailments will return `SAT`. This rep-

resentation is inherited from the examples themselves, where entailments are represented in an SMT-appropriate format of a pair of assertions: an assertion establishing the antecedent and an assertion establishing the non-satisfiability of the consequent. In such a case, a valid entailment would be unsatisfiable, as no solution would exist which ensures the antecedent holds while the consequent does not.

The benchmarking for both systems was performed on an Ubuntu 20.04 virtual machine, running on an Intel i5-10210U processor. A 30s limit on execution time was enforced throughout these experiments. Due to the large number of examples tested during the course of this evaluation, only excepts of the total results set is presented here, though all results were considered during the writing. To allow the reader to examine the full benchmark set, the data may be accessed via OneDrive[2].

7.2 System 1 Experimentation

The experimentation performed over the initial list-and-tree system focused on the `qf_shls_entl` division of the SL-COMP competition, which consisted entirely of entailments utilising singly-linked lists, and a small amount of examples sourced from the `qf_shidlia_entl` division that made use of sorted singly-linked list segments, modified to match the sorted list predicates supported by the tool. Each of these benchmarks operated within a quantifier-free fragment of separation logic that was supported by the list-and-tree system and allowed for examination of the tool against a strong standardised set of benchmarks.

In total, the list-and-tree implementation was applied to 313 entailments: all 296 *ls*-targeted examples from the `qf_shls_entl` division - 174 of which described immediately valid entailments and a further 122 entailments that were invalid without some additional inferred constraints - alongside 17 sorted list examples sourced from the `qf_shidlia_entl` division, modified to use the *sls* definition

[2] `https://1drv.ms/u/s!AmMbJr1N3MyJkj65EuGgabrDj56Q?e=B05682`

Category	Valid	Invalid
qf_shls_entl		
smallfoot	23	54
clones	40	60
bolognesa	57	53
ls	6	3
qf_shidlia_entl		
sls_join	17	0

Table 7.1: Breakdown of list benchmark sources

integrated into the system (Section 2.3a). These number of valid and invalid examples present in the division, divided by the source of the benchmark, are presented in Table 7.1.

For each valid entailment in the set of benchmarks, our implementation aims to identify either an empty frame and anti-frame or a trivial one, where the final impact of the inferred fragments is essentially negligible - often in such cases, these fragments simply describe previously implicit constraints. For invalid entailments, our implementation either identifies a suitable frame and anti-frame, ensuring that the refined entailment is valid, or determines that there is is no possible solution for that particular benchmark.

The benchmarks will be presented in three parts: first, we will present and discuss the results of applying the implementation over examples featuring basic singly-linked lists with no ordering constraints. Subsequently, we will discuss the results of the sorted list benchmarks, and finally conclude with the results of applying our technique to examples utilising trees, both simple binary trees as well as binary search trees.

7.2.1 Simple Singly-Linked List Benchmarks

As will be discussed in the evaluation of the generalised system, the `qf_shls_entl` division is one of the largest divisions in the competition, likely due to the number and maturity of tools in this specific area. As a result of the volume of benchmarks examined, the analysis of these results will be somewhat general, though a small number of specific results from each category will be presented. Additional focus is placed on the Smallfoot benchmarks due to the similarities between that tool and our own.

Smallfoot

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
smallfoot-vc09	UNSAT	UNSAT	0.975	2.24
smallfoot-vc21	UNSAT	UNSAT	0.141	0.183
smallfoot-vc22	SAT	UNSAT	0.276	0.558
smallfoot-vc23	SAT	UNSAT	0.152	0.28
smallfoot-vc29	SAT	UNSAT	0.598	1.105
smallfoot-vc33	SAT	UNSAT	0.809	1.958
smallfoot-vc37	SAT	SAT	0.866	1.15
smallfoot-vc40	SAT	UNSAT	0.608	1.192
smallfoot-vc65	UNSAT	UNSAT	0.422	1.113
smallfoot-vc70	SAT	UNSAT	1.437	3.891

Table 7.2: Benchmarks for Smallfoot Example

Our implementation proved to be highly effective over the Smallfoot benchmarks, typically identifying and evaluating all found solutions in only a few ms of CPU time. Our technique was able to solve the vast majority of the examples within this set, either confirming the validity of the entailment or identifying some suitable frame or anti-frame to correct it. The execution times and validation percentages of a subset of the benchmarks of this set can be seen in Table

7.2.

Many of the benchmarks in the Smallfoot set are essentially equivalences, in which the antecedent and consequent are two descriptions of a symbolic heap. As a result, the majority of the work undertaken by the tool is to show those equivalences through match-and-subtract operations, unfolding and eliminating shape predicates from the entailment where possible. Other examples featured aliased pointer fields, requiring the inference of an equality enforcing that alias for the entailment to be satisfied. Of note in this category is the `smallfoot-vc37` benchmark, where the core alias is applied to the pointer to the node instead of the more typical alias over the values stored within the node. At the time of testing, the list-and-tree system does not possess the capabilities to handle the inference of such aliases, preventing the identification of a suitable frame and anti-frame. As a result of this limiation, the tool is unable to "correct" the entailment, which in turn causes the analysis to (correctly) return a verdict of `SAT`.

Another important aspect to note is that several of the examples within this set were made trivial from simplifications applied during the initial parsing of the entailment from file. Due to the chosen representation of equalities in the Cyclist SL library, trivial equalities are naturally filtered out without any input or application of proof rules. As a result, a small number of the benchmarks featured either antecedents or consequents that were reduced to $\texttt{true} \wedge \texttt{emp}$ immediately, vastly simplifying the entailment as a whole. In such cases, the tool represents the entailment to be analysed in that simplified form, with several examples being reduced to $\texttt{emp} \vdash \texttt{emp}$; while equivalent to the original input, the simplifications applied to reach this point are not detailed in the proof, though the equivalence itself is apparent on review.

However, the most relevant examples in this set are those that require bi-abduction in order to be effectively resolved. A number of the benchmarks of the Smallfoot set feature entailments with aspects of the symbolic heap that are missing or extra within that entailment. In these cases, our technique must identify those fragments, and infer that they form part of the frame or anti-frame. Our technique is remarkably effective in these cases, readily identifying fragments

of the entailment that are missing or extra in the example, and moving them to the frame or anti-frame as appropriate. There are fewer examples in which a spatial fragment is inferred into the anti-frame than into the frame, but these benchmarks are resolved with equal ease.

LS

Alongside the larger groups of entailments, there are also a small number of list-targeting examples sourced from [Enea et al., 2017], listed under the group `ls`. This group of entailments features a number of list segments and connecting points-to terms in the antecedent, and requires that the implementation identify the equivalence of those fragments with a single, unified list over the same points defined in the consequent. In order to resolve these entailments, the prototype must identify each overlapping fragment of the heap and subtract it from entailment.

Over these examples, the list-and-tree implementation performs well, proving each of the 3 valid entailments in less than 5ms CPU time each. Indeed, even over the initially invalid examples, the prototype system is able to identify suitable frames and anti-frames and complete the proof in typically under 10ms.

Clones

The implementation's performance was also evaluated over the set of `Clones` benchmarks. This set of examples feature entailments that include a number of unidentified aliases. In each of these cases, in order to satisfy the entailment, these aliases must be identified, and an equality representing this alias must be introduced into the anti-frame or frame as needed.

The benchmarks are arranged into 10 groups of similar problems, with each group featuring a different number of terms. For the benchmarks in the earliest groups, there are only a small number of terms to be processed, increasing as the benchmarks proceed. In each of these groups, examples `e01` through `e06`

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
clones-05-e07	SAT	UNSAT	112.303	405.434
clones-05-e08	SAT	SAT	385.914	488.572
clones-05-e09	SAT	UNSAT	125.945	550.924
clones-05-e10	SAT	UNSAT	118.661	412.916
clones-06-e07	SAT	UNSAT	322.856	1485.969
clones-06-e08	SAT	SAT	1465.989	1806.74
clones-06-e09	SAT	UNSAT	395.007	2038.484
clones-06-e10	SAT	UNSAT	321.687	1473.443
clones-07-e07	SAT	UNSAT	978.656	5210.374
clones-07-e08	SAT	SAT	5400.865	6558.041
clones-07-e09	SAT	UNSAT	1315.282	8163.251
clones-07-e10	SAT	UNSAT	1068.186	6274.345
clones-08-e07	SAT	UNSAT	3324.708	18917.91
clones-08-e08	SAT	SAT	18494.566	21974.284
clones-08-e09	SAT	UNSAT	3824.512	28160.246
clones-08-e10	SAT	UNSAT	3034.947	21004.405

Table 7.3: Selection of Clones Benchmarks

are relatively trivial problems featuring empty sides of the entailment or simple equivalences, and are solved either by simplification during the initial construction of the entailment from the benchmark file, or by early applications of one of our axioms. The remaining examples in each group are more complex, and are thus the only ones discussed in detail here; a selection of these benchmarks can be found in Table 7.3.

The examples within the earlier groups are solved handily by our implementation, with all examples in groups `clones-01` through `clones-05` solved in less than one second of CPU time, including validation of results. However, beyond this point the size of the entailments begin to have an increasingly large impact on the time required to complete, with `clones-06` taking over a second for many

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
bolognesa-10-e03	UNSAT	UNSAT	27.108	108.891
bolognesa-10-e04	SAT	UNSAT	27.762	108.354
bolognesa-10-e07	SAT	UNSAT	22.585	91.18
bolognesa-10-e09	SAT	UNSAT	17.624	62.9
bolognesa-12-e03	SAT	UNSAT	461.298	2335.377
bolognesa-12-e04	SAT	UNSAT	51.263	225.285
bolognesa-12-e07	SAT	UNSAT	86.138	407.951
bolognesa-12-e09	UNSAT	UNSAT	236.53	1213.385
bolognesa-14-e03	UNSAT	UNSAT	272.295	1550.829
bolognesa-14-e04	UNSAT	UNSAT	67.704	292.969
bolognesa-14-e07	UNSAT	UNSAT	170.403	882.233
bolognesa-14-e09	UNSAT	UNSAT	273.686	1508.229
bolognesa-16-e03	SAT	UNSAT	272.25	1711.448
bolognesa-16-e04	SAT	UNSAT	318.775	1798.791
bolognesa-16-e07	UNSAT	UNSAT	475.66	2977.077
bolognesa-16-e09	UNSAT	UNSAT	451.557	3007.654

Table 7.4: Selection of Bolognesa Benchmarks

of the examples, and dramatically increasing from there. Indeed, examples in `clones-07` takes over 5 seconds to complete, and the implementation begins to time out from `clones-08` onwards.

In each group, our implementation was able to solve and identify suitable aliases for each instance of `e7`, `e9` and `e10` (timeouts notwithstanding). In most instances, this process was undertaken via the application of the INF-PTO rule, identifying an alias implied by the shared root of points-to terms in the antecedent and consequent, and applying it to the entailment as a whole. This in turn allows for the remainder of the proof to be resolved as standard, matching and subtracting equivalent fragments. This process was accomplished reasonably efficiently, though the impact of larger entailments on the overall performance is

quite noticeable.

Within each complexity level, example `e08` is a special case. Each instance of these examples within the `clones` benchmarks describes an invalid entailment, an ideal test for the bi-abductive system. However, the system fails to identify a frame or anti-frame that would enable the entailment to hold, one of the extremely few cases where the list-and-tree implementation cannot identify a correction.

When looking at the breakdown in execution times, it seems that the bulk of the analysis time is spent on the validation of the produced results, becoming increasingly costly as the number of terms and potential solutions increases. Indeed, this aspect seems to be common throughout the examples tested here, and while entailments with a small number of terms are solved almost instantly, larger entailments take dramatically more time to complete.

Bolognesa

The Bolognesa examples consist primarily of a large number of interlinking spatial fragments. In order to prove these examples, our technique must identify equivalent fragments of the entailment, often formed of a mix of points-to entailments and list predicates, and eliminate them, with any remaining fragments inferred as a part of the frame or anti-frame. A selection of example benchmarks can be found in Table 7.4.

The Bolognesa examples were arranged into 20 groups of 10 examples, with each group increasing in the number of terms and expressions included within the entailment. Much like the `Clones` examples, our implementation was able to rapidly solve the majority of the smaller examples in very short periods of time; the majority of these earlier problems were solved within only a few seconds of CPU time, validation included. However, as in the other examples, as the number of terms increased, the overall execution times increased dramatically. While a small number of exceptions were present within the benchmarks, the larger entailments took significantly more time to complete as the number of

terms increased, with the benchmarks of the later groups needing the majority of the 30s execution time to complete with a small number of timeouts.

With that said, when the implementation did complete, the results showed promise: for the majority of the invalid examples, our implementation produced a range of potential valid solutions for the frame and anti-frame, usually in an acceptable timeframe.

7.2.2 Sorted List Benchmarks

Benchmark `sls`	Expected	Actual	Proof Time (ms)	Total time (ms)
join_2_known_bnd	UNSAT	UNSAT	2.296	4.855
join_2_no_cond	UNSAT	UNSAT	1.999	4.937
join_2	UNSAT	UNSAT	2.105	5.453
join_2_unk_lower_bnd	UNSAT	UNSAT	1.914	5.358
join_2_unk_upper_bnd	UNSAT	UNSAT	2.13	5.399
join_3_2_cond	UNSAT	UNSAT	7.641	22.939
join_3_no_cond	UNSAT	SAT	6.195	6.284
join_3	UNSAT	UNSAT	7.114	21.946
join_3_unk_lower_bnd	UNSAT	UNSAT	7.061	23.277
join_3_unk_upper_bnd	UNSAT	UNSAT	6.167	11.008
join_4_1_cond_unk_both	UNSAT	UNSAT	18.836	33.453
join_4_1_cond_unk_lower_bnd	UNSAT	SAT	17.484	17.627
join_4_1_cond_unk_upper_bnd	UNSAT	SAT	18.638	18.835
join_4_2_cond_unk_lower_bnd	UNSAT	UNSAT	19.916	40.407
join_4_2_cond_unk_upper_bnd	UNSAT	UNSAT	18.656	45.132
join_4_3_cond	UNSAT	UNSAT	22.874	76.558
join_4_no_cond	UNSAT	SAT	17.295	17.401

Table 7.5: Benchmarks for Sorted List Entailments

The *sls* benchmarks utilised in this section of the evaluation were sourced from

the `qf_shidlia_entl` division of SL-COMP and adapted to be supported by the language fragment of the list-and-tree system. This adaptation is minor, adjusting the shape predicate utilised by the example to match the predicate embedded within System 1 and carrying that modification through to the entailment itself; the example is unchanged by these modifications.

Unfortunately, there are only a small number of *sls*-targeted examples within the SL-COMP benchmarks, and so while the results here are a reasonable indicator of the effectiveness of System 1, the sample size remains relatively small. Regarding the examples themselves, each of the benchmarks examined in this section are valid entailments, each aiming to demonstrate that several indirectly linked sorted lists entail a single combined sorted list over the same range. The primary difference between each example is the amount of explicit information regarding the values of each segment. For some of these examples, the exact bounds are known, making the ordering between each segment trivial. However, for the majority of the examples, only a selection of the bounds are known, requiring the identification of ordering constraints that will enforce the order between the segments. In order to successfully solve these examples, our technique must be able to identify those ordering constraints, and then identify that the multiple linked segments are equivalent to a single sorted list segment between the same two points. The results for this section are detailed in Table 7.5.

In summary, our technique was able to solve the sorted list examples in very short periods of time, with the longest execution time only just exceeding half a second. For each of the cases, implicit or missing ordering constraints were identified, and as in the simple linked-list examples, each valid state of the shape predicates are explored separately, often producing a range of valid specifications upon completion. Curiously, the performance of the tool seems to be better over examples with *fewer* known boundaries, suggesting that the more general symbolic ordering constraints are simpler to identify with our technique than ordering constraints with known values. More investigation into this phenomenon may be undertaken at a later date.

There are some minor issues, however. Of particular note is the presence of

Entailment	**Time (ms)**
$x{\mapsto}[l,r] \vdash tree(x)$	2
$x{\mapsto}[l,r] * tree(l) * tree(r) \vdash tree(x)$	9
$tree(x) \vdash x{\mapsto}[l,r] * tree(l) * tree(r)$	7
$tree(x) \vdash tree(x) * tree(y)$	3
$x{\mapsto}[l,r] * l{\mapsto}[m,n] * r{\mapsto}[s,t] \vdash tree(x)$	70
$tree(x) * tree(y) \vdash x{\mapsto}[i,j] * y{\mapsto}[k,l]$	62
$x{\mapsto}[l,r,i] \vdash stree(x,i,j)$	7
$x{\mapsto}[l,r,i] * stree(l,a,h) * stree(r,j,k) \vdash stree(x,a,k)$	18
$stree(x,a,k) \vdash x{\mapsto}[l,r,i] * stree(l,a,h) * stree(r,j,k)$	16
$stree(x,a,b) \vdash stree(x,c,d) * stree(y,i,j)$	4
$x{\mapsto}[l,r,i] * l{\mapsto}[m,n,a] * r{\mapsto}[s,t,k] \vdash stree(x,a,k)$	196
$stree(x,i,j) * stree(y,k,l) \vdash x{\mapsto}[a,b,i] * y{\mapsto}[c,d,k]$	114
$stree(x,i,j) \vdash tree(x)$	2
$stree(x,i,k) \vdash x{\mapsto}[l,r,i] * tree(l) * tree(r)$	14

Table 7.6: Benchmarks for Tree Entailments

the trivial numerical terms within the final outputs. Terms such as $100 \leq 200$ are obviously true, but appear within the final output as a side-effect of our simplistic treatment of numerical terms. While modifying the implementation to eliminate such terms would not be overly complex, their presence does not cause a significant negative effect, and so has been ignored for now.

While these examples are relatively simplistic, the ease with with our technique was able to identify the requisite ordering constraints, either explicit or implicit, shows some promise for the approach taken, which will be further tested in the evaluation over the generalised system.

7.2.3 Binary Trees

Unlike the other experiments performed in this evaluation, the examples used for exploring the effectiveness of our technique over trees were created by-hand due to difficulties in identifying relevant examples supported by the tool's current implementation. These benchmarks were created in the spirit of the Smallfoot benchmarks, aiming to evaluate the implementations performance and capabilities over a range of fairly "primitive" entailments. These entailments and their corresponding results can be seen in Table 7.6.

For the majority of the examples tested here, it can be seen that the performance of the tool over the tree-based examples is comparable in many ways to the performance over the SL-COMP examples. Over these entailments, the tool was able to create a proof tree identifying a number of possible solutions in reasonably short times, typically under 100ms. However, as in the previous examples, the amount of time spent validating these examples is significantly larger than the initial identification. In particular, entailment 5 of this benchmark set spent almost the entirety of its execution time validating the potential solutions, likely due to the inferred anti-frame including a number of additional predicates.

With that said, the performance over the *stree* entailments does continue to indicate that the core of the list-and-tree system shows promise. The construction of the initial proof tree, including the inference of frames and anti-frames, continues to take only a short time over these simpler entailments, and while the verification continues to be a source of extreme inefficiency, the overall CPU time required remains quite brief. Additionally, due to the similarities between the entailments using *tree* and *stree* predicates, it can be seen that the inclusion of ordering constraints does not seem to introduce a large cost to performance. As in the *sls* examples previously, the technique reliably identified the omitted ordering constraints necessary for the entailments to hold, as well as identifying any spatial fragments of the frames and anti-frames, with the overall execution time increasing by only a few ms.

In addition to the more standard entailments, we have also included 2 spe-

cial cases, aiming to examine the performance of the tool when GENERALIZE rules are required to solve the bi-abductive problem. In each example, the antecedent features a binary search tree, whereas the consequent describes only simple binary trees. In order to resolve these examples, the technique must identify the relationship between the two predicates and generalize the *stree* predicate to a simple *tree*, before continuing on to complete the proof. As can be seen in the experimental results, the need to apply one of these rules does not have a significant effect on the overall execution time when compared to similar entailments.

7.2.4 System 1 Analysis

Overall, the list-and-tree system appears to be reasonably efficient and effective over the basic list examples detailed here. Despite some issues regarding the scalability of the proof search over large entailments, the search overall was typically effective, accurate, and fast in the majority of cases, producing useful and accurate frames and anti-frames where needed. From these benchmarks, it appears that the largest factor in the execution time is the verification of the potential results, not the preceding calculations. Indeed, throughout the benchmarks, it is apparent that the majority of the execution time for a given entailment is spent upon the validation of the results, not the construction of the underlying proof. While this is clearly not ideal, the speed at which our technique is able to explore and identify all possible solutions for a given entailment does indicate that the list-and-tree system is effective over its restricted fragment, and that a more efficient validation mechanism, possibly taking advantage of SMT-based solutions, may help overcome many of these issues.

7.3 System 2 Experimentation

The experimentation applied to the list-and-tree system was subsequently applied to the generalised system developed during this project. While the core of the experimentation remains the same, exploring the overall quality and effectiveness

of the system when applied to examples taken from the SV-COMP benchmark set [Sighireanu et al., 2019a], the scope of the experimentation has been greatly expanded when compared to the first set of results. Due to the significantly greater expressiveness of the generalised system, the set of supported examples has been expanded: where the first system was restricted to the `qf_shls_entl` division, along with a small number of examples based on sorted lists and binary search trees, the generalised system was able to be applied to all of the quantifier-free symbolic heap divisions, covering examples with a wide range of shape predicates such as skip lists and tree segments.

There are a number of key restrictions and concessions that were necessary for the tool to be correctly applied to some of the examples in this collection, however. One of the most significant of these concessions arises from the representation used to describe these benchmarks. Each of the example entailments present in this division - as well as subsequent divisions - provide definitions for the shape predicates utilised in the example and is parsed and recorded by the tool during the initial setup. These definitions, as well as the entailments themselves, are written using the SMT-LIB representation, and occasionally makes use of notations that do not directly translate to the internal representation of the symbolic heap used in System 1 and System 2. Though this limitation was present in the prior system, the examples utilised in the corresponding experimentation could be mapped to an equivalent internal representation without issue, allowing for the issue to be temporarily ignored. However, arithmetic formulae are expressed in a form which allows for the direct use of formulae as an argument to an ordering constraint or an equality, a mechanism not currently supported in the implementation. In order to resolve this, the parsing of the benchmarks was adapted to introduce some "intermediary" variables into the definitions or entailments where necessary, acting as a substitute and placeholder for arithmetic formulae in other constraint forms. While not ideal, this mechanism is not expected to affect the overall meaning of the benchmarks and so have no effect the soundness or validity of the results, beyond a slight inefficiency in their presentation. As an example, one of the terms in a *dll* shape predicate utilised in the

`qf_shidlia_entl` division is encoded as

$$(>\ dt1\ (+\ dt3\ 1))$$

This would be mapped to the internal equivalent

$$\exists x'.\ x' = (dt3\ 1\ +)\ \wedge\ dt1\ >\ x'$$

which describes the same constraints, though makes use of x' as a intermediary.

An additional, more significant, limitation is introduced through the design of some of the examples. Due to the definition of a well-formed shape predicate used throughout the generalised system (Section 2.5.2), not all of the examples in the examined divisions are fully supported by System 2. While the majority of the examples are supported without issue, a small number of the examples do not meet the conditions necessary for the predicates used to be considered well-formed. Such cases will be identified and discussed alongside the division where examples were encountered.

On final aspect of note is one shared with the prior system; the range of possible solutions presented by the system following the conclusion of the analysis. Due to the case splitting introduced by the EXCLUDE-MIDDLE rule of the list-and-tree system and the predicate matching rules of the generalised system, a range of possible solutions may be identified by tools, of which only a selection may be valid. As in the list-and-tree system, the "candidate" solutions are passed back into a restricted, non-inferring, form of the proof search, aiming to check the validity of the entailment without the identification of any new state information. While this process does ensure that the resulting frames and anti-frames are in fact appropriate solutions for the entailment supplied, the process remains expensive and forms a significant portion of the overall execution time. This remains a significant point of concern for the system overall and the implementation of a more effective approach is one of the most immediate points of improvement for future enhancements to the tools.

Overall, the combined divisions comprised of 729 examples. For the examples, 541 are valid without inference and 188 are not immediately valid, requiring some

Division	Total Examples	Valid	Invalid
QF_SHLS_ENTL	296	174	122
QF_SHID_ENTL	312	280	32
QF_SHIDLIA_ENTL	61	50	11
QF_SHLID_ENTL	60	37	23
Totals	729	541	188

Table 7.7: Breakdown of General System Benchmarks

additional inferred fragment to hold. A breakdown of the number of examples and the validity of those examples by division may be found in Table 7.7.

7.3.1 QF_SHLS_ENTL

The `qf_shls_entl` division is the most basic division present in the benchmarks of SL-COMP, consisting of examples containing only basic singly-linked list predicates and points-to terms, alongside basic pure equalities and disequalities. These benchmarks formed the bulk of the examples utilised in the evaluation of System 1 and are described in detail in Section 7.3.

This division was anticipated to be largely comparable in terms of the identification of solutions, but questions around the potential costs of the generalised rules utilised by System 2 over the targeted rules remained. This division in particular would be the best opportunity to identify any additional costs introduced from the use of generalised proof rules.

Bolognesa

Similar to the results gathered from System 1, each of the Bolognesa examples were successfully solved by the Generalised system, either identifying the existing entailment as a valid solution or identifying potential corrections. System 2 was able to complete the construction of the proof tree and identify candidate solu-

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
bolognesa-18-e03	SAT	UNSAT	115.647	293.603
bolognesa-18-e04	SAT	UNSAT	111.414	304.905
bolognesa-18-e07	SAT	UNSAT	147.645	386.462
bolognesa-18-e09	SAT	UNSAT	132.558	331.775
bolognesa-20-e03	SAT	UNSAT	181.502	419.03
bolognesa-20-e04	SAT	UNSAT	176.229	405.243
bolognesa-20-e07	UNSAT	UNSAT	416.759	1018.304
bolognesa-20-e09	SAT	UNSAT	356.93	927.651
clones-08-e07	SAT	UNSAT	31.846	35.86
clones-08-e08	SAT	UNK	29763.271	29763.271
clones-08-e09	SAT	UNSAT	57.056	62.341
clones-08-e10	SAT	UNSAT	53.745	58.225
clones-09-e07	SAT	UNSAT	58.076	64.505
clones-09-e08	SAT	UNK	29828.746	29828.746
clones-09-e09	SAT	UNSAT	49.834	55.344
clones-09-e10	SAT	UNSAT	41.762	46.076
clones-10-e07	SAT	UNSAT	62.1	67.536
clones-10-e08	SAT	UNK	29699.342	29699.342
clones-10-e09	SAT	UNSAT	95.293	101.528
clones-10-e10	SAT	UNSAT	81.381	86.787
smallfoot-vc09	UNSAT	UNSAT	0.306	0.566
smallfoot-vc21	UNSAT	UNSAT	0.513	0.763
smallfoot-vc22	SAT	UNSAT	0.516	0.816
smallfoot-vc33	SAT	UNSAT	2.3	5.811
smallfoot-vc37	SAT	UNSAT	3.789	4.987
smallfoot-vc40	SAT	UNSAT	0.943	1.233
smallfoot-vc65	UNSAT	UNSAT	1.647	4.03
smallfoot-vc70	SAT	UNSAT	1.551	2.439

Table 7.8: `qf_shls_entl` Division Benchmarks

tions for each of the Bolognesa examples within 500ms in the worst cases, with the majority of the smaller examples being analysed within 100ms. Factoring in validation, the generalised system was able to resolve each of the entailments under this group of benchmarks, successfully identifying all of the valid entailments as such and identifying at least one set of inferred frames or anti-frames for each of the invalid entailments, typically in under 1s of CPU time.

One of the key observations from this dataset is that the generalised system was able to solve all of the bolognesa examples without timing out, unlike the the prior system which reached timeout over 5 examples from the subgroup. Additionally, execution times overall improved quite significantly, with only 3 benchmarks taking over 1s to complete. This improvement in efficiency is expected to arise from two potential sources: overall improvements to the implementation as a whole and the generalised system itself. While the overall quality of System 2 is higher than that of System 1, it is not anticipated that those enhancements would be sufficient to cause the improvements seen here. While such refinements would accelerate the overall execution of the tool, the difference in execution times between the two systems is not uniform, with the generalised system solving some examples in a slightly longer timeframe than the list-and-tree system and others in an equal or shorter time. This non-direct relation seems to indicate that the underlying generalised system is able to process some of the entailments more effectively than the initial list-and-tree system, presumably allowing System 2 to solve examples that System 1 was unable to resolve in the 30s time limit. In particular, the replacement of the EXCLUDE-MIDDLE rule of the list-and-tree system may have had a notable effect, reducing the number of branches being produced in the search and reducing the overall complexity of constructing the proof.

Clones

System 2 appears to handle the `clones` benchmarks in an effective manner, identifying and verifying solutions for almost all of the entailments in the subcategory with CPU times well under 100ms. While a number of these examples are near-trivial, consisting of highly basic entailments using empty or identical antecedents

and consequents, examples `e07`–`e10` from each level of complexity detail more meaningful entailments and are the focus of the discussion.

The results clearly indicate the generalised system is more efficient over the majority of cases when compared with the results gathered from System 1. While the trivial examples from each complexity level completed in comparable times, the most significant examples show an increasing difference in overall execution time, with the generalised system consistently showing faster overall execution times, approaching execution times a full order of magnitude lower than the list-and-tree implementation.

Another notable advance of the generalised implementation over System 1 is that the generalised tool is capable of identifying solutions to the `e08` benchmarks from many of the complexity levels. These entailments were a set of benchmarks that the list-and-tree system had previously identified as invalid, being unable to identify a valid solution. The `e08` examples, from all complexity levels, describe entailments based around multiple overlapping and interlinking list segments that proved extremely challenging for the both System 1 and System 2, though in differing manners. The list-and-tree system, though able to construct a proof and extract candidate solutions for most of the differing complexities of the `e08` benchmarks, was unable to identify a valid frame and anti-frame set for the entailments. As the system did construct a proof and terminate (in most cases), it is believed that this failure was due to a limitation of the underlying system itself. The generalised system, on the other hand, was able to identify such solutions, identifying a singular frame and anti-frame that would satisfy the bi-abductive problem, though these examples still provided a significant challenge to the generalised system.

While the lower-complexity variations of the `e08` examples are solvable in relatively typical times when compared to other examples at the same level in the `clones` subgroup and others, it can be seen that the performance of System 2 degraded rapidly as the complexity increased. Initial analysis suggests that the core of the issue is the number of candidate solutions produced during the proof process. While the `clones-01-e08` example only produces 3 candidate solutions,

each increase in complexity level appears to increase the number of identified candidates by a factor of 3. Though the tool is able to handle this increase in potential solutions up to a point, by `clones-06-e08`, the tool is producing 729 candidate solutions and is approaching the timeout limit of 30s. As the subsequent examples are anticipated to continue the geometric growth in candidate solutions, it is anticipated that the implementation would be able to identify a valid solution for those entailments, though not within a reasonable timeframe. Attempting to apply the system to `clones-07-e08` without a timeout limit reinforces this theory, as the tool was able to identify a single valid solution from over 2000 candidates, though the overall execution far exceeded a reasonable timeframe.

Smallfoot

The examples supplied by Smallfoot [Berdine et al., 2006] describe relatively simplistic scenarios with typically small numbers of predicates and points-to terms and was an area where the previous system showed a great deal of success, typically solving the examples in only a few milliseconds of CPU time. For the generalised system, this success continues, with System 2 able to identify valid solutions for each of the benchmarks in only a few milliseconds on average, with only a small number being completed in less than 1ms of CPU time. A key highlight is once again `vc-37`, as unlike in the list-and-tree system, the generalised system was able to identify an alias for the pointer variable through an application of the PRED-MATCH rule. The resulting anti-frame and frame consists solely of the alias, making the final specification the optimal solution for that particular bi-abductive problem. However, it is important to note that this resolution was essentially a fortunate side effect of the `PRED-MATCH` rule; the alias was identified through the "equivalence" branch of the rule

This form of entailment continues to be an area where the systems developed in this project excel: relatively compact, interlinking predicates and points-to nodes. Whether this success would extend into full program verification remains to be seen.

LS

The final subgroup of the `qf_shls_entl` division is the LS subgroup. Another smaller subgroup in the division, the LS examples describe relatively basic interactions between lists and points-to terms. As with the Smallfoot examples, the generalised system is able to resolve these bi-abductive problems in a very short length of time - typically between 3ms and 10ms, though `ls-vc03` consumed 53ms of CPU time - and successfully inferred frames and anti-frames where needed.

7.3.2 QF_SHID_ENTL

The `qf_shid_entl` division is the first set of examples to feature shape predicates outside of the restricted fragment utilised by the list-and-tree system and is the largest division considered in this evaluation. The benchmarks of this division consist of entailments that may feature shape predicates of greater complexity than simple singly-linked lists, though remain restricted to basic (dis)equalities and spatial constraints. These examples cover a range of shape predicates commonly explored in verification and entailment proving works, including variations of singly linked lists such as odd or even linked lists, doubly-linked lists, and skip lists, and are supplied to the implementation at the start of each of the benchmarks. In this manner, these examples are essentially making use of user-defined predicates, a powerful capability targeted by a number of tools in the literature [Qin et al., 2017]. Unfortunately, this division also features a large number of entailments that exploit behaviours that are not supported by the current generalised system.

Many the examples in the `qf_shid_entl` division consist of entailments that focus on the interaction between two predicates of differing types but comparable structures. Such pairs are not entirely unexpected during program verification and were lightly explored in the list-and-tree system through the use of the Generalise rules. However, as the Generalise rules do not have an equivalent in System 2, the restrictions placed around unfolding prevent the generalised bi-

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
dll-vc08	SAT	UNSAT	2.53	4.075
dll-vc09	UNSAT	UNSAT	20.858	48.542
dll-vc10	SAT	UNSAT	1.799	3.401
dll-vc11	UNSAT	UNSAT	2.35	6.188
dll-vc12	SAT	UNSAT	4.355	12.718
dll-vc14	SAT	UNSAT	2.718	6.783
dll-vc15	UNSAT	UNSAT	2.861	6.721
dll-vc16	SAT	UNSAT	15.666	39.478
nll-vc06	SAT	UNSAT	12.499	30.434
nll-vc07	UNSAT	UNSAT	7.261	14.268
nll-vc08	UNSAT	UNSAT	13.494	32.637
nll-vc09	UNSAT	UNSAT	8.912	18.689
nll-vc11	UNSAT	UNSAT	13.066	31.117
nll-vc13	SAT	SAT	9.873	10.638
nll-vc14	SAT	SAT	10.557	11.295
nll-vc15	SAT	UNSAT	7.723	10.272
nll-vc17	SAT	UNSAT	30.433	63.313
nll-vc18	SAT	UNSAT	33.889	74.59
nll-vc19	SAT	UNSAT	57.528	106.347
skl2-vc05	SAT	UNSAT	18.09	56.195
skl2-vc06	SAT	UNSAT	21.244	70.803
skl2-vc07	SAT	UNSAT	31.854	64.78
skl3-vc01	UNSAT	UNSAT	6.636	14.342
skl3-vc02	SAT	SAT	5.37	6.07
skl3-vc07	UNSAT	UNSAT	59.885	91.34
skl3-vc09	UNSAT	UNSAT	47.578	80.306
skl3-vc12	SAT	SAT	444.392	598.425
sll-vc01	SAT	UNSAT	532.765	2423.524
sll-vc02	SAT	UNSAT	630.144	2842.882

Table 7.9: `qf_shid_entl` Division Benchmarks

abductive system from inspecting the predicates in depth, preventing the system from solving the entailment as intended. System 2 does attempt to resolve the entailment through the use of inference, but the fragments of the anti-frame and frame produced by these operations are highly likely to introduce violations of the separation constraint of separation logic. As the tool typically infers one of the predicates as missing or extra, the frame or anti-frame essentially produces a state where both predicates occupy the same space, an invalid configuration. While potential solutions to this limitation do exist - a less restrictive approach to unfolding operations, the use of lemmas [Enea et al., 2015, Ta et al., 2017] or cyclic proofs [Brotherston et al., 2011, Brotherston and Gorogiannis, 2014] present three such possibilities - they are not present in the current version of the system and must be considered potential avenues for future work. Additionally, a number of examples make use of shape predicates that are not considered well-formed by the system, typically using recursive definitions, as in even and odd lists, or unfolding from a "non-root" variable, as in reverse lists. As the definition of a well-formed predicate was selected to assist in the design of the generalised proof rules, there is no intention of adapting the system to these predicates at this time.

Because of the large range of differing sources and predicates utilised in this division, as well as the dispersed nature of the supported examples throughout the division, the data gathered will be presented in a more general setting. For the majority of the valid examples present in this division, the generalised system is able to complete the analysis in total CPU times of around 10ms, both in confirming the validity of entailments and the inference of frames and anti-frames to augment entailments. While a range of predicates are represented in the supported set, there does not appear to be a significant impact on the execution time between doubly-linked lists and singly-linked lists, though the analysis times of the examples making use of nested linked lists are notably longer overall, with the values clustered around 50ms for a complete analysis. Examples based upon skip lists, both 2-level and 3-level, also typically complete in longer overall CPU times. While many of the examples complete in CPU times between 30ms and 85ms, a small number of skip list examples take a notably longer time: `skl3-vc11` and

`skl3-vc12` complete in 237ms and 551ms respectively.

From this data, it is suspected that the types of predicate in the entailment have little direct impact on the overall execution time of the system. Instead, it is suspected that it is the *number* of predicates present that has the most significant effect on the overall performance of the tool: both skip lists and nested lists will, by their construction, produce a larger number of predicates from basic matches and unfolding operations. While inference operations will also have an impact, it is viewed as a secondary factor, as the number of possible solutions is also a heavy factor of the number of branches produced by the unfolding and predicate matching rules.

While the performance of the tool over the valid examples continues to show promise, this division does highlight the impact of the limitations and restrictions currently present within the generalised proof systems. Though the system was able to handle examples featuring a range of shape predicates, a large portion of the division could not be supported by the current version of this tool, either through the definition of a well-formed predicate or through examples based around divergent predicates. While the predicate definition is a core aspect of the system and not so easily adapted, the lack of support for matching predicates against predicates of similar structure is a notable limitation. As many of the predicates in later divisions are variations of existing structures with additional constraints over the contents, such as singly-linked lists and sorted singly-linked lists, the ability to identify the structural equivalence of shape predicates is anticipated to be of significant importance during program verification operations. As a result, the inability of System 2 to process such entailments is a significant issue.

7.3.3 QF_SHLID_ENTL

The `qf_shlid_entl` division consists of example entailments that utilise a subset of shape predicates that are *linear* in their construction. Similar to the `qf_shid_entl` division, the entailments in the `qf_shlid_entl` division feature

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
dll-vc03	UNSAT	UNSAT	7.219	17.396
dll-vc06	SAT	UNSAT	5.843	13.974
dll-vc09	UNSAT	UNSAT	14.26	34.046
dll-vc12	SAT	UNSAT	5.045	13.845
dll-vc15	UNSAT	UNSAT	3.421	8
lss-vc01	UNSAT	UNSAT	1.942	4.965
nll-vc03	UNSAT	UNSAT	28.454	72.397
nll-vc06	SAT	UNSAT	14.031	30.024
nll-vc09	UNSAT	UNSAT	10.481	19.66
nll-vc12	UNSAT	UNSAT	6.335	14.406
nll-vc15	SAT	UNSAT	7.388	9.89
nll-vc18	SAT	UNSAT	31.355	72.477
skl2-vc02	UNSAT	UNSAT	10.814	23.841
skl2-vc05	SAT	UNSAT	20.013	56.751
skl3-vc01	UNSAT	UNSAT	6.547	13.86
skl3-vc04	UNSAT	UNSAT	73.051	114.995
skl3-vc07	UNSAT	UNSAT	54.708	88.192
skl3-vc10	UNSAT	UNSAT	19.252	40.812
skl3-vc11	SAT	UNSAT	169.881	276.573
skl3-vc12	SAT	SAT	449.7	604.542
sll-vc01	SAT	UNSAT	535.332	2423.969
sll-vc02	SAT	UNSAT	662.648	2885.812

Table 7.10: `qf_shlid_entl` Division Benchmarks

predicates consisting of (dis)equalities and spatial constraints only. These predicates include doubly-linked lists, nested linked lists, 2 and 3-level skip lists and variations of simple singly-linked lists, sourced from a variety of works in the literature.

DLL

The DLL examples of the `qf_shlid_entl` division describe shape-only doubly-linked lists, consisting of a mix of valid and invalid entailments Over these entailments, the generalised bi-abductive system was able to successfully confirm the validity of or identify some frame and anti-frame for each example. Each of the examples here were successfully identified as - or corrected to - a valid entailment, with the tool completing its analysis in times around 10ms, with a small number of more complex examples taking only 30ms to resolve. As with the previous examples, the validity of the starting entailment does not seem to have a direct impact on execution time, as the while the largest execution time was 32ms over `dll_vc16`, an invalid entailment, the second largest time was 26ms over `dll_vc09`.

LSS

Three simple singly-linked list examples, where the nodes of the list contain two separate pointers to the next node in the list. As with other list-based examples, the generalised system is able to complete its analysis correctly in a minimal amount of time, requiring only 3ms to 15ms, depending on the benchmark.

NLL

The NLL examples are some of the few examples of nested data structures represented in the SL-COMP benchmark set, describing a relatively simple list-of-lists structure. These structures are represented as a pair of shape predicates, a top level singly-linked list with the nodes containing an additional "down" field for the nested list, which is itself represented as a simple singly-linked list predicate. The boundaries of the nested list is defined within the top-level predicate, and for the majority of the examples, is set as `null`.

Over these examples, the generalised system continues to operate effectively. Over each of the examples (save one, as will be discussed below), the tool was able to identify the entailment as valid or infer some frame and anti-frame in

times around 50ms,

One of the examples, `nll_vc13`, represents one of the few entailments where our system is unable to infer a correcting frame and anti-frame. In this example, one of the points-to nodes in the antecedent erroneously uses itself as the boundary value, which must be `null` for the consequent to hold. Such a fault is beyond the capabilities of the bi-abductive system, and is likely not an error that can recovered from in general.

SKL

The skip lists in this division have two possible forms, a 2-level skip list and a 3-level skip list, based upon a pair (or triple respectively) of shape predicates, one predicate representing each level of the structure. These entailments, much like the others in this division, describe a series of overlapping nodes and segments, typically equivalent or near-equivalent to another configuration.

For each of the two-level skip examples, the general system was able to verify the validity of of the entailment or infer some appropriate frame and anti-frame in under 100ms. For three-level skip lists, the success of the tool largely continues, with all but one skip list example resolved in less than 120ms of CPU time and many being resolved faster than that.

While the tool was able to successfully handle each of the skip list examples, `skl3_vc12` ultimately exceeded the 30s timeout limit enforced within the tool. As the timeout occurred in the validation stage, and the validation independently succeeded within 30s, the tool did not return an UNK result. However, it is still apparent that much of the execution time was consumed by the validation of the candidate solutions; only 1% of the overall execution time was spent on the construction of the proof and gathering the candidate solutions.

SLL

The two examples making up the SLL subgroup of the division describe heavily overlapping simple singly-linked lists and points-to predicates. These two examples are both invalid in their initial state, but the generalised tool is able to identify a small number of solutions to "correct" them. In the case of `sll_vc01`, the tool identifies a number of possible aliases that would put the antecedent and consequent into a configuration where the entailment would hold.

These examples required a much larger amount of execution time than other benchmarks in this set, competing the analysis of `sll_vc01` in 2.29s and `sll_vc02` in 2.96s, respectively. As with some of the other list-based examples, it is anticipated that it is the presence of large amounts of overlapping and interlinking nodes and predicates that is the source of the larger time requirement.

7.3.4 QF_SHIDLIA_ENTL

The `qf_shidlia_entl` division is the most significant of those tested during this phase of experimentation, as this division features entailments containing shape predicates making use of both spatial and pure constraints, including arithmetic and ordering constraints, the primary target for the systems described in Chapters 4 and 5. Though there are a small number of restrictions applied to the arithmetic properties, namely that any arithmetic constraints are constrained to linear integer arithmetic, these examples do provide a varied range of examples to examine the capabilities of our system when applied to properties of this form. The predicates tested in this division may include ordering constraints and basic size constraints in addition to the more standard equalities and disequalities. Bag constraints were unfortunately not present in these divisions, leaving that particular aspect of the fragment unrepresented in results.

Benchmark	Expected	Actual	Proof Time (ms)	Total time (ms)
ls-entl-04	SAT	UNSAT	6.245	15.992
ls-entl-05	UNSAT	UNSAT	3.961	8.37
ls-entl-06	SAT	SAT	0.186	0.186
ls-entl-07	UNSAT	UNSAT	4.421	9.569
sls_join_2_known_bnd	UNSAT	SAT	1.511	1.511
sls_join_2_no_cond	UNSAT	UNSAT	10.316	26.484
sls_join_2	UNSAT	UNSAT	9.878	26.275
sls_join_2_unk_lower_bnd	UNSAT	UNSAT	8.698	25.448
sls_join_2_unk_upper_bnd	UNSAT	UNSAT	10.814	30.385
sls_join_3_2_cond	UNSAT	UNSAT	33.408	89.737
sls_join_3_no_cond	UNSAT	UNSAT	19.508	63.713
sls_join_3	UNSAT	UNSAT	27.929	84.539
sls_join_3_unk_lower_bnd	UNSAT	UNSAT	26.831	82.576
sls_join_3_unk_upper_bnd	UNSAT	UNSAT	41.816	115.471
sls_join_4_1_cond_unk_both	UNSAT	UNSAT	96.041	224.691
sls_join_4_1_cond_unk_lbnd	UNSAT	UNSAT	29.471	95.488
sls_join_4_1_cond_unk_ubnd	UNSAT	UNSAT	30.229	89.765
sls_join_4_2_cond_unk_lbnd	UNSAT	UNSAT	74.947	195.511
sls_join_4_2_cond_unk_ubnd	UNSAT	UNSAT	94.925	242.29
sls_join_4_3_cond	UNSAT	UNSAT	81.377	195.564
sls_join_4_no_cond	UNSAT	UNSAT	26.381	82.47

Table 7.11: `qf_shidlia_entl` Division Benchmarks

DLL 1

There are two groups of examples based around the use of doubly-linked list predicates present in the division. The first group is sourced from the COMPSPEN system [Gu et al., 2016b] and describes doubly-linked lists with a tracked length component in the predicate. Generally, the implementation continues to perform well over these examples, completing the analysis and verifying the results

in CPU times around 100ms. This subdivision is notable, however, as it highlighted significant limitations in the generalised system which cause the system to fail to correctly process two of the examples in this category, `dll-entl-09` and `dll-entl-10`.

The two limitations are rooted in the BASE and R-EX rules, and produce a scenario where a should-be empty predicate cannot be reduced and is identified as a (eventually conflicting) component of the frame or anti-frame. The issue of the BASE rule lies in the fact that, by design, the rule will not reduce a shape predicate if there is a possibility that a non-empty state may be extracted within the current environment. Due to the design of the *ldll* predicate used in these examples, a doubly-linked list where all conditions of the base case are met does not invalidate the non-empty case, preventing the BASE rule from being applied. While this is not ideal, this does not become a serious issue until another limitation is encountered; a restriction applied to the `R-EX` rule. In order to ensure the soundness of renaming operations and to reduce the likelihood of incorrect substitutions, the R-EX rule is unable to instantiate existential variables as another existential variable. While this rarely becomes an issue, like the limitation identified in the base rule, it is possible for a state to arise where such behaviour is necessary. In the course of `dll-entl-09`, the system identifies a single candidate solution, in which the anti-frame contains the tail of the list structure, with the first list parameter represented as an existential variable.

$$\frac{\Pi \wedge \Sigma \vdash \exists i.\Pi' \wedge \Sigma' * ldll(E2, E2\_P, i, E2, E2\_P, x2)}{\Pi \wedge \Sigma * E2\_P \mapsto E2, E1\_P \vdash \Pi' \wedge \Sigma' * ldll(E2\_P, E1\_P, x1, E2, E2\_P, x2)}$$
$$\ldots$$

During the verification, the original shape predicate in the consequent is reduced in the same manner, but with a different existential variable representing the length. As a result of this, a configuration is produced where the predicate in the consequent and the predicate in the antecedent are equivalent, but are not identified as such due to the aliased length variables. As the BASE rule cannot reduce the predicates and R-EX cannot produce the instantiation required to unify the predicates, the verification is unable to eliminate the predicates and discards the solution as invalid. Resolving either of these limitations would permit the tool to successfully analyse the examples in question, though care will need to be

taken in order to prevent either unsound operations or non-terminating sequences from being introduced by these refinements.

DLL 2

The second group of doubly-linked list examples in the division are sourced from the Songbird prover [Ta et al., 2016, Ta et al., 2017] and make use of doubly-linked lists with additional ordering constraints.

As with the entailments based around structurally comparable predicates of differing types in the `qf_shid_entl` division, our tool is unable to fully explore the shape predicates in detail, making the system unable to resolve these entailments effectively. While the tool does attempt to apply inference to resolve the entailment, the inferred fragments are highly likely to cause inconsistencies in the entailment when introduced, typically resulting in an output result of SAT.

SLS

The *sls* entailments in this division describe interlinking sorted singly-linked list segments with various levels of information about the upper and lower bounds. These examples are also of note due to their presence - in a modified form - in the evaluation of the initial list-and-tree system, where System 1 was able to confirm the validity of each of the entailments in relatively short times. For the most part, this success continues into the generalised system, with System 2 able to confirm the validity of the entailments with minimal inference in less than 200ms for each of the examples.

However, an erroneous result is produced by the generalised system. When applied to the benchmark `sls_join_2_known_bnd`, the tool is unable to identify a single candidate solution, despite the entailment being readily identified as valid. Upon inspecting the proof tree constructed by the tool during the course of its execution, the cause of this unexpected behaviour becomes clear: an edge case produced by the `PRED-MATCH` rule. The `PRED-MATCH` rule, in its current form,

simulates the compete unfolding of a predicate by replacing it with one of its minimal cases. This process emulates unfolding and matching two matching predicates in parallel until one is fully unfolded. However, in the case of *sls* predicates with known boundaries, this process causes that branch to result in an erroneous state. By reducing the *sls* to its base case, the head and tail, as well as the min and max values, are identified as equal. When using variables for these numerical bounds, there is no issue, as the variables are temporarily equated for the rest of the branch. However, when using two numerical variables, this equality can cause the tool to identify the branch as erroneous. In the case of `sls_join_2_known_bnd`, the "left-minimum" branch produces an expression $100 = 200$, which the tool correctly identifies as `false`, ending the search over that branch. Potential corrections to this issue have been identified, though such fixes have not been implemented at this time. Nevertheless, this scenario does serve to highlight some of the difficulties introduced by handling generalised predicates.

LS

In addition to the doubly-linked list examples, there are also a small number of examples based around singly-linked list structures. Generally, the tool performed in a similar capacity as the performance over the more basic list examples, completing each of the analyses within 10ms.

One example, `ls-entl-03`, does cause the system to fail, returning an "invalid" result over a valid entailment. The root cause of this appears to be the same limitation that caused issues over the known-bounds example of the SLS sub-group, where the PRED-MATCH rule causes a contradiction when unfolding, preventing the valid configuration from being explored.

TreeSeg

The `qf_shidlia_entl` division also contains a single entailment making use of the tree segment data structure. This shape predicate describes a binary tree structure with a defined path from the root to a specified leaf node alongside the expected subtrees. This example is solve with the typical speed and effectiveness, confirming the validity of the tree within 4ms.

7.3.5 System 2 Analysis

The overall performance of System 2 over the examples in the SL-COMP divisions examined, as well as the supporting examples, indicates a reasonable degree of promise.

For the list-only division, the generalised implementation was able to confirm the validity of all of the valid benchmarks and was further able to identify valid solutions to all but 4 of the invalid examples within the 30s timeout limit. Even in the case of the 4 examples which could not be solved within 30s, it is suspected that a valid solution would be found, though not within a reasonable timeframe. When compared with the results gathered during the evaluation of System 1, there does appear to be a small increase in execution time over some examples, though such increases are marginal, often in the range of a few milliseconds. However, the performance of the generalised system over examples of greater complexity indicate that despite some potential inefficiencies over small entailments, System 2 is more efficient overall, able to identify solutions over examples which introduced timeouts in System 1. A further observation appears to indicate that the generalised system is also more effective than the initial list-and-tree system, able to identify solutions to a small number of bi-abductive problems that System 1 was unable to resolve. This improvement in capabilities from the list-and-tree system does contradict the initial expectations towards the generalised system, as it was initially anticipated that the generalised system would be more computationally expensive than the targeted rules of the list-and-tree

system. That System 2 proved to be a more effective tool is an unexpected result, but a positive one nonetheless.

When exploring the capabilities of the generalised system over generalised shape predicates (without arithmetic constraints), the tool continues to show promising results. The generalised tool remains able to complete the analysis and verify the results in relatively short time frames, with the majority of the examples in the two divisions being resolved in times around 100ms. Overall analysis times do appear to be slightly higher when compared to the examples sourced from the `qf_shls_entl` division, though the difference is minimal, again suggesting that the generalised system is a more efficient design despite its greater expressiveness. Whether this is entirely due to the underlying proof systems or the implementation itself is unclear at this time. The generalised tool did encounter a number of issues during the analysis over the `qf_shid_entl` and `qf_shlid_entl` divisions, however. In addition to the anticipated issue of unsupported predicates due to the well-formedness properties expected by the system, there were a small number of cases where the implementation was unable to effectively handle the entailments, resulting in scenarios where tool incorrectly identified a valid entailment as invalid. At present, the root cause of this issue is unknown.

Finally, for the examples in the `qf_shidlia_entl` division, the tool continues to resolve the majority of the examples in relatively short timeframes, with the inclusion of additional pure constraints not appearing to have a significant impact overall. While the limitations identified in the experimentation over the `qf_shid_entl` and `qf_shlid_entl` divisions are still present, they do not appear to have been further amplified.

7.4 Summary

The experimental results gathered over the course of this project presents an interesting picture regarding our approach towards combined domain bi-abduction, as well as providing some insight into its anticipated usefulness.

The implementations, despite their nature as a prototype automated tools, were able to successfully handle the majority of the examples they were applied to, either establishing the entailments validity without the inference of new state - though certain constraints may be identified explicitly instead of implicitly in some cases - or identifying some valid solution for the vast majority of invalid examples. The tools were also able to fully complete these analyses within the 30s timeout limit in the overwhelming majority of cases, with the validation stage typically making up the bulk of the overall execution time. Should the current validation process be made more efficient through the implementation of alternate methods such as the EXP-PURE mechanism [Chin et al., 2012] and integrated SMT solvers such as Z3 [de Moura and Bjørner, 2008], the performance of the implementations are likely to increase even further. The additional complexity introduced through the conversion of separation logic formulae would be a factor in this modification, however, though a small number of approaches have already been presented in the literature [Piskac et al., 2013, Piskac et al., 2014a, Madhusudan et al., 2012, Qiu et al., 2013].

The comparison between the targeted list-and-tree system and the generalised system also raises some interesting points. While the list-and-tree system does operate more effectively over a number of the list-based examples, this difference in overall execution time is relatively minor. In many cases, the generalised system actually showed superior results when compared with the list-and-tree system, and when combined with the increased expressiveness, which in some cases allowed the generalised system to identify solutions the list-and-tree system could not, it can be confidently stated that the generalised system is the superior approach of the two, though some part of that increased success is likely to be due to the overall improvements to the tool when compared to the original.

However, the experimentation done over the bi-abductive systems highlighted a number of key issues and limitations that significantly hampered their capabilities.

For System 1, the most significant issue was the cost of validating the candidate solutions identified from the constructed proof tree. While a critical aspect

of the system, the validation process often took the majority of the overall execution time, often being the primary cause of a time-out result. As this aspect of the analysis was directly impacted by the frequent production of branches in the proof tree through the application of the EXCLUDE-MIDDLE rule, this was an aspect that could not be fully addressed. This aspect was partially alleviated in the later generalised system, though the validation process remained a significant proportion of the overall execution time.

System 2 also had a number of key limitations identified by the experimental results. Several of the benchmarks included in the divisions used during the evaluation included shape predicates that did not meet the conditions needed to be considered well-formed. As was discussed alongside the language fragment 2.5.2, this definition of well-formed was selected in order to simplify the unfolding process and to ensure progression when applying the rules of the generalised system. However, it became apparent during testing that this definition prevented a number of relatively simple shape predicates from being supported by the system, with examples including even-odd lists and reverse linked lists. Additionally, the experimentation revealed a number of weaknesses in the design of several of the underlying proof rules of the generalised system which had a significant effect on the results. In particular, several of the proof rules featured restrictions and assumptions that simplified the application of those rules, such as preventing the instantiation of an existential variable as another during applications of R-EX or the simplified handling of parallel unfolding in the PRED-MATCH rule, which directly caused issues when handling specific examples. These limitations are significant enough that large numbers of examples could not be effectively supported by the system, hampering its capabilities dramatically. Additionally, the omission of a mechanism to explore matches between predicates of differing types further hampered the overall capabilities of the tool, preventing a number of valid entailments from being effectively handled. These issues highlight the complexities of generalised proof systems, as well as the relative immaturity of the systems developed during this project. While these limitations may be addressed in future works, the weaknesses of the systems are still present in this evaluation.

Finally, for both tools, questions regarding the overall scalability of the tools were raised by the results. While performance generally was reasonably effective overall, there were a few notable examples which appear to indicate a potential scalability issue over the number of shape predicates in the entailment. While the performance of System 1 was quite variable over the examples encountered, there was a clear degradation of overall execution time as the entailments became more complex. As an example, comparable `clones` entailments show a clear increase in execution times, with example `e07` from the `clones-05` sub-group being resolved in $112ms$, quickly increasing to $322ms$ for sub-group `clones-06`, $978ms$ for `clones-07` and $3324ms$ for `clones-08`, a significant increase for relatively minor increases in complexity. As has been mentioned, initial examinations suggest the EXCLUDE-MIDDLE to be the rule with the most significant impact, with the increase in entailments triggering a corresponding increase in the complexity of both the proof and subsequent validation of results.

While System 2 performs better overall, the pattern of larger numbers of predicates increasing execution times continues, though in this case, the root cause appears more straightforward; the `e08` benchmarks in the `clones` set of entailments is one of the few fully supported entailments to trigger timeouts in the generalised system and shows a clear and predictable increase in candidate solutions of a factor of 3 as the number of shape predicates present in the example increases. This predictable growth aligns with the number of branches produced by the core case-split rule of the generalised system, UNFOLD-MATCH, again suggesting a strong link between the number of shape predicates, case-split rules and increases in execution times. As this case-split is a critical step in our proof systems, allowing for each valid configuration of a shape predicate to be explored without the need for typical proof search mechanisms such as rollbacks, removing these rules is not a feasible solution. Refinements around the design of the case-split rules themselves, or replacement of these rules with a more efficient alternative may be viable approaches however. Despite this issue, as shown by the successes of both systems over entailments with smaller numbers of shape predicates, these systems may still be suitable for use as part of a real-world analysis strategy. As it is anticipated that entailments with large numbers of

shape predicates will be extremely rare, the systems are most likely to operate on entailments with only a handful of entailments, which are unlikely to result in significant execution times. Such a scenario cannot be guaranteed, however, and a more formal analysis will be needed to establish the potential validity of this theory. As a result, the true scalability of the systems does remain in question.

Overall, the data gathered over the course of this chapter appears to suggest that the underlying bi-abductive systems, as well as the automated implementations, could be a viable approach to resolving bi-abductive inference and entailment problems in the combined shape and pure domain, once a number of critical improvements are made. Though there are a number of limitations and issues surrounding both the implementations and the underlying bi-abductive systems, these flaws appear to be an aspect that can be addressed, either through the integration of alternate approaches to core system such as validation and invalidity checks, or through refinement and addition to the underlying proof rules. As a result of these issues, the prototype systems in their current form cannot not be recommended, as the issues that are likely to arise from their use are likely to undermine the potential benefits they could bring.

Chapter 8

Conclusions and Future Works

To summarise and conclude this work, the potential viability and usefulness of bi-abductive inference techniques designed to operate within the combined shape and pure domain was investigated. A pair of novel entailment provers with bi-abductive inference capabilities - one proof-of-concept system for a restricted language of potentially sorted singly-linked lists and binary trees, and one prototype generalised system for a range of user-defined predicates with arithmetic, sortedness and bag properties - were developed and subsequently implemented into automated versions based on the Cyclist framework. These implementations were then evaluated over benchmarks sourced from four divisions from SL-COMP competition [Sighireanu et al., 2019a] in order to explore the effectiveness of the technique over a range of benchmarks from the literature.

Though the prototype systems developed here had a number of severe and notable limitations, restricting the expressiveness of the system and, in some cases, degrading the overall quality of the output, the results produced from the initial versions indicate a reasonably effective approach. The implemented systems were able to resolve a large portion of the examples within the benchmark sets in sub-second CPU times, either confirming the validity of the entailment, establishing the invalidity of the entailment or inferring some suitable frame and anti-frame to resolve the corresponding bi-abductive problem. In fact, very few supported examples were not resolved within 30s, either through confirmation of

validity (or invalidity) or the inference of appropriate frames and anti-frames.

These results, while not entirely competitive with leading entailment proof systems [Ta et al., 2017, Qin et al., 2017, Le et al., 2018, Sighireanu et al., 2019b], are also not completely out of contention, with overall execution times often only tens to low-hundreds of milliseconds longer than these competitors while also performing inference operations. With further refinements and appropriate extensions, the bi-abductive implementations may be improved to the point of closer competition, both in supported examples and overall execution times, making the systems a more viable alternative.

8.1 Future Works

A number of further avenues of research were identified during the course of this project, ranging from relatively simple refinements to more novel developments and improvements. A selection of the most interesting of these potential future works are presented below.

Current System Enhancements

While showing promise, both the underlying bi-abductive systems and the corresponding implementations have a number of areas where further refinements could be applied. Though not a new work, per se, the refinement and improvement of the systems developed during this project is an area of interest for future efforts. While a number of improvements could be made directly to the existing systems, such as more efficient algorithms and refined functions for the proof rules, there are also a number of additional features that may be integrated into the system to improve it.

Initial efforts towards this work would likely focus on the implementation of the proof rules and search algorithm. As was observed in the experimental analysis, the systems appeared to exhibit scalability issues, particularly over en-

tailments containing a large amount of shape predicates. Refining the system to more effectively handle such predicates would be a good initial approach, either through the refinement of related proof rules, or potentially reworking the core search algorithm, taking advantage of mechanisms such as the "tactic" approach utilised by Cyclist, combining rules to reduce the number of proof rules examined during an iteration of the search, or adding additional mechanisms, such as results caching, to reduce the impact of checking the appropriateness of the proof rules.

As a further example, the integration and use of an SMT solver - alongside mechanisms to convert separation logic formulae to a form that can be discharged by that solver - would provide a number of benefits to the implementations when compared to the current versions, potentially enabling a more effective and efficient validity check and accelerating the completion of the analysis.

Another significant addition would be the addition of a symbolic execution engine for some basic programming language. One of the notable aspects that was absent from the current version of the system and implementation is the capabilities and features required to enable program verification in the implementations, one of the more typical applications of bi-abductive techniques in general [Calcagno et al., 2009, Trinh et al., 2013, Qin et al., 2017]. Adding the capability to examine program code and extract entailments from the point of procedure call would permit the examination of the current bi-abductive systems and implementations in a full verification setting, exploring the capabilities of the system over real-world examples and providing a more complete perspective on the usefulness of the technique.

Lemmas and Automated Lemma Synthesis

Some of the difficulties and limitations encountered by the bi-abductive systems and implementations presented in this thesis arise primarily from the difficulties of handling particular interactions between predicates. While further proof rules could be developed in order to support the reasoning necessary to handle such

scenarios, interactions of this form are varied enough that a rule may not be the best approach. Instead, in order to support the proof rules, the addition of support for lemmas may be introduced. Such work would enable more precise and specialised reasoning for specific scenarios, with the further potential for the automated inference of such lemmas, as in [Ta et al., 2017].

Cyclic Bi-Abduction

Another possible approach to refining the bi-abductive inference systems, and the construction of the proof tree in particular, would be the introduction of cyclic proof mechanisms into the system. As the current implementations are already built within the Cyclist framework - which was initially developed for the cyclic proof of separation logic formulae - the systems are already in a position to take advantage of such mechanisms. However, the interaction between cyclic proofs and bi-abductive inference has yet to be explored, and while there is a possibility of creating a cyclic link in the proof tree without applying inference, such applications are likely to be of limited use. The potential benefits of a bi-abductive cyclic backlink seem intriguing, and thus may be of interest in future investigations.

Higher-Order Predicates

A final avenue for future research is the development or integration of higher-order predicates into the bi-abductive entailment provers. A limitation of the system presented in this document, as well as the majority of bi-abductive systems known in the literature [Calcagno et al., 2009, Qin et al., 2017], is a reliance upon known shape predicates. The use of higher-order predicates, alongside mechanisms to infer those predicates, may allow for the creation of bi-abductive inference systems that will not be reliant upon a set of supplied definitions. The integration of higher-order predicates into a bi-abductive inference system has been achieved in previous works [Le et al., 2017], but remains limited to the shape domain.

As a result, a higher-order bi-abductive inference system operating within a combined shape and pure domain, such as the one utilised by the work presented here, has yet to be developed, and any potential benefits have yet to be explored.

Bibliography

[Abdulla et al., 2013] Abdulla, P. A., Holík, L., Jonsson, B., Lengál, O., Trinh, C. Q., and Vojnar, T. (2013). Verification of heap manipulating prorams with ordered data by extended forest automata. In *ATVA 2013, October 15-18, Hanoi, Vietnam*, pages 224–239. Springer Berlin/Heidelberg.

[Bach et al., 2016] Bach, L. D. X., Le, Q. L., Lo, D., and Goues, C. L. (2016). Enhancing automated program repair with deductive verification. In *2016 IEEE International Conference on Software Maintenance and Evolution*, pages 428–432.

[Balasubramanian et al., 2017] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. (2017). System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 156–161, New York, NY, USA. Association for Computing Machinery.

[Bansal et al., 2009] Bansal, K., Brochenin, R., and Lozes, E. (2009). Beyond shapes: Lists with ordered data. In *International Conference on Foundations of Software Science and Computational Structures*, pages 425–439. Springer.

[Berdine et al., 2007] Berdine, J., Calcagn, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. (2007). Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, pages 178–192. Springer.

[Berdine et al., 2005a] Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005a). A decidable fragment of separation logic. In Lodaya, K. and Mahajan, M., editors,

FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, pages 97–109, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Berdine et al., 2006] Berdine, J., Calcagno, C., and O'Hearn, P. W. (2006). Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W.-P., editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Berdine et al., 2005b] Berdine, J., Calcagno, C., and O'hearn, P. W. (2005b). Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, pages 52–68. Springer.

[Berdine et al., 2011] Berdine, J., Cook, B., and Ishtiaq, S. (2011). Slayer: Memory-safety for sytem-level code. In *International Conference on Computer Aided Verification*, pages 178–183.

[Beyer, 2015] Beyer, D. (2015). Software verification and verifiable witnesses. In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Brotherston, 2005] Brotherston, J. (2005). Cyclic proofs for first-order logic with inductive definitions. In Beckert, B., editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Brotherston et al., 2011] Brotherston, J., Distefano, D., and Peterson, R. L. (2011). Automated cyclic entailment proof in separation logic. In Bjørner, N. and Sofronie-Stokkermans, V., editors, *Automated Deduction – CADE-23*, pages 131–146. Springer Berlin Heidelberg.

[Brotherston and Gorogiannis, 2014] Brotherston, J. and Gorogiannis, N. (2014). Cyclic abduction of inductively defined safety and termination preconditions. In Muller-Olm, M. and Seidl, H., editors, *Static Analysis*, volume 8723 of *SAS 2014*, pages 68–84, Cham. Springer International Publishing.

[Calcagno and Distefano, 2011] Calcagno, C. and Distefano, D. (2011). Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer.

[Calcagno et al., 2009] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. (2009). Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–300. ACM.

[Calcagno et al., 2011] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. (2011). Compositional shape analysis by means of bi-abduction. *Journal of the ACM (JACM)*, 58(6):26.

[Chang and Rival, 2008] Chang, B.-Y. and Rival, X. (2008). Relational inductive shape analysis. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–260. ACM.

[Chang et al., 2007] Chang, B.-Y. E., Rival, X., and Necula, G. C. (2007). Shape analysis with structural invariant checkers. In Nielson, H. R. and Filé, G., editors, *Static Analysis*, pages 384–401, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Chin et al., 2012] Chin, W.-N., David, C., Nguyen, H. H., and Qin, S. (2012). Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. R., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252.

[Cousot et al., 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30.

[Cowan et al., 1998] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. (1998). Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX.

[Curry et al., 2019] Curry, C., Le, Q. L., and Qin, S. (2019). Bi-abductive inference for shape and ordering properties. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 220–225. IEEE.

[Cusumano, 2020] Cusumano, M. A. (2020). Boeing's 737 max: a failure of management, not just technology. *Communications of the ACM*, 64(1):22–25.

[CVE, 2006] CVE (2006). CWE-119: Improper restriction of operations within the bounds of a memory buffer. [Online]. Found at: https://cwe.mitre.org/data/definitions/119.html Last accessed: 21/06/2022.

[CVE, 2020] CVE (2020). 2020 CWE Top 25 most dangerous software weaknesses. [Online]. Found at: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html Last accessed: 19/04/2021.

[CVE, 2021] CVE (2021). 2021 CWE Top 25 most dangerous software weaknesses. [Online]. Found at: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html Last accessed: 21/06/2022.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin Heidelberg. Springer Berlin Heidelberg.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.

[Distefano et al., 2006a] Distefano, D., O'Hearn, P. W., and Yang, H. (2006a). A local shape analysis based on separation logic. In Hermanns, H. and Palsberg, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Distefano et al., 2006b] Distefano, D., O'Hearn, P. W., and Yang, H. (2006b). A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer.

[Distefano and Parkinson J, 2008] Distefano, D. and Parkinson J, M. J. (2008). Jstar: Towards practical verification for java. OOPSLA '08, page 213–226, New York, NY, USA. Association for Computing Machinery.

[Dowson, 1997] Dowson, M. (1997). The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84.

[Dudka et al., 2011] Dudka, K., Peringer, P., and Vojnar, T. (2011). Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *International Conference on Computer Aided Verification*, pages 372–376.

[Enea et al., 2017] Enea, C., Lengál, O., Sighireanu, M., and Vojnar, T. (2017). Compositional entailment checking for a fragment of separation logic. *Formal Methods in System Design*, 51(3):575–607.

[Enea et al., 2015] Enea, C., Sighireanu, M., and Wu, Z. (2015). On automated lemma generation for separation logic with inductive definitions. In Finkbeiner, B., Pu, G., and Zhang, L., editors, *Automated Technology for Verification and Analysis*, pages 80–96, Cham. Springer International Publishing.

[Finkelstein and Dowell, 1996] Finkelstein, A. and Dowell, J. (1996). A comedy of errors: the london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 2–4.

[Floyd, 1993] Floyd, R. W. (1993). *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht.

[Fragoso Santos et al., 2020] Fragoso Santos, J., Maksimović, P., Ayoun, S.-E., and Gardner, P. (2020). Gillian, part i: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA. Association for Computing Machinery.

[Fragoso Santos et al., 2019] Fragoso Santos, J., Maksimović, P., Sampaio, G., and Gardner, P. (2019). JaVerT 2.0: Compositional symbolic execution for javascript. *Proceedings of the ACM on Programming Languages*, 3(POPL).

[Gorogiannis, nd] Gorogiannis, N. (n.d.). The cyclist framework and provers.

[Gu et al., 2016a] Gu, X., Chen, T., and Wu, Z. (2016a). A complete decision procedure for linearly compositional separation logic with data constraints. In Olivetti, N. and Tiwari, A., editors, *Automated Reasoning*, pages 532–549, Cham. Springer International Publishing.

[Gu et al., 2016b] Gu, X., Chen, T., and Wu, Z. (2016b). A complete decision procedure for linearly compositional separation logic with data constraints. In Olivetti, N. and Tiwari, A., editors, *Automated Reasoning*, pages 532–549, Cham. Springer International Publishing.

[Habermehl et al., 2011] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., and Vojnar, T. (2011). Forest automata for verification of heap manipulation. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 424–440, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hackett and Rugina, 2005] Hackett, B. and Rugina, R. (2005). Region-based shape analysis with tracked locations. *SIGPLAN Not.*, 40(1):310–323.

[He et al., 2013] He, G., Qin, S., Chin, W.-N., and Craciun, F. (2013). Automated specification discovery via user-defined predicates. In *International Conference on Formal Engineering Methods*, pages 397–414.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[Holík et al., 2013] Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. (2013). Fully automated shape analysis based on forest automata. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, pages 740–755, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hutton and Meijer, 1996] Hutton, G. and Meijer, E. (1996). Monadic parser combinators.

[Ishtiaq and O'Hearn, 2001] Ishtiaq, S. S. and O'Hearn, P. W. (2001). Bi as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26.

[Jacobs and Piessens, 2008] Jacobs, B. and Piessens, F. (2008). The verifast program verifier. Technical report, Technical Report CW-520, Department of Computer Science, Katholieke ....

[Jacobs et al., 2011] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. (2011). Verifast: A powerful, sound, predictable, fast verifier for c and java. In Bobaru, M., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Le et al., 2016] Le, Q. L., Sun, J., and Chin, W.-N. (2016). Satisfiability modulo heap-based programs. In Chaudhuri, S. and Farzan, A., editors, *Computer Aided Verification*, pages 382–404, Cham. Springer International Publishing.

[Le et al., 2018] Le, Q. L., Sun, J., and Qin, S. (2018). Frame inference for inductive entailment proofs in separation logic. In Beyer, D. and Huisman, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–60, Cham. Springer International Publishing.

[Le et al., 2017] Le, Q. L., Tatsuta, M., Sun, J., and Chin, W.-N. (2017). A decidable fragment in separation logic with inductive predicates and arithmetic.

In Majumdar, R. and Kunčak, V., editors, *Computer Aided Verification*, pages 495–517, Cham. Springer International Publishing.

[Lee et al., 2005] Lee, O., Yang, H., and Yi, K. (2005). Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming*, pages 124–140.

[Leveson and Turner, 1993] Leveson, N. and Turner, C. (1993). An investigation of the therac-25 accidents. *Computer*, 26(7):18–41.

[Madhusudan et al., 2012] Madhusudan, P., Qiu, X., and Stefanescu, A. (2012). Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 123–136, New York, NY, USA. Association for Computing Machinery.

[Magill et al., 2007] Magill, S., Berdine, J., Clarke, E., and Cook, B. (2007). Arithmetic strengthening for shape analysis. In Nielson, H. R. and Filé, G., editors, *Static Analysis*, pages 419–436, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Magill et al., 2008] Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. (2008). THOR: A tool for reasoning about shape and arithmetic. In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, pages 428–432, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Magill et al., 2010] Magill, S., Tsai, M.-H., Lee, P., and Tsay, Y.-K. (2010). Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 211–222. ACM.

[Maksimović et al., 2021] Maksimović, P., Ayoun, S.-É., Santos, J. F., and Gardner, P. (2021). Gillian, Part II: Real-World Verification for JavaScript and C. In Silva, A. and Leino, K. R. M., editors, *Computer Aided Verification*, pages 827–850, Cham. Springer International Publishing.

[Matsakis and Klock, 2014] Matsakis, N. D. and Klock, F. S. (2014). The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA. Association for Computing Machinery.

[McCloskey et al., 2010] McCloskey, B., Reps, T., and Sagiv, M. (2010). Statically inferring complex heap, array, and numeric invariants. In Cousot, R. and Martel, M., editors, *Static Analysis*, pages 71–99, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Microsoft, 2019] Microsoft (2019). A proactive approach to more secure code.

[Muller et al., 2015] Muller, P., Peringer, P., and Vojnar, T. (2015). Predator hunting party (competition contribution). In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–446, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Navarro Pérez and Rybalchenko, 2013] Navarro Pérez, J. A. and Rybalchenko, A. (2013). Separation logic modulo theories. In Shan, C.-c., editor, *Programming Languages and Systems*, pages 90–106, Cham. Springer International Publishing.

[O'Hearn, 2019] O'Hearn, P. W. (2019). Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL).

[O'Hearn et al., 2001] O'Hearn, P. W., Reynolds, J., and Yang, H. (2001). Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19.

[Peirce, 1965] Peirce, C. S. (1965). *Collected Papers of Charles Sanders Peirce*. Harvard University Press.

[Piskac et al., 2013] Piskac, R., Wies, T., and Zufferey, D. (2013). Automating separation logic using smt. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, pages 773–789, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Piskac et al., 2014a] Piskac, R., Wies, T., and Zufferey, D. (2014a). Automating separation logic with trees and data. In Biere, A. and Bloem, R., editors, *Computer Aided Verification*, pages 711–728, Cham. Springer International Publishing.

[Piskac et al., 2014b] Piskac, R., Wies, T., and Zufferey, D. (2014b). Grasshopper. In Ábrahám, E. and Havelund, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Qin et al., 2017] Qin, S., He, G., Chin, W.-N., Craciun, F., He, M., and Ming, Z. (2017). Automated specification inference in a combined domain via user-defined predicates. *Science of Computer Programming*, pages 189–212.

[Qiu et al., 2013] Qiu, X., Garg, P., Ştefănescu, A., and Madhusudan, P. (2013). Natural proofs for structure, data, and separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 231–242, New York, NY, USA. Association for Computing Machinery.

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE.

[Sagiv et al., 1998] Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50.

[Sagiv et al., 2002] Sagiv, M., Reps, T., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298.

[Sighireanu et al., 2019a] Sighireanu, M., Gorogiannis, N., and Iosif, R. (2019a). SL-COMP 2019. https://sl-comp.github.io/. [Online; accessed 01-Jul-2019].

[Sighireanu et al., 2019b] Sighireanu, M., Pérez, J. A. N., Rybalchenko, A., Gorogiannis, N., Iosif, R., Reynolds, A., Serban, C., Katelaan, J., Matheja, C., Noll,

T., Zuleger, F., Chin, W., Le, Q. L., Ta, Q., Le, T., Nguyen, T., Khoo, S., Cyprian, M., Rogalewicz, A., Vojnar, T., Enea, C., Lengál, O., Gao, C., and Wu, Z. (2019b). SL-COMP: competition of solvers for separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, pages 116–132.

[Szekeres et al., 2013] Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62.

[Ta et al., 2016] Ta, Q.-T., Le, T. C., Khoo, S.-C., and Chin, W.-N. (2016). Automated mutual explicit induction proof in separation logic. In Fitzgerald, J., Heitmeyer, C., Gnesi, S., and Philippou, A., editors, *FM 2016: Formal Methods*, pages 659–676, Cham. Springer International Publishing.

[Ta et al., 2017] Ta, Q.-T., Le, T. C., Khoo, S.-C., and Chin, W.-N. (2017). Automated lemma synthesis in symbolic-heap separation logic. volume 2, New York, NY, USA. Association for Computing Machinery.

[Trinh et al., 2013] Trinh, M.-T., Le, Q. L., David, C., and Chin, W.-N. (2013). Bi-abduction with pure properties for specification inference. In *Asian Symposium on Programming Languages and Systems*, pages 107–123.

[Woodruff et al., 2014] Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. (2014). The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468.

[Xu et al., 2017] Xu, Z., Chen, T., and Wu, Z. (2017). Satisfiability of compositional separation logic with tree predicates and data constraints. In de Moura, L., editor, *Automated Deduction – CADE 26*, pages 509–527, Cham. Springer International Publishing.

[Yang et al., 2008] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. (2008). Scalable shape analysis for systems code. In *CAV*, pages 385–398. Springer.

Appendix A

Testing Scripts

In this appendix, the testing scripts utilised in the evaluation stage of this project will be presented and briefly discussed. Two of the scripts used for the development and evaluation of the implementations are included in Figure A.1 and A.2 as illustrative examples.

The benchmarking scripts are hand-written Linux Bash scripts developed to assist in the evaluation of the implementations described in Chapter 6 of this document. The core mechanism of these testing scripts is a loop that iterates through a predefined set of files, passing the path into one of the implementations for analysis and piping the resulting output to a text file. The output file is created - or overwritten - at the start of the script (line 6) and follows a basic naming convention, consisting of the string "`_benchmark_`" followed by the system date. This relatively basic structure allowed for the automatic preservation of records for comparisons between outputs over multiple days, helping to inform and guide the development and evaluation process.

The bi-abductive system utilised during the execution of these scripts was hardcoded into the testing script itself; each script required that the path of the system to be used in the evaluation was specified in the `cmd` variable, alongside any additional execution parameters. In the example presented in Figure A.1,

```
1  #!/bin/bash
2  cmd="./cdbiab_prove_v2.native -s -SLCOMP "
3  output="_benchmark_"
4  output=$output$(date +"%Y-%m-%d").txt
5
6  echo "" > $output
7
8  for file in ./benchmarks/SL-COMP18-master/bench/qf_shls_entl/*.smt2
9  do
10         count=$[$count + 1]
11         echo "Benchmark $count: $file" >> $output
12         $cmd $file >> $output
13         echo "" >> $output
14         echo "Benchmark $count complete..."
15 done
```

Figure A.1: Initial Benchmarking Shell Script

a partially refined version of the initial list-and-tree system was specified[1], making use of the `-s` and `-SLCOMP` options. The `-s` option triggers the systems to output execution statistics necessary for the later evaluation of the results, and the `-SLCOMP` option alongside its corresponding value indicates the location of a file that describes an entailment in the SMTLib format utilised by the SL-COMP competition examples, which formed the bulk of the inputs used during the experimental analysis (Chapter 7).

Additionally, in order to support the user of these scripts, simple updates - a message indicating an analysis has been completed - are output to the shell, allowing for the user to track the progress of the script, albeit with limited information.

[1]This refined version of the list-and-tree system was upgraded with some of the general improvements developed alongside System 2. These updates did not affect the underlying proof system.

```
#!/bin/bash

cmd="./nCdbiab_prove_v2.native -CSV -SLCOMP "
output="_benchmark_csv_"
output=$output$(date +"%Y-%m-%d").txt

echo "" > $output
count=0
for file in ./benchmarks/SL-COMP18-master/bench/qf_shls_entl/*.smt2
do
        count=$[$count + 1]
        $cmd $file >> $output
        echo "Benchmark $count complete..."
done
```

Figure A.2: CSV Variant Shell Script

Each of the scripts developed and deployed during the evaluation of the systems share these core features, with the key differences between the variants being the system targeted for evaluation and whether the `-CSV` option is present in the `cmd` string. An alternative to the `-s` command, the `-CSV` option triggers the system to output all relevant statistics as a comma-separated string of values. When used in conjunction with additional support in the core script, the resulting record is a simple CSV file, ideal for export and further analysis. An example of these CSV evaluation scripts is included in Figure A.2.

While a more refined version of these testing scripts would be possible, such a development was deemed low-priority and was passed over in favour of further refinement of the core system implementations.