MECHANIZED VERIFICATION OF GRAPH-MANIPULATING PROGRAMS

WANG SHENGYI

(M.Sc. in Applied Mathematics, B.Sc. in Mathematics, Peking University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2019

Supervisor:
Dr Aquinas Hobor

Examiners:
Professor Olivier Gerard Henri Marie Danvy
Professor Joxan Jaffar
Assistant Professor Neelakantan Krishnaswami, University of Cambridge

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Wang Shengyi

July 19, 2019

Acknowledgements

This thesis would not have been possible without the help, cooperation, faith and support of many people.

First of all, I would like to thank my supervisor Professor Aquinas Hobor for his guidance and support during my graduate studies at NUS. We had so many insightful discussions. His helpful suggestions and ideas have always been the sources of inspiration to me.

I am grateful to my coauthors on the work presented in this thesis: Qinxiang Cao and Anshuman Mohan. Through these collaborations I have learned a lot, not only theories and techniques but also great passion and a cautious attitude to research. Anshuman Mohan carefully proofread my thesis and provided many excellent suggestions.

I wish to express my gratitude to Professor Andrew W. Appel for his support, ranging from academic research to daily life, during my visit to Princeton University. He showed me the way to overcome some subtle technical odds when I was verifying the garbage collector. His invaluable experience in systems and formal proofs deepened my understanding of VST. I would also like to thank the members of his group: Lennart Beringer, Olivier Savary Bélanger, Qinxiang Cao, Santiago Cuellar and William Mansky, for all the intellectual conversations and casual chats.

I also want to acknowledge Professor Chin Wei Ngan for introducing me to Coq as a way of formalizing mathematical proofs and advising me to do solid research instead of playing publication games.

I would like to thank Professor Zongyan Qiu, who supervised my master's thesis. He introduced me to the fantastic and broad world of formal methods. It is he who told me that I could and should treat computer programs mathematically.

I would like to thank Wen Cao, my high school computer technology teacher and trainer, for his encouragement all these years.

I would like to thank all my labmates in the PLS2 lab for the last six years of friendship and support. We shared many fun hours and precious memories.

My girlfriend Cheng Chen deserves special thanks for her genuine care and warm support through some of the worst times of my life.

Last but not least, I would like thank my parents. They have always been supportive and caring, no matter what happened. I am very fortunate to have had access to computers in my early years, in addition to various books about recreational mathematics. My interest in mathematics and computers was certainly dependent on these things brought into my life by my parents. Mom and Dad, thank you for always standing by my side.

Contents

Summary

This thesis tackles the mechanized verification of realistic programs which manipulate heap-represented graphs. Many practical problems can be finally abstracted as problems about graphs, and so graph-manipulating programs are widely used in many aspects of human society. Although the modern development of formal proof techniques makes the verification of real programs feasible, verification is still rather difficult when dealing with programs involving graphs. This is because graphs often exhibit deep intrinsic sharing, which is not a typical scenario for existing techniques like separation logic.

Since the specifications of graph-manipulating programs are easily expressed using the language of graph theory, we construct a reusable library of formalized graph theory. It is a modular and general library for reasoning about abstract mathematical graphs. Our setup supports various graphs, from well-organized graphs with specific properties such as directed acyclic graphs or a disjoint-forest, to totally unstructured graphs such as objects laid out in the memory of a running program. We distill and formalize several key concepts in graph theory, such as reachability and graph isomorphism. We prove hundreds of theorems about these concepts so as to support further inference.

Next, we use separation logic to define how such abstract graphs are represented concretely in the heap. To facilitate the spatial entailments involving graphs, we propose an inference rule called Localize which generalize the Ramify rule. We show how this rule can support existential quantifiers in postconditions and smoothly handle modified program variables. Furthermore, we summarize several common patterns in premises of the Localize rule. We prove several supporting theorems about the patterns to ease the applications of the rule. The spatial representations, the Localize rule, and the supporting theorems together constitute a spatial graph library.

To illustrate the generality and power of our techniques, we integrate the mathematical and spatial graph libraries into the Verified Software Toolchain and certify the functional correctness of six graph-manipulating programs written in C. They include a graph marking program for binary graphs and acyclic binary graphs, a spanning tree program, three different implementations of the classical union-find algorithm, and a 400-line generational garbage collector for the CertiCoq project. The verification of the garbage collector is a huge project which took eight months' effort and contains more than 700 theorems.

Our proofs are entirely machine-checked in Coq.

List of Figures

List of Tables

Chapter 1

Introduction

With the development of theory and technology, formal verification of software is becoming more and more practical and widely adopted. Now researchers can verify a large class of programs with full formal machine-checked proofs [Appel, 2015; Ye et al., 2017]. But it is still an important and big challenge to formally verify graph-manipulating programs written in imperative programming languages. In this thesis we developed a general and powerful framework which facilitates reasoning about such programs. We show that this framework has reached a certain degree of maturity by using it to verify several classical graph algorithms written in C, and further, a substantial and real-world generational garbage collector. Our framework and all proofs of programs are machine-checked in the Coq proof assistant [Coq Development Team, 2019].

1.1 Quality Control of Software Systems

Human society is more deeply entangled with software systems than ever. Software controls many aspects of our modern world: electrical power, communications, transportation, finance, medicine, and entertainment. In fact, it is hard to find an area in which we do not depend on software.

In this sense, modern society relies on the correct functioning of software. However, people have gotten used to errors in their daily software. Most of the time it is not a big problem if a word processor crashes or a video player has no sound when playing movies. People usually just shrug and try restarting. But sometimes there are extremely serious consequences if some software systems behave improperly.

In the history of software, it is not rare for people to suffer severe loss on account of defective software. Between 1985 and 1987, at least six patients were given massive overdoses of radiation because of concurrent programming errors in Therac-25, a radiation therapy machine [Leveson and Turner, 1993]. In 1996, the malfunction in the control software of Ariane 5 space rocket resulted in a loss more than 370 million dollars [Lions et al., 1996]. In 2003, a widespread power outage throughout parts of the Northeastern and Midwestern United States and the Canadian province of Ontario called "Northeast blackout of 2003" happened due to a software bug known as a race condition in the alarm system at the control room of a company [North American Electric Reliability Council, 2004]. In 2009, Google's search engine erroneously marked every web site worldwide as potentially malicious because of a bug in Google's malware detector [Davies, 2009]. In 2010, Toyota announced recalls of approximately 7.5 million vehicles partially because Toyota's electronic throttle control system could cause sudden unintended acceleration, which had killed at least 89 people in past decades [Dunn, 2013]. In 2012, Knight Capital Group lost 440 million dollars in 45 minutes caused by a glitch about its trading software [Popper, 2012]. In 2016, twenty years later after the Ariane 5 failure, Hitomi—an X-ray astronomy satellite costing about 280 million dollars—broke into pieces due to multiple incidents with the attitude control system [Clark, 2016]. This enumeration of misfortune caused by software bugs is just the tip of the iceberg. Besides these exposed ones, there are almost certainly many other unexposed bugs in existing software systems. Reducing bugs in software, in other words, the quality control of software, is an extremely necessary work in running modern human society smoothly and safely.

Furthermore, the quality of software is closely related to our understanding of nature. Researchers are writing software to model biological structures, analyze data generated by particle colliders, process signals from radio telescopes, and simulate the early evolution of the universe. The increasingly important role played by software in research implies that bugs in the software may threaten the correctness of scientific discoveries. In fact, we already know that this is happening. In 2007, a structural-biology group retracted five previously published papers because of the discovery of a critical bug in the its own custom software tools [Miller, 2006]. At CERN, physicists adopted some software techniques to find more than 40,000 defects in the system which is used in their pioneering Large Hadron Collider (LHC) experiment [Ling, 2011]. In cosmology, when two sky images from the same direction made by two different experiments did not agree at all, one group of scientists checked and realized that it is caused by a bug in their computer program [Tegmark, 2014]. Such known defects not only weaken scientific claims that have been *shown* to be buggy, but also reduce the trust we can have in science in general. Moreover, for some critical topics such as climate change, the weaknesses in software will impede the ability of collective decision-making [Easterbrook, 2010].

Traditionally software engineers, just like engineers in other fields, adopt testing to detect potential defects. Although software testing is widely used and can guarantee the quality of software in some sense, it can never promise that the software is bug-free. As Dijkstra pointed out, “program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence” [Dijkstra, 1972]. A typical recent example is TimSort, the standard sorting algorithm used in Python, Java, and Android platform. Such a fundamentally important algorithm implemented by very experienced developers of the Java standard library with a lot of tests still contains a sophisticated bug [de Gouw et al., 2015]. Sometimes it is expensive or even impossible to test the software because of the uncontrollable outside conditions. Besides the well-known difficulty in testing concurrent programs, modern development of software techniques raise new challenges in testing. For example, testing software designed to run on mobile networks is very hard and expensive because mobile terminals would move around and the software has to change subnetworks, which leads to complicated situations that are hard to reproduce [Satoh, 2004]. Another example is the traffic situation for a self-driving car, which could be rather complicated and difficult to test [Cerf, 2018].

Since software systems have huge responsibilities for human well-being, and testing is inadequate to eliminate bugs in software, researchers explore alternative approaches to control the quality of programs. One of them is model checking, a technique for automatically verifying correctness properties of finite-state systems. In its classical form, model checking consists of three components: modeling, specification and algorithms [Clarke et al., 2018]. For example, let us consider temporal-logic model checking, an important class of model checking methods.

In temporal-logic model checking, the system under investigation is usually modeled as a finite state-transition graph (a.k.a. Kripke structure) K. The specification, which is the description of correctness properties for the systems, such as the absence of deadlocks, is expressed as a temporal-logic formula ϕ. After these are prepared, an algorithm—the model checker—decides whether $K \models \phi$, i.e., whether the structure K is a model of the formula ϕ, by exhaustively checking every possible state of K. If $\neg(K \models \phi)$, then the model checker will give a counterexample violating ϕ. Since K has finitely many different states, the checking is always decidable provided there is a decision procedure for any single state. As compared to conventional testing, where a test simply passes or fails, model checking gives a certificate of the correctness on the declared property ϕ when no counterexample is found. When the model is buggy, it will give a helpful counterexample, which can serve as an aid to the programmer who then needs to fix the code. The major limitation of model checking is the combinatorial blow up of the state-space—the state explosion problem. Thanks to the ingenious algorithms and data structures developed over the past three decades, together with the increasing computer speed and memory size, model checking is now a viable technique. Model checking is widely used for the verification of hardware and software in industry especially because its good scalability.

Besides the prohibitive cost caused by the state explosion problem, model checking has another limitation: expressiveness. There is a mismatch between the model and the complex system under investigation, even though the research spanning the last thirty years has helped to close the gap in many areas. Because of the mismatch, for some programs, model checking cannot tackle the data manipulation precisely. If someone aims at full functional correctness, there is another good option: proving the correctness of the program deductively. Since a program is just a formal description of a calculation, there is no reason that it cannot be verified in the same way that mathematicians have proved theorems using logic for the past 2600 years. Compared to software testing and model checking, theorem proving has the greatest expressive power while it sacrifices scalability: even for experts, developing proofs is a time-consuming and non-trivial effort. Over the years, researchers in computer science have developed various theories and tools to help the theorem proving about programs and mathematics. This is what we want to discuss in the following sections.

1.2 Formal Proof of Mathematics

The history of mathematical proof can be traced back to the era of ancient Greece around the sixth century BC. Thales of Miletus (624–546 BC) is the first known individual to use deductive reasoning in mathematics, by proving what is now known as Thales' theorem. Around 300 BC, Euclid codified geometry in his *Elements* by adopting the axiomatic deductive approach. The logically coherent framework and rigorous proofs presented in this treatise epitomized the axiomatic deductive method. In the following twenty three centuries until now, generation after generation of mathematicians write mathematical proofs in the same style as Euclid did in *Elements*.

Although ancient Greek philosophers like Aristotle and Chrysippus analyzed patterns of reasoning and summarized some theories like syllogism and Stoic logic, Euclid's Elements does not use these theories because the patterns of language addressed by their theories for reasoning are quite limited. In other words, they are not rich enough. Until 1879, it was a German mathematician Gottlob Frege who first proposed a unified and comprehensive system of logic capable of analyzing and representing mathematical proofs completely and adequately [Kneale and Kneale, 1985]. His theory has gradually developed into modern mathematical logic, which has become the foundation of mathematics and any other deductive science relying on rigorous reasoning.

Traditional mathematical proofs are written in informal languages. In such typical proofs, routine logical steps are usually omitted, and a large amount of context is assumed on the part of the readers because the proofs are only supposed to be understood by trained mathematicians. Although mathematics is thought to consist of analytic truths, it is still common to find gaps or errors in published and accepted proofs because sometimes even the intuition of professional mathematicians can be wrong. Furthermore, modern proofs can be thousands of pages long. This fact makes it extremely difficult for mathematicians to develop enough confidence in the proofs. Since the mechanism underlying proofs—the mathematical logic—is well-studied, a natural expectation is that people can employ programs to check mathematical proofs mechanically to gain more assurance of correctness. This is one of the motivations for the sprouting up of formal proofs.

In contrast to traditional mathematical proofs, formal proofs are written in formal languages, which are explicitly defined by a specific set of formation rules. In formal proofs, all the intermediate logical steps should be supplied even though they are routine, basic application of inference rules. Since a formal proof is in such a greatly expanded form without any intuitive arguments, it can be easily checked by a simple program all the way back to the fundamental axioms. Thus the trust of formal proofs is built on the trust of the checking program and the underlying theory of the program. As we mentioned before, a computer program is error-prone. Therefore ideally the checking program should be simple enough to be easily verified by people to ensure that there are obviously no deficiencies. Besides the checking program, the other trust base is usually the mathematical logic itself or some equivalent theory of logic underlying the checking program.

In 1954, Martin Davis implemented Presburger's decision procedure, which proved that the sum of two even numbers is even for a JOHNNIAC vacuum tube computer at the Institute for Advanced Study [Bibel, 2007]. In 1959, Hao Wang's program on an IBM704 mechanically proved several hundred mathematical logic theorems in Whitehead and Russell's Principia Mathematica [Wang, 1960]. In 1968, N.G. de Bruijn designed the first computer system Automath which can verify the correctness of mathematical proofs written in its own defined formal language. Automath was later used in verifying propositions from Edmund Landau's Foundations of Analysis [Kamareddine, 2011]. Influenced by de Bruijn's Automath, many computer proof assistants were developed in the following years. They do not only provide languages to express theories, write proofs and programs to check the validity of proofs, but also provide some procedures called *tactics* to help generate proof terms sometimes because formal proofs could be rather verbose and complicated. Since the validity of a formal proof is guaranteed by the small checking core, those tactics which help forming proofs could be arbitrarily complex and not necessarily be as reliable as the checking core. Over the decades researchers developed various proof assistants, but only recently were those assistants robust and efficient enough to formally prove substantial theorems in practice. In 2004, Georges Gonthier proved the famous four-color theorem in proof assistant Coq [Gonthier, 2008]. It should be noted that this is different from the Appel–Haken proof which also used a program to check the thousands of reducible configurations one-by-one in 1976 [Appel et al., 1977]. If Appel–Haken proof is the computation of $1 + 1 = 2$, Gonthier's formal proof is the justification of $1 + 1 = 2$ from the rigorous construction of the natural numbers and the plus operation. In 2005, the Jordan curve theorem was formally proved by Thomas Callister Hales in another proof assistant HOL Light [Hales, 2007]. The proof contains about 60,000 lines. In 2012, the Feit–Thompson theorem was proved in Coq by a team led by Georges Gonthier after a 6-year long research effort [Gonthier et al., 2013]. This is a big proof which contains 170,000 lines. In 2014, Thomas Hales et al. finished a ten-year collaborative project called Flyspeck—the formal proof of the Kepler conjecture in the HOL Light and Isabelle proof assistants [Hales et al., 2017].

Each of those established proofs is not just a single statement of the declared proposition but a large formal library of mathematical theories. For example, the library of the Feit–Thompson theorem formalized a large portion of the finite group theory. It contains about 15,000 definitions and 4,200 theorems. Researchers can further contribute to the library to prove other important theorems, just like mathematicians did in centuries past. It is hoped that one day the formal system can help to support state-of-the-art research in mathematics, with a sufficiently large formalized library of theorems. In 1993, researchers proposed the QED manifesto, a proposal for a computer-based database of all mathematical knowledge, strictly formalized. Although it failed in 1996 partly because the serious shortcomings of existing proof assistants to express mathematics [Wiedijk, 2007], the spirit lived on. Now there is another on-going project called UniMath that aims to formalize a substantial body of mathematics [Voevodsky et al.].

1.3 Formal Verification of Programs

Abstractly, computer programs are just formal descriptions of calculations. The essence of a Python program which computes the greatest common divisor (GCD) of two natural numbers using the Euclidean algorithm is no different from the procedure described in Euclid's Elements at 300 BC, except that the program is executable in a machine. Thus the mathematical logic is definitely capable of proving the correctness of computer programs, just as it proved the validity of mathematical theorems. The proof about program correctness could be rather complex because it is not only about the abstract algorithms but also involves the concrete semantics of the programming languages used in the program. With the help of modern proof assistants, we can now manage such complexity.

The awareness of the need for proofs of program correctness by mathematics can date back to the beginning of computer science history. In the first introduction of "von Neumann machine", John von Neumann described how to prove the correctness of machine code programs [Goldstine and Von Neumann, 1948]. Alan Turing gave a proof of a program with two nested loops and an indication of a general proof method [Turing, 1949]. Unfortunately this work had little impact on the subject of reasoning about software. In early years of 1960s, John McCarthy pointed out the direction of modern program verification in his two visionary papers [McCarthy, 1962, 1963]: Instead of debugging a program, we need a machine-checked proof saying that a program meets its specification.

The more fruitful development of program proofs started fifty years ago. Peter Naur gave a first workable method for verifying programs written in ALGOL 60 [Naur, 1966]. Later Robert Floyd provided a system in flowcharts which can prove correctness, equivalence, and termination of computer programs [Floyd, 1967]. C.A.R. Hoare formalized Floyd's partial correctness proof method in a logic (called "Hoare logic" or "Floyd-Hoare logic") [Hoare, 1969], which now can be seen as a cornerstone in verifying programs written in imperative programming languages. Edsger Dijkstra introduced predicate transformers which gives complete strategies to build valid deductions of Hoare logic [Dijkstra, 1975]. Around 2000, Peter O'Hearn et al. developed separation logic as an extension of Hoare logic, which support local reasoning about a program component, not the entire global state of the system [O'Hearn et al., 2001; Reynolds, 2002].

Along with the evolution of the proof theories, researchers also developed many tools which can assist program verification. In 1970s, Robert S. Boyer and J Strother Moore started to make a fully automatic, logic-based theorem prover Nqthm, which was used in the verification of many symbolic programs [Boyer et al., 1995]. Its successor is ACL2 [Kaufmann et al., 2000b,a], which can be seen as an "industrial strength" version of Nqthm. Robin Milner developed LCF (Logic for Computable Functions), an interactive automated theorem prover which allows users to write theorem-proving tactics [Milner, 1972]. LCF has two successors, HOL [Gordon, 2000] and Isabelle [Paulson, 1989]. HOL (High Order Logic) is a family of interactive theorem proving systems using higher-order logic and implementation strategies. Isabelle is another interactive theorem prover in the LCF-style which features efficient automatic reasoning tools. In this thesis, we use Coq [Coq Development Team, 2019]—a powerful interactive theorem prover built within the theory of the calculus of inductive constructions—to verify the correctness of programs. The initial release of Coq can be dated back to 1989. After twenty years of continuous development, it is widely used in both program verification and mathematical theorem proving. As mentioned before, Coq has been used in proving very large mathematical theorems like the four-color theorem and the Feit–Thompson theorem. Another large application of Coq is the development of CompCert.

Starting from 2005, a team of researchers led by Xavier Leroy developed CompCert—a formally verified optimizing compiler for the majority of C99 programming language which is specified, programmed and proved in Coq [Leroy et al., 2012]. The executable code generated by this realistic compiler is proved with a mathematical, machine-checked proof to behave exactly as prescribed by the semantics of the source program. In other words, the code compiled from CompCert is exempt from miscompilation. Before the CompCert project, even though we could prove that a C source program is bug-free, the executing machine code could still behave incorrectly because it is difficult to rule out the possibility of compiler-introduced bugs. It is the appearance of CompCert that makes end-to-end verification of C programs much easier without sacrificing performance of the machine code.

With the help of theoretical approaches and applicable software, there are several other impressive breakthroughs in recent years in the code-level formal verification of real-life programs. Compared to some theoretical exploration projects which are usually built on simple/pseudo programming languages and illustrative semantics, the code-level formal proof emphasizes pragmatic benefit of verifying real implementations and reveals unseen complexity of real programming language semantics. NICTA's seL4 (Secure Embedded L4 microkernel) is a general-purpose operating-system kernel whose formal proof of functional correctness was completed via Isabelle in 2009 [Klein et al., 2009]. Encouraged by the success of CompCert, Ramana Kumar et al. developed CakeML, a mechanically verified compiler which supports a substantial subset of Standard ML [Kumar et al., 2014]. In particular, this formally verified compiler can bootstrap itself: the verified compiler is applied to itself to produce a verified machine-code implementation of the compiler. In 2015, researchers in Massachusetts Institute of Technology developed a verified crash-resistant file system FSCQ through Coq [Chen et al., 2015]. In 2016, researchers in Yale University presented the first functional correctness proof of a complete, general-purpose concurrent OS kernel with fine-grained locking called CertiKOS [Gu et al., 2016]. The complete formal verification of CertiKOS leverages CompCert and Coq. There are even verified interactive theorem provers such as Milawa [Davis and Myreen, 2015] and Candle [Kumar et al., 2016]. None of these projects are toy systems. Their verification required significant effort.

1.4 Gaps in Formal Verification of Programs

As we can see, over the years great progress has been made in the formal verification of real programs. However, the effort needed for full functional correctness proof of real programs is considerably high so far. We must decrease the effort for such verification to be practical for widespread use.

The high cost of formal verification of real programs could be due to several factors and researchers developed various strategies to attack them. One of them is the magnitude of the program to be verified. The de Bruijn factor, a ratio between sizes of the formalized proof and the corresponding conventional proof is a standard benchmark to measure the overhead of a formal proof. For mathematical theories, the factor is around 4 [Wiedijk, 2000]. For formal proofs of programs, we can adjust the factor to be the ratio between sizes of the proof and the corresponding program. In the case of MIT's FSCQ file system, the ratio is around 10, which is much higher than 4 [Wang, 2017]. This high ratio is not feasible for large applications. One solution of this ratio problem is splitting the software system into several isolated components. For the critical components that a system's safety or security relies on, we can formally verify them. For the less critical components, we can use some other techniques such as conventional testing. The combination of formal and informal techniques can still dramatically increase the assurance of the system. Even the informal part can reap the benefits from the formal part. For example, the formal proof of a program usually assumes some explicit conditions under which the program would run. Thus when developers are testing the callers of the program, they can focus on confirming those assumptions hold. Another solution to this ratio problem is partially automating the formal proofs. The halting problem is undecidable [Turing, 1937]. In addition, Rice's theorem tells us that all non-trivial, semantic properties of programs are undecidable too [Rice, 1953], which means there exists no automatic method that decides with generality non-trivial questions on the behavior of computer programs. So we cannot expect fully automatic provers. However, a certain portion of proof obligations generated by proof assistants can be transformed into decision problems which are decidable. Thus researchers developed decision procedures—algorithms that terminate with correct yes/no answers—for decidable decision problems [Kroening and Strichman, 2016]. Furthermore, those decidable theories can be combined and expressed in classical first-order logic as logical formulas, which are instances of so-called satisfiability modulo theories (SMT). There are various SMT solvers such as STP [Ganesh and Dill, 2007], CVC4 [Barrett et al., 2011] and Z3 [de Moura and Bjørner, 2008] which are widely used in verification now.

Despite the various strategies for the high ratio problem of formal verification, another problem which prevents the widespread applications of full functional verification of programs—the inherent complication of proofs—is rather intricate to solve. The complication is twofold. One is to describe and reason about the state transition between instructions in a program. For verification at the source code level, it partially depends on the semantics of the concrete programming language, which could be rather complicated or verbose. In recent years, this factor has been alleviated by separation logic. The second complication is to prove the mathematical truth about the program. Here we use *mathematical truth* to refer to the statement of the program behavior which is independent of program-related representations in the specification of a program, since miscellaneous mathematical concepts and theories provide good abstractions and support for various phenomena. For example, to describe the specification of a sorting program, we abstract the concept of number sequence to say "the resulting sequence is a permutation of the original sequence and the numbers in the sequence are in ascending order", regardless of whether the program stores the numbers in an array or in a linked list. In order to prove such mathematical truth, it is natural to define some helper concepts and to prove some lemmas, which is the main activity when developing proofs of full functional correctness. Some mathematical properties like the transitive law of the "less than or equal to" relation of integers in verifying a sorting algorithm is not difficult to prove while some other properties of more complicated mathematical concepts such as the limit of a real number series which is needed to prove the correctness of a numerical algorithm or prime number related lemmas which is needed in verifying a cryptography program may need significant human effort to establish. To ease the burden of such kinds of verification, we can classify programs according to the mathematical concepts involved and then develop frameworks composed by definitions and theorems all around the central concept. The leitmotif of this thesis is one such concept: graphs. As we shall see, for programs which manipulate graphs, the two factors of the verification complication is sometimes entangled, which makes their full functional verification a very challenging problem.

1.5 Goals and Scope of the Research

In the field of discrete mathematics, graphs are a general and powerful abstraction for modeling relationships and processes in physical, biological, social and information systems. Graph-related programs are widely applicable since many practical problems can be represented by graphs. So it is definitely valuable to verify graph-manipulating programs.

On the other hand, verifying graph-manipulating programs is also a difficult problem. Separation logic greatly simplifies the verification of programs on shared mutable data structures such as lists and trees while the inference for data structures with intrinsic sharing (e.g., graphs) is notoriously hard, especially for modular reasoning. Over the years, much effort was devoted to verifying algorithms on graphs. In 2001, Hongseok Yang verified the Schorr-Waite graph marking algorithm using separation logic [Yang, 2001]. In 2004, Lars Birkedal et al. verified the Cheney two-space garbage-collection algorithm by global invariants [Birkedal et al., 2004]. Richard Bornat et al. verified graph disposing and copying algorithms via a more systematic way [Bornat et al., 2004]. In 2008, Carsten Varming and Lars Birkedal [Varming and Birkedal, 2008] gave a machine-checked proof of Cheney's copying garbage collector, which generalized the results in [Birkedal et al., 2004]. In 2013, Aquinas Hobor and Jules Villard proposed a new proof theory called ramification theory which is handy for data structures with intrinsic sharing. They also verified various graph-related algorithms such as graph marking and dag copying using their new theory [Hobor and Villard, 2013]. However, most of them are theoretical explorations. The only mechanically checked verification is based on a simplified language with a standard operational semantics, to say nothing of being end-to-end checked for a real programming language.

Apart from the category of programs under investigation—the graph-manipulating programs, as for the category of verification, we focus on the end-to-end, full functional correctness verification of realistic programs, i.e., C programs. One obvious reason for our interest in end-to-end proof of C programs is that recent research developments have made such proofs feasible. The theoretical support (e.g., separation logic) and tool-chain support (e.g., CompCert and VST [Appel et al., 2014]) are ready. But more importantly, the end-to-end proof provides the highest assurance of correctness and a lot of fundamentally important programs are written in C. It should be noticed that even with the theoretical support and tool-chain support, it is still intricate to mechanically verify the functional correctness of C programs. Some elaborate conditions or assertions to exclude undefined behavior will only be exposed with full formal semantics of the C programming language [Krebbers, 2015].

Another critical issue is to draw inferences from the abstract, mathematical graphs which underly these programs. We need to develop a library of formal graph theory. Although there is a long history, going back at least twenty-five years, of mechanized reasoning about mathematical graphs, to our knowledge, there is no modular and general-purpose framework for graphs with broad applications yet. Some of them are used for very specific purposes [Wong, 1991; Yamamoto et al., 1995; Tamai, 2000; Nordhoff and Lammich, 2012]. Some of them are restricted to graphs with special shapes or properties [Chou, 1994; Butler and Sjogren, 1998; Ridge, 2005; Nipkow et al., 2006]. Some of them only provide support for some basic concepts [Duprat, 2001]. Some of them cannot be integrated into existing frameworks smoothly [Ridge, 2005; Noschinski, 2015a]. We hope our framework has no such limitations. It should be general and powerful enough to support end-to-end proof.

The goal of this research is to ease the burden of the formal verification of programs with intrinsic sharing data structures. One objective of this research is to explore and develop a general and more simple way to verify realistic graph-manipulating programs mechanically. This verification focuses on proving the fully functional correctness of graph algorithms formally and manually. We intend to verify real, runnable programs written in the C programming language. The scope of the research includes separation logic, certified proof and formalization of mathematical structures.

1.6 Our Method and Contribution

To this end, we decide to combine the theoretical research with practical proof engineering. The theoretical research is based on survey and reasoning while the proof engineering is mainly about proving a set of theorems with the Coq proof assistant, so as to build a framework for graph-related reasoning. The whole framework is built on top of CompCert (giving formal semantics of C) and VST (giving a higher-order separation logic) to be cable of verifying C programs from end to end.

The main contributions are as follows. We provide a general and modular framework for graph-related reasoning which can be roughly split into two parts: a formalization of graph theory in mathematics and a spatial library in separation logic. The latter connects the underlying mathematical graph model and the graph representations in heaps. We use the framework to verify several graph-manipulating algorithms, including graph mark, spanning tree, two versions of union-find, and a generational garbage collector with industrial strength.

Chapter 2

Preliminaries

This chapter gives some preliminaries about our work. It is not a complete survey of all aspects about mechanized program verification but only contains closely-related topics. For example, there are many examples of proof-assistant software such as HOL, Isabelle, LEGO, etc. but only Coq is discussed here. Other topics include Hoare logic, separation logic, ramification, dependent type theory and their applications.

2.1 Logic

2.1.1 Hoare logic

Hoare logic is a great attempt to reason about program properties rigorously [Hoare, 1969]. It explores and builds the logical foundations of computer programming by using axioms and inference rules for the first time. The central concept of Hoare logic is the Hoare triple of the form $\{P\}\,C\,\{Q\}$ where P and Q are assertions called the precondition and postcondition respectively and C is a program. It can be interpreted "If P is satisfied, successfully executing C establishes Q". The assertions (P and Q) about properties of programs can be expressed as formulas in mathematical logic in terms of values which the relevant program variables will take. In imperative programming languages, a program C is composed by various basic constructs. Thus Hoare logic provides axiom schemas and inference rules for several typical constructs of imperative languages so as to deduce properties of a whole program composed by those constructs.

$$\textsc{Empty}\quad \frac{}{\{P\}\,\textbf{skip}\,\{P\}} \qquad \textsc{Assignment}\quad \frac{}{\{P[E/x]\}\,x := E\,\{P\}} \qquad \textsc{Composition}\quad \frac{\{P\}\,S\,\{Q\} \quad \{Q\}\,T\,\{R\}}{\{P\}\,S;T\,\{R\}}$$

$$\textsc{Branch}\quad \frac{\{B \wedge P\}\,S\,\{Q\} \quad \{\neg B \wedge P\}\,T\,\{Q\}}{\{P\}\,\textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\,\{Q\}} \qquad \textsc{Consequence}\quad \frac{P_1 \rightarrow P_2 \quad \{P_2\}\,S\,\{Q_2\} \quad Q_2 \rightarrow Q_1}{\{P_1\}\,S\,\{Q_1\}}$$

$$\textsc{Iteration}\quad \frac{\{P \wedge B\}\,S\,\{P\}}{\{P\}\,\textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\,\{\neg B \wedge P\}}$$

Figure 2.1: Axioms and Rules of Hoare Logic

Figure 2.1 gives the axiom schemas and inference rules of empty statement, assignment, composition, branch, consequence, and iteration. The EMPTY axiom is the simplest one: **skip** does not change the state of the program. In the ASSIGNMENT axiom, $P[E/x]$ denotes the assertion P in which each free occurrence of x has been replaced by the expression E. This axiom means the truth of assertion P after the assignment depends on the truth of $P[E/x]$ before the assignment. The rule of COMPOSITION is quite easy to understand: with the postcondition of S and the precondition of T are the same, then we can conclude the Hoare triple of the sequential execution of S and T. The BRANCH rule states that two branches of the program share the same postcondition as the global postcondition while their preconditions have the unnegated and negated condition B respectively. Technically it is not included in Hoare's original publication but is derived from the other Hoare rules by program transformation. The CONSEQUENCE rule is applied to strengthen the precondition and/or to weaken the postcondition. In the Iteration rule, P is the loop invariant, which is to be preserved by the loop body S. It still holds after the loop is finished because of $\neg B$.

It should be noted that the original Hoare logic can only be used to prove partial correctness because the rule for **while** construct gives no basis for a proof of termination. This can be fixed by adding a loop invariant whose value strictly decreases with respect to a well-founded relation on some domain during each iteration [Reynolds, 1998]. In [Hoare, 1969], very limited kinds of constructs can be proved formally. But the axiomatic treatment of programming languages does not rely on concrete language constructs and the whole framework of Hoare logic can be extended. Over the years, more kinds of programs have been shown to be verifiable by adding more and more rules for other constructs under the paradigm of Hoare logic.

2.1.2 Separation logic

Separation logic is one of the most promising extensions for low-level programs manipulating structured data containing embedded pointers among various extensions of Hoare logic [O'Hearn et al., 2001; Reynolds, 2002]. The programs which can be verified by separation logic can access/modify shared data structures and explicitly allocate/deallocate storage, as required by programs written in C and Java. In principle, Hoare logic is sufficient to deal with pointer operations but there is a mismatch between the simple computational intuition of pointer operations and the complexity of their axiomatic treatments. To be more specific, one feature of the pointers is aliasing, a situation where the same memory cell is accessed by several pointers. When there is aliasing, a change of a single memory location may affect many syntactically unrelated expressions, which will lead to extremely complex logical entailment.

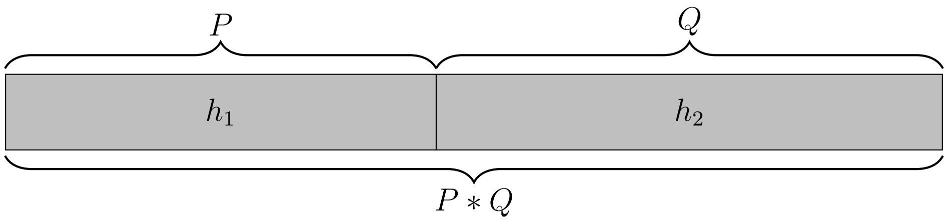


Figure 2.2: Separating Conjunction

To avoid this difficulty, separation logic introduces a novel logical operator $*$ called separating conjunction or spatial conjunction to prohibit sharing. An assertion $P * Q$ can be interpreted "P and Q hold for disjoint portions of the addressable storage (which is usually the heap in computer memory)". As illustrated in Figure 2.2, $P * Q$ is true of a heap if it can be split into two disjoint heaplets, one of which makes P true and the other of which makes Q true. This can be formally written as:

$$h \models P * Q \overset{\text{def}}{=} \exists\, h_1, h_2 \text{ s.t. } h_1 \oplus h_2 = h \wedge h_1 \models P \wedge h_2 \models Q \tag{2.1}$$

where $h_1 \models P$ means P holds in heap h and $h_1 \oplus h_2 = h$ means heap h can be split into disjoint heaplets h_1 and h_2. In the assertion $P*Q$, P and Q are local assertions on isolated addressable storage which they actually access. So in separation logic, assertions of a program component only describe the portion of memory used by the component. A distinction between $*$ and Boolean conjunction $\wedge$ is that $P * P \neq P$ whereas $P \wedge P = P$.

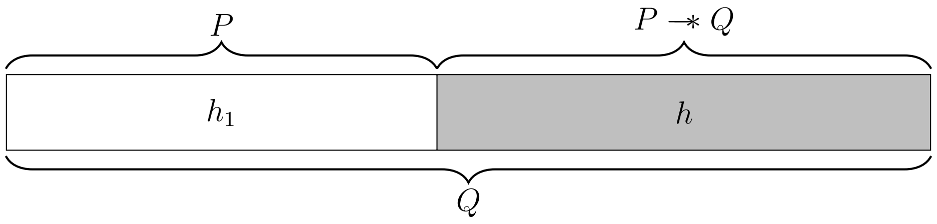


Figure 2.3: Separating Implication

There is another connective, the separating implication or "magic wand".

$P \mathrel{-\!\!*} Q$ means if the current heaplet can be extended with a disjoint heaplet satisfying P, the combined heaplet will satisfy Q. As illustrated in Figure 2.3, $P \mathrel{-\!\!*} Q$ can be formally defined as:

$$h \models P \mathrel{-\!\!*} Q \overset{\text{def}}{=} \forall h_1, h_2 \,.\, h_1 \oplus h = h_2 \rightarrow h_1 \models P \rightarrow h_2 \models Q \tag{2.2}$$

The relation between $*$ and $\mathrel{-\!\!*}$ is like the relation between conjunction $\wedge$ and implication $\rightarrow$ in standard logic:

$$\forall P, Q \,.\, P * (P \mathrel{-\!\!*} Q) \vdash Q \tag{2.3}$$

where $\vdash$ is read as "entails" and is defined as:

$$P \vdash Q \overset{\text{def}}{=} \exists\, h \text{ s.t. } h \models P \rightarrow h \models Q. \tag{2.4}$$

Finally there is an assertion `emp` which means "the heaplet is empty". It is the unit of $*$, meaning that we have $P = \mathsf{emp} * P = P * \mathsf{emp}$.

$$\textsc{Store}\quad \frac{}{\{x \mapsto -\}\, [x] := v\, \{x \mapsto v\}} \qquad \textsc{Load}\quad \frac{}{\{x \mapsto v\}\, y := [x]\, \{y = v \wedge x \mapsto v\}}$$

$$\textsc{Alloc}\quad \frac{}{\{\mathsf{emp}\}\, x := \mathtt{alloc}()\, \{x \mapsto -\}} \qquad \textsc{DeAlloc}\quad \frac{}{\{x \mapsto -\}\, \mathtt{alloc}(x)\, \{\mathsf{emp}\}}$$

$$\textsc{Frame}\quad \frac{\{P\}\, C\, \{Q\}}{\{P * F\}\, C\, \{Q * F\}}\ \mathsf{FreeVar}(F) \cap \mathsf{ModVar}(C) = \emptyset$$

$$x \mapsto - \overset{\text{def}}{=} \exists v \,\text{s.t.}\, x \mapsto v$$

Figure 2.4: Axioms and Rules of Separation Logic

The introduction of $*$ and $\mathrel{-\!\!*}$ does not only simplify the specifications greatly for reasoning but also provides the benefit of scalability. Figure 2.4

gives axiom schemas and inference rules in separation logic. Here the notation $x \mapsto v$ says that pointer variable x holds the address of a memory location where value v is stored. In other words, x points to v. The four axioms, Store, Load, Alloc and DeAlloc are quite straightforward. The inference rule Frame, the key feature of separation logic, says that an assertion which holds for some portion of the addressable storage would still hold for any expansion of that portion. The Frame rule has a side condition, which says that no variable occurring free in the "frame" F (i.e., $\mathsf{FreeVar}(F)$) is modified by the program C (i.e., $\mathsf{ModVar}(C)$). This rule is vital for scalability because one can extend a local specification by adding arbitrary assertions about disjoint parts of the heap. From another perspective, once a specification of C on a heap is proved, it can be reused on any bigger heap containing the original. This Frame rule makes compositional proofs of programs with list or tree-like data structures easier. It should be noted that researchers further developed concurrent separation logic (CSL), which shows efficient reasoning about threads that share access to storage. Although CSL has made significant progress, it is outside the scope of this thesis.

Now separation logic is a cornerstone for efficient proof search about programs in automatic and semi-automatic proof tools. In a recent survey, Peter O'Hearn said "separation logic is a key development in formal reasoning about programs, opening up new lines of attack on longstanding problems" [O'Hearn, 2019].

2.1.3 Ramification

Hobor and Villard [2013] provide a compositional proof system to address the more difficult "ramification" problem rather than the "frame" problem which has been successfully handled by separation logic. The term "ramification" refers to the problem of reasoning about the indirect (global) consequences of (local) actions which naturally arises in verification of programs having intrinsic shared data structures with deep aliasing inside such as graphs, because changes in one part of the structure (e.g. the left child of a graph node) can affect other parts (e.g. the right child or other descendants) which may point into it. This problem is difficult under the framework of separation logic because the data structures involved cannot easily fit into the form $P * Q$: their parts are not usually disjoint. They attack this problem by proposing a proof rule called Ramify:

$$\textsc{Ramify}\quad \frac{\{L_1\}\,C\,\{L_2\} \qquad G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\,C\,\{G_2\}} \tag{2.5}$$

$$\text{where } \mathsf{FreeVar}(L_2 \mathrel{-\!\!*} G_2) \cap \mathsf{ModVar}(C) = \emptyset$$

which actually splits the verification process into two different parts: the verification of programs as transformations on a simple local specification ($\{L_1\}\,C\,\{L_2\}$) which ignores the global context and the verification of a ramification part ($G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)$) which is the result of the changes on the global state. Generally speaking, proving $\{L_1\}\,C\,\{L_2\}$ is easier than proving $\{G_1\}\,C\,\{G_2\}$ directly while proving the ramification part needs significant effort. The good news is that usually one only needs to prove the ramification part for a particular data structure once. In the paper they proved the ramification for spatial graphs and dags (directed acyclic graphs) so as to reuse them in various proofs. More technical details including the enhancement of $G_1 \vdash L_1 * (L_2 \mathrel{-\!\!*} G_2)$ would be explained in §4.2 because the mechanization work is built on it.

2.2 Dependent Type Theory

Martin-Löf [1998] introduced dependent types—types that depend on values in his intuitionistic theory of types, which strengthens the connection between programming and logic because it can be used to encode the universal quantifier $\forall$ and the existential quantifier $\exists$. In 1934, Haskell Curry observed that the types of the combinators can be seen as axiom-schemes for intuitionistic implicational logic. Further investigations showed that this is not a coincidence: various proofs systems can be interpreted as typed variants of lambda calculus. This observation, referred to as the Curry–Howard correspondence, reveals that the two families of formalization—the proof systems and the models of computation which seem unrelated are in fact structurally identical. That is to say: a proof is a program, and the proposition it proves is a type for the program. Dependent types play important roles because the correspondence of predicate logic which is the most widely used logic to make assertions in mathematics and program verification is the dependent type system.

The key concepts in dependent type systems are the dependent product type $\prod_{(x:A)} B(x)$ and the dependent sum type $\sum_{(x:A)} B(x)$. The former captures the idea of a function when the argument is of type A and the type of the codomain varies depending on the value of the argument. The latter captures the idea of a pair (a, b) of type $\sum_{(x:A)} B(x)$ where a has type A and b has type $B(a)$. Under the Curry–Howard correspondence, the type $\prod_{(x:A)} B(x)$ can be seen as a proposition about the universal quantifier: $\forall x \in A, B(x)$. A function f of type $\prod_{(x:A)} B(x)$ can be seen as a constructive proof of this proposition because for every $x \in A$, $f(x)$ whose type is $B(x)$ is a proof of $B(x)$. Similarly, the type $\sum_{(x:A)} B(x)$ can be seen as a proposition about the existential quantifier: $\exists x \in A, B(x)$. A pair (a, b) of type $\sum_{(x:A)} B(x)$ can be seen as a constructive proof of this proposition because it gives a which is an instance of type A and a value b with type $B(a)$ which can be seen as a proof of a satisfying the predicate B.

The Curry–Howard correspondence implies that propositions in predicate logic can be written as dependent type signatures and proofs of such propositions can be written as programs. Then a type checker of dependent type programming languages can be used to check the correctness of a proof. If the type checker is trustable, the correctness of all proofs passed by the type checker is guaranteed.

2.3 Coq

Coq [Coq Development Team, 2019] is one of several proof assistants which have been developed to make it possible to construct proofs in dependent type theories and have them formally checked by computer [Constable et al., 1986; Magnusson and Nordström, 1994; Pollack, 1994; de Moura et al., 2015]. The theoretical foundation of Coq is a formal system called the Calculus of Constructions [Coquand and Huet, 1988] which is a extension of dependent type theory by adding polymorphism and higher-kinded type constructors. The theory is developed alongside Coq. So in 1991 it was extended as Calculus of Inductive Constructions by adding inductive types and then extended as Calculus of (Co)inductive Constructions by adding coinductive types.

Besides giving an interactive environment that allows users to develop proofs interactively, Coq has several advantages over other proof assistants. It is based on a fully-featured higher-order functional programming language Gallina which can be used to construct complex functions and propositions. All functions in Coq must terminate, which is often not a problem. But sometimes extra effort is needed to prove the termination of a recursive function. Although Coq provides very complex decision procedures and rich mechanisms to search possible proofs, the proof terms produced by Coq would be checked by a standalone proof checker which is intentionally small to limit the risk of implementation bugs. This feature greatly enhances the reliability of Coq because the correctness is built on the trust of a simple type checker rather than a highly complicated system. Writing out all formal proofs in full detail could be a serious obstacle to practicability and productivity. Coq is only a semi-automated theorem prover but it provides Ltac, a Turing-complete tactic language which can by used to write automatic theorem proving tactics and decision procedures conveniently while the small kernel feature ensures that users do not need to worry about tricking the system into accepting invalid proofs. Furthermore, since the programs and proof terms in Coq are in the same syntactic class, it is possible to write programs that construct proofs themselves. Such programs that are verified to obey their specifications are called certified.

As mentioned in §1, the most successful applications of Coq include the proof of the four-color theorem [Gonthier, 2008], the proof of Feit–Thompson theorem [Gonthier et al., 2013] and CompCert, a formally verified optimizing compiler for a subset of the C programming language [Leroy et al., 2012].

Chlipala [2013] systematically discusses the practical technology of program verification in Coq by emphasizing programming with dependent types and proving with scripted proof automation. He gives quite a lot of invaluable advice and demonstrations for researchers to exploit the full potential of Coq in their own research problems. Coq is a very powerful and complicated system with a very steep learning curve. Even for experienced users, it is tempting to write proof scripts that manipulate proof goals directly with no structure to aid readers or ease the maintainability. Chlipala's work fulfills the gap between naïve approaches for toy examples and sophisticated techniques to utilize Coq productively.

He does not only discusses the traditional topics of dependent typed programs such as length-indexed lists but also introduces several advanced concepts such as subset types and well-founded recursion. The latter can be used to define more general recursive functions which plays a crucial role in the work of §3. To ensure the termination of all programs, Coq uses a small set of conservative, syntactic criteria to check termination of recursive definitions. Those criteria are insufficient to support the encoding of many natural recursive functions such as merge sort. Well-founded recursion is a standard technique in Coq that allow users to establish a well-founded relation to guarantee that there are no infinite chains of nested recursive calls. It is a workaround that leaves the obligation of proving termination to the users.

In addition, Chlipala [2013] illustrates how to write tactics effectively to handle routine proof obligations as a “design pattern” to reliably avoid the really grungy parts of theorem proving. Consistent use of these custom tactics as understandable artifacts in proofs makes the whole proof more readable and greatly improves the proving productivity.

2.4 Applications

2.4.1 Hoare type theory

It is not surprising that both Hoare logic and separation logic can be formalized in Coq because they are still in the scope of predicate logic. Hoare type theory [Nanevski et al., 2008a] incorporates Hoare-style specifications into types to statically track the side effect. The key concept in this theory is the Hoare type $\{P\}\, x : A\, \{Q\}$ specifying computations with precondition P and postcondition Q which returns a result of type A. This theory was implemented as an axiomatic extension to Coq called "Ynot" [Nanevski et al., 2008b] in a monadic style to include effectful computations, which can be seen as a generalization of the well-known type-and-effect system [Gifford and Lucassen, 1986]. Ynot also supports separation logic by defining a separation monad `STsep` which allows separation specifications on top of the original Hoare type.

All primitives such as `read`, `write`, `new`, `free` are defined on a concrete memory model. In other words, Ynot embeds an imperative programming language in Coq and users can use this language to write programs with Hoare-style specification as their type signature. Coq forces users to prove that the definitions of functions satisfy their specification.

2.4.2 Separation algebras

Appel et al. [2014] also implement separation logic in Coq but with a quite different approach. They propose separation algebras as models of separation logic and indirection theory for constructing step-indexed separation algebras. As mentioned earlier, the assertion $P * Q$ in separation logic means that P and Q hold on disjoint subheaps. Recall that the definition of $P * Q$ in (2.1) includes that entire heap h. Typically heaps like h are modeled as partial functions [Nanevski et al., 2008b] but Appel et al. [2014] abstracted $h_1 \oplus h_2 = h$ as a three-place relation `join` which cannot be simplified as a binary function because $h_1 \oplus h_2$ may not exist if they overlap.

Instead of giving the definition of `join` directly, a pure mathematical structure called separation algebra composed by several laws like commutativity is given and `join` is claimed the operator of the separation algebra. More separation related algebras are defined by adding extra axioms into the separation algebra to provide flexibility. Thus the model of separation logic has been split into two layers: a `join` relation satisfying certain laws and the separating operator $*$ defined in terms of `join`. With such a definition, the inference rules for $*$ can be derived from the laws of separation algebra.

This definition which does not rely on any particular heap model is so abstract and general that it can serve as a basis for a wide variety of logical theories. Theorems built on this definition can directly apply to any concrete structures satisfying the laws of those algebras. Appel et al. [2014] developed a library called Mechanized Semantic Library (MSL) in Coq which contains definitions of those algebras implemented as type classes. Once a concrete definition such as a partial function is proved as a instance of a specific type class, all theorems related to the particular type class can be called directly in the proofs about the definition.

To support the representation and deduction of recursive predicates which appear frequently in reasoning using separation logic, MSL provides two mechanisms: covariant recursion (i.e., Tarski's fixed-point, see Tarski et al., 1955) and contravariant recursion [Ahmed, 2006]. Both provide a fixed-point function μ which given F of type $A \to A$, μF is a fixed point of F, i.e., $F(\mu F) = \mu F$. Then to get the effect of a self-referencing predicate $P(x) = \ldots x \ldots P \ldots$, one can define

$$F(p) = \lambda x.(\ldots x \ldots p \ldots) \quad \text{and} \quad P = \mu F$$

Then $P(x) = (\mu F)(x) = F(\mu F)(x) = F(P)(x) = (\ldots x \ldots P \ldots)$. The difference is that when the functor F is covariant, the construction of μ is easier than when F is contravariant. For covariant F, fixed points are more easily constructed by the Knaster-Tarski fixed-point theorem [Appel et al., 2014]. The trick to find fixed points for contravariant F is to consider a data type as a sequence of accurate approximations taken successively.

The general formulation of this idea as step-indexing is called indirection theory which is implemented in MSL [Hobor et al., 2010].

Chapter 3

A Reusable Library of Formalized Graph Theory

It is common to adopt terms from a mathematical domain in the specification of programs, especially when the concepts in the domain are well studied and widely used, as in graph theory. Since we focus on graph-manipulating programs, it is very natural to use the terminology of graph theory in the specification to describe the behavior of those programs. As a result, an essential component of our work is a library of formalized graph theory: a framework which provides the definitions of various concepts from graph theory such as graph, path, reachability, etc. along with necessary theorems for reasoning about properties of various kinds of graphs in different programs.

This chapter introduces our general-purpose, reusable library of formalized graph theory. It is designed to be expressive and powerful enough to adapt a large variety of application scenarios while also having good modularity to make the theorems in the library highly reusable. The first section is about the formalization and organization of concepts of graph theory, in the context of type theory. The second section is about the hundreds of general theorems proved around those concepts in our library. The last section is a literature review of other existing graph reasoning libraries.

3.1 Formalization of Key Concepts in Graph Theory

The starting point of a mathematical theory, formalized or not, is always the definition of the objects of study in that theory. This section explains how we formalize graph theory objects such as graph, path, and reachability. There are more than 200 definitions of entities, relations, predicates, and operations in our formalized graph library. It is unnecessary and impossible to explain the details of every definition, so we will only show the overall design principle and several key definitions.

§3.1.1 introduces the design decisions and hierarchical definitions of the core concept—graphs—in our library. §3.1.2 illustrates the definitions of two closely related concepts, path and reachability. The latter is the most widely used concept in our work and relies on the definition of the former. §3.1.3 to §3.1.5 enumerate several important relations, operations, and predicates about graphs respectively.

3.1.1 Definitions of graphs

There are two major challenges in formalizing concepts from graph theory. One is that graph theory is usually based on set theory but our formalization has to be based on type theory. The two underlying theories are incompatible, and so it is impossible to translate the definitions from textbooks directly. The other challenge is in balancing the dichotomy between the two objectives of the library. The definitions should be as general as possible so that we can reuse them in any application. At the same time, they should also be representative enough when necessary so that we are able to prove the unique properties in concrete cases.

There are two obvious options to resolve the inconsistency raised by the first challenge. One is formalizing set theory in Coq first as the foundation of graph theory, and building a graph theory atop that in a natural manner. The other is formalizing graph theory directly. We choose the latter one for the following reasons. First, set theory has a huge amount of content. Formalizing it in its entirety is a daunting and distracting task. Second, representing basic concepts and terms of graph theory directly in type theory is not very difficult. The price of converting the language of graph theory from set theory to type theory is affordable. Third, adding set theory as an extra layer between graph theory and type theory would prevent us from taking advantage of Coq's built-in support for type-related constructions directly for graphs, which could cause lengthy and tedious proofs.

Most textbooks about graph theory do not specify a formal foundation explicitly, but can generally be considered as relying on set theory. A typical definition [Bondy and Murty, 2008] using set theory is[1]:

Definition 3.1. *A directed graph G is an ordered pair (V, E) consisting of a set V of vertices, and a set E, disjoint from V, of edges, together with an incidence function ψ_G that associates with each edge of G an ordered pair of (not necessarily distinct) vertices of G. If e is an edge and $\psi_G(e) = (u, v)$, then e is said to join u to v. The vertex u is the destination of e, and the vertex v its source; they are the two ends of e.*

Bondy and Murty [2008] actually give two definitions for undirected and directed graphs respectively. The only difference is that ψ_G gives unordered pairs of vertices if G is undirected. We argue that the definition of directed graph is general because an undirected graph can be seen as a special

[1]We change some terminology for consistency.

directed graph by doubling the edges, i.e. an edge e indicating a two-way relationship between u and v can be replaced by two edges, one from u to v and the other from v to u. It should also be noted that Definition 3.1 is a very general definition of graph. For instance, it does not treat the elements of E as 2-element subsets of V, which allows multiple edges from one vertex to another.

According to Definition 3.1, we can formalize graph in Coq analogously. As the first step, we assume two abstract types `Vertex` and `Edge` of vertices and edges. Since they are just assumptions or parameters, they can be instantiated with concrete definitions in further applications. Then we split the incidence function ψ_G into two functions `src` and `dst` mapping edges to their sources and destinations. This splitting is just for convenience to refer to source and destination separately. So far, we have simply translated the Definition 3.1 directly to Coq. We will see that this is just the beginning of the formalization because abstract types such as `Vertex` and `Edge` lack some properties that a set has naturally or implicitly.

In set theory, one element can belong to multiple sets, e.g. to a subset or a superset. Thus, in set theory, when we add or remove vertices and/or edges, we just need to change the set accordingly to represent the result of the operation. But in type theory, one term can only belong to one type, which makes it difficult if not impossible to change the type to represent the result. When a subset would be used in type theory, a common practice is to use a predicate function that returns true if the term is in the subset and returns false if not. So we add two more predicates `vvalid` and `evalid` to classify vertices and edges as *valid* (in) or not (out). Thanks to these two predicates, we can instantiate `Vertex` and `Edge` as non-dependent types instead of dependent types and restrict the range of valid vertices and edges through the two validity predicates at the very beginning. There are at least three benefits of this setting. One is that we can get superset or subset by weakening or strengthening the two predicates. Another is that simple types can simplify our development because in general definitions and proofs about dependent types are more complicated. The third benefit is that we can even present incomplete graphs, e.g. an edge with invalid ends. For this reason, we decide to name our formalization of graph as `PreGraph`. A graph type that can represent missing vertices and edges is convenient for verifying real programs. For example, consider the difference of two graphs, $G_1 - G_2$. Even if both of these graphs are "well-formed" to begin with, in the sense that valid nodes have only valid edges and vice versa, their difference may not be well-formed since there may be dangling edges pointing to the now-removed vertices of G_2.

Besides the subset/superset issue, there is an implicit presumption in set theory which is not true for types in general: decidable equality. A set or type X has decidable equality if we can judge whether any two elements of X are the same or not. In classical mathematics, every set has decidable equality. We found that it is such a fundamental property that almost all sensible graph-manipulating algorithms employ it whether or not they realize it. There is a good reason for this. Unlike trees or lists, a key feature of graphs is that they can have loops. The algorithm must be able to detect loops to terminate, which it does by distinguishing processed vertices from unprocessed ones. Since decidable equality is so important, we have to put decidability of vertices and edges as additional parameters of a graph. Coq has a suitable built-in `Class` called `EqDec` which indicates the decidability of equivalence. For any type `T` and any equivalence relation `equiv` of `T`, `EqDec T equiv` means the type `T` has decidable equality with respect to the equivalence relation `equiv`. We use this in our final definition of `PreGraph`:

```
Definition Ensemble (U: Type) := U -> Prop.
Record PreGraph (Vertex Edge: Type)
```

```
        {EV: EqDec Vertex eq} {EE: EqDec Edge eq} := {
  vvalid: Ensemble Vertex;
  evalid: Ensemble Edge;
  src: Edge -> Vertex;
  dst: Edge -> Vertex }.
```

Here the `Record` construction is a macro in Coq which not just defines a type `PreGraph` with four parameters `Vertex`, `Edge`, `EV`, and `EE` but also defines four functions which all take a `PreGraph` as their first argument. For example, the complete signature of `src` is `PreGraph` → `Edge` → `Vertex`. `EV` and `EE` represent the decidable equality of `Vertex` and `Edge`. As shown in Figure 3.1, a `PreGraph` can contain invalid nodes and edges in an arbitrary configuration. The `PreGraph` is the bedrock of the whole graph library.

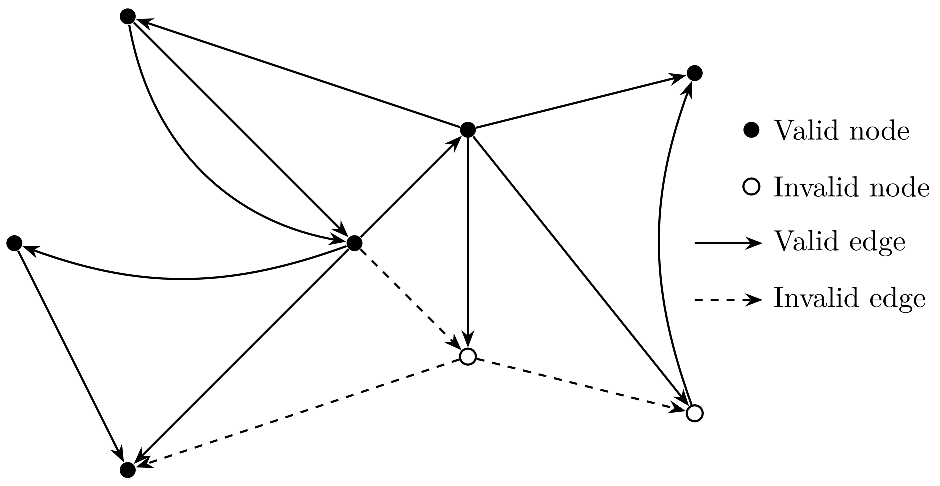


Figure 3.1: A PreGraph with Valid and Invalid Nodes and Edges.

In both theory and programming practice, the bare vertex and edge setting of a `PreGraph` provides too little information to accommodate interesting problems or theorems. For example, many problems and theorems in graph theory are related to various ways of coloring graphs. The famous four-color theorem is a typical case in which each vertex can be assigned a color. In fact, graph coloring is a special case of graph labeling, an assignment of labels to vertices and/or edges. In the shortest path problem, the distances between adjacent vertices are usually treated as labels of edges. Similarly, the network flow problem attaches capacities to edges as labels. To model the graphs in these problems, we define `LabeledGraph`, a `PreGraph` with labels. From the definition below, we can see a `LabeledGraph` can have label (`vlabel` G v) of parametric type `DV` for arbitrary vertex v in G, label (`elabel` G e) of parametric type `DE` for arbitrary edge e and a label (`glabel` G) of parametric type `DG` for the whole graph G. Another key component of `LabeledGraph` is `pg_lg`, which is of type `PreGraph`. That means `LabeledGraph` is built on `PreGraph`. We can always get back a `PreGraph` by `pg_lg` G for a `LabeledGraph` G.

```
Record LabeledGraph {DV DE DG: Type} := {
  pg_lg: PreGraph;
  vlabel: Vertex -> DV;
  elabel: Edge -> DE;
  glabel: DG }.
```

As we mentioned before, the balance between generality and specificity is our second challenge. Until now the two definitions of graph are very universal, but sometimes in a concrete application we need a particular kind of graph which has restrictions that exactly fit the scenario. For example, the data structure used in the classical union-find algorithm—disjoint sets—can be seen as a special graph in which each vertex has only one out-edge. Of course, we can add predicates to specify the unique characteristics of the graph throughout all proofs in each application. At least two drawbacks make this a poor option. One is that it leads to untidy theorem statements because the predicates will need to appear in each related theorem. The other is that it prevents us from reusing theorems about some common characteristic such as finiteness easily and systematically because the predicate is highly specific. A novelty of our library here is that we establish a new `Record` called `GeneralGraph` which augments `LabeledGraph` by binding a predicate about the graph. We call the bound predicate `Sound` the soundness condition:

```
Record GeneralGraph {DV DE DG: Type}
                   {Sound: @LabeledGraph DV DE DG -> Type} := {
  lg_gg: @LabeledGraph DV DE DG;
  sound_gg: Sound lg_gg }.
```

In the definition above, the component `lg_gg` G represents the wrapped `LabeledGraph` and `sound_gg` G means the graph G satisfies the soundness condition `Sound`. This new `GeneralGraph` shrinks the length of the theorem statements. Furthermore, we can define several independent soundness conditions, each of which summarizes one typical feature such as finiteness or acyclicity, and compose them flexibly into one large predicate for different cases. The mechanism of composable soundness condition enables us to achieve an equilibrium between generality and specificity. On the one hand we can define several typical soundness conditions and prove many highly reusable theorems. On the other hand we can combine these soundness conditions to get a highly specific predicates when necessary to adapt to different situations. We will discuss more details about this in §3.1.5.

So far, we defined three `Record`s in Coq: `PreGraph`, `LabeledGraph` and `GeneralGraph`, which form a hierarchical interface of the core concept—graph—in our library. `PreGraph` is the fundamental layer for all structure-related concepts. In fact there are more than 750 functions, predicates and theorems involving `PreGraph` in the whole project, including verification of several programs. `LabeledGraph` is the intermediate layer which supports attaching information to vertices, edges and even the whole graph. There are about 150 functions, predicates and theorems about `LabeledGraph`. `GeneralGraph` is the top layer which provides a concise interface by binding soundness conditions to the `LabeledGraph`. In addition, the soundness conditions are composable, which provides good modularity. In the rest of the chapter, we will introduce more major concepts and important theorems proved in the library.

3.1.2 Path and reachability

In addition to the hierarchical interface of graphs, reachability is one of the most crucial concepts in our library of graph theory. For example, the predicate `reachable` is most widely used all over the library files, which is quite reasonable. No matter what the graph-manipulating program is, it always needs to retrieve information or change the structure along the path in a graph, which is associated with reachability. From this perspective, various algorithms of graphs like DFS, BFS, shortest path, and spanning tree are all of one kind.

There are several high-level treatments for reachability problems [Tarjan, 1981; Dolan, 2013], which describe paths as regular expressions and reachability as the Kleene star A^* of the adjacency matrix A. With such abstractions, a wide range of problems can be solved by standard techniques. Our library is developed from program verification side. We choose a low-level approach to formalize related definitions directly, which avoids formalizing regular algebra theory and matrix theory.

The definition of reachability is based on path, which is a subtle concept to formalize. According to [Bondy and Murty, 2008]:

Definition 3.2. *A path is a simple graph (i.e. no parallel edges) whose vertices can be arranged in a linear sequence in such a way that two vertices are adjacent if they are consecutive in the sequence, and are nonadjacent otherwise.*

From this definition, we can formalize a path as an alternating sequence of vertices and edges: $v_0, e_0, v_1, e_1, v_2, \ldots, v_{k-1}, e_{k-1}, v_k$ because a graph is composed of vertices and edges. But soon we found this to be redundant. For any edge e, its source and destination vertices are known, so the triple v_i, e_i, v_{i+1} is unnecessary. We hoped the definition was both general and concise. It seemed that either a list of vertices or a list of edges would work. But eventually it turned out that neither of these approaches was good enough.

We tried several definitions of path. At first, we chose a list of vertices as the definition, which was fine in most cases until we encountered a multiple-edge graph. For such a graph, a vertex-list representation of path cannot tell us which one of the multiple edges is part of the path. Then we defined a path as a list of edges but this edge-representation has its own flaw: we cannot represent the empty path for certain vertices naturally. Usually a path p in a graph should has a starting vertex v_1 and an ending vertex v_2 (they may be the same) so that we can say v_2 is reachable from v_1 via p. The problem is that a vertex is always reachable from itself, thus it needs an empty path. In a vertex-list representation of path, a singleton list $[v]$ is ideal to be the empty path of vertex v while an edge-list representation is inadequate because one cannot get vertex information from an empty list. To solve this dilemma, we combine the good parts of both candidates together to get our final definition:

```
Definition path: Type := (V * list E)%type.
```

The `path` is defined as a pair of starting vertex with type `V` and an edge list of type `list E`. The edge list resolves the multi-edge issue and the start vertex resolves the empty-path issue. Unlike `PreGraph`, a soundness condition of `path` is mandatory: all edges in the edge list of a path must connect consecutively and the starting vertex of a path must be the same as the source of the first edge in the edge list. When a `path` meets these conditions, we call it a `valid_path`. The definition of `valid_path` and related predicates are listed as follows:

```
Definition strong_evalid (pg: PreGraph V E) (e: E): Prop :=
  evalid pg e /\ vvalid pg (src pg e) /\ vvalid pg (dst pg e).
Fixpoint valid_path' (g: PreGraph V E) (p: list E): Prop :=
  match p with
    | nil => True
    | n :: nil => strong_evalid g n
    | n1 :: ((n2 :: _) as p') => strong_evalid g n1 /\
                dst g n1 = src g n2 /\ valid_path' g p'
  end.
Definition valid_path (g: PreGraph V E) (p: path) :=
  match p with
  | (v, nil) => vvalid g v
  | (v, (e :: _) as p') => v = src g e /\ valid_path' g p'
  end.
```

- The first predicate `strong_evalid` of an edge is an enhanced version of `evalid`, which emphasizes not only the validity of the edge but also the validity of its two ends.
- The second predicate `valid_path'` of an edge list is recursively defined, which emphasizes that the destination of the preceding edge is the same as the source of the succeeding edge besides the `strong_evalid` requirement for every edge.

With the two definitions we can define the final predicate `valid_path` of a path. When the path p only contains one vertex, it is `valid_path` if the vertex is valid. When p contains a vertex v and a non-empty list edge p', then p is valid if v is the source of the first edge of p' and p' is `valid_path'`.

The predicate `valid_path` is just a basic requirement for a `path`: the edges along the path are consecutive. At first glance it is sufficient to define the reachability between two vertices, which just means there exists a valid path connecting them. But during the development we found that sometimes we need to claim that each vertex along the path satisfies a property `P`. We can define this generalized version first then define the original reachability as a special case. So we define two more predicates: `path_prop` and `good_path` first:

```
Definition path_prop (g: PreGraph V E)
                     (P: Ensemble V)  (p: path): Prop :=
  P (fst p) /\ Forall (fun e => P (src g e) /\ P (dst g e))
                      (snd p).
Definition good_path (g: PreGraph V E)
                     (P: Ensemble V): (path -> Prop) :=
  fun p => valid_path g p /\ path_prop g P p.
```

The predicate `path_prop` says that for any path `p`, the head vertex and both ends of each edge `e` of `p` satisfy the property `P`. A path `p` of graph `g` is a `good_path` with respect to property `P`, if `valid_path g p` and `path_prop g P p`. Then we can define the most general predicate about reachability: `reachable_by_path`, which is composed by two predicates in Figure 3.2. One is `good_path` and the other is `path_ends`, which specifies the two ends of the entire path. Here we omit the definitions of `phead` and `pfoot`, which are functions retrieving first and last vertex of a path respectively. The definition of `phead` is trivial and the definition of `pfoot` is straightforward by recursion. It should be noted that the last two lines in Figure 3.2 give a notation to say that `g |= p is n1 ~o~> n2 satisfying P` means in graph `g`, `p` is a path connecting `n1` and `n2` with property `P` for each vertex in `p`.

From the definition of `reachable_by_path`, we can define two more spe-

```
Definition path_ends (g: PreGraph V E) (p: path)
  (n1 n2: V): Prop := phead p = n1 /\ pfoot g p = n2.
Definition reachable_by_path (g: PreGraph V E) (p: path)
              (n: V) (P: Ensemble V): Ensemble V :=
  fun n' => path_ends g p n n' /\ good_path g P p.
Notation "g '|=' p 'is' n1 '~o~>' n2 'satisfying' P" :=
  (reachable_by_path g p n1 P n2) (at level 1).
```

Figure 3.2: Definition of `reachable_by_path` and Its Notation

cial cases: `reachable_by` and `reachable`. We also defined a special notation

```
Definition reachable_by (g: PreGraph V E) (n: V)
                        (P: Ensemble V): Ensemble V :=
  fun n' => exists p, g |= p is n ~o~> n' satisfying P.
Notation " g '|=' n1 '~o~>' n2 'satisfying' P " :=
  (reachable_by g n1 P n2) (at level 1).
Definition reachable (g: PreGraph V E) (n: V): Ensemble V:=
  reachable_by g n (fun _ => True).
```

Figure 3.3: Definition of `reachable_by` and `reachable`

`g |= n1 ~o~> n2 satisfying P` for the predicate `reachable_by` as a short-cut to say that in graph `g`, there is a path from `n1` to `n2` with property `P` for each vertex in this path. When the property `P` is set to be trivially `True` in `reachable_by`, we get the predicate `reachable` as a special case of `reachable_by`. All of these definitions are shown in Figure 3.3.

From the definition of `reachable`, we illustrate that a seemingly simple definition may involve many things. Our `reachable` includes entities like `PreGraph` and `path`; predicates like `path_ends`, `good_path`, `valid_path`, and `path_prop`; and functions like `phead` and `pfoot`. This is very common during the development because formalization always requires unambiguous and detailed definitions. When writing a long definition in detail, one would name intermediate constructions as ancillary definitions for simplicity. Sometimes we have no choice but to write more generic definitions like `reachable_by_path` because both the generic form and the special form are used in the development. Sometimes it is just convenient to have a generic definition because it may be shared as a base definition among multiple definitions or some theorems need generic functions to have easy and short proofs. The same situation happens in formal proving of theorems, which usually leads to several helper lemmas. We will discuss the theorems in the library in §3.2.

There are more functions and predicates about `path`. We will only enumerate them briefly here because they are not as important as `reachable`. We defined a function `path_glue` to concatenate two paths p_1 and p_2 when `pfoot` of p_1 is the same as `phead` of p_2. This condition is abstracted as a predicate `paths_meet`. We defined a function `epath_to_vpath` to convert a path to a list of vertices. This function is used when we need to argue that there are no duplicate vertices in a path. The relation `Subpath` expresses that a path p_1 is contained in another path p_2. Another relation `In_path` says that a vertex v is on a path p. From these concepts about `path`, we proved nearly 80 theorems. Some of them are almost trivial but widely used in later development. For example, theorem `reachable_foot_valid` says that the last vertex of a path is valid. Some of them are about the relations among these path-derived concepts. For example, `valid_path_split` says that if two paths satisfy `paths_meet` and the `path_glue` of them is `valid_path`, then both of them satisfy `valid_path`. We did not prove these lemmas at first, but rather we encountered a need for them when proving more complex lemmas. In such cases, we tried to prove the most general forms of the lemmas, thus encouraging the clever reuse of such concepts. Again, we will discuss some interesting ones about `path` in §3.2.

3.1.3 Relations between graphs

The concepts defined previously are far from complete for a mature graph library. For example, it should be able to express the relations between the states of a graph before and after the execution of graph-manipulating programs. In general, the relation is unique for each particular graph-manipulating algorithm. Besides those concrete, algorithm-specific relations which will be discussed in §5, here we introduce two generic relations between two graphs: the subgraph relation and graph isomorphism.

The relation `is_partial_graph` defined below says that `g1` is a subgraph of `g2`. The definition is composed by four conjunctions, which are all reasonable and straightforward requirements.

- The first two conjunctions state that if a vertex or an edge is valid in `g1`, then it must be valid in `g2`.
- The remaining two conjunctions ensure that if the two end-vertices of a valid edge `e` are valid in `g1`, then they are still the two end-vertices of the same `e` in `g2`, respectively.

To justify our definition, we proved that `is_partial_graph` is a reflexive and transitive relation, capturing that a graph is a subgraph of itself and that if a graph g_1 is a subgraph of g_2, g_2 is a subgraph of g_3, then g_1 is a subgraph of g_3.

```
Definition is_partial_graph (g1 g2: PreGraph V E) :=
   (forall v: V, vvalid g1 v -> vvalid g2 v) /\
   (forall e: E, evalid g1 e -> evalid g2 e) /\
   (forall e: E, evalid g1 e -> vvalid g1 (src g1 e) ->
                 src g1 e = src g2 e) /\
   (forall e: E, evalid g1 e -> vvalid g1 (dst g1 e) ->
                 dst g1 e = dst g2 e).
```

We also proved the following basic fact when `path` is involved: if `g1` is a subgraph of `g2`, then any valid path in `g1` is also a valid path in `g2`. This basic fact is necessary for three other theorems: the `reachable_by_path`, `reachable_by` and `reachable` are all preserved when switching from `g1` to `g2` if `g1` is a subgraph of `g2`. We do not provide subgraph relationship for `LabeledGraph` and `GeneralGraph` so far because empirically we have found that labels are often changed during the execution of a program and the soundness condition does not hold for the subgraph.

Graph isomorphism is more complicated than the subgraph relation. The typical definition [Bondy and Murty, 2008] of graph isomorphism is:

Definition 3.3. *In general, two graphs G and H are isomorphic, written $G \cong H$, if there are bijections $\theta : V(G) \to V(H)$ and $\varphi : E(G) \to E(H)$ such that $\psi_G(e) = (u, v)$ if and only if $\psi_H(\varphi(e)) = (\theta(u), \theta(v))$; such a pair of mappings is called an isomorphism between G and H.*

Recall that ψ_G defined in Definition 3.1 on page 35 is the incidence function of a graph G which associates each edge with its own end-vertices. In our corresponding definition of `PreGraph`, it is split into two functions `src` and `dst`. According to Definition 3.3, we first define bijection as follows:

```
Record bijective {A B} (f: A -> B) (invf: B -> A): Prop :=
  { injective: forall x y, f x = f y -> x = y;
    surjective: forall x, f (invf x) = x; }.
```

Unlike the traditional definition, this `bijective` comes with two functions, `f` and its inverse `invf`. The exposure of the inverse function eases the statement of the symmetric law: if a function f is a bijection, then its inverse function is also a bijection. This is proved as a lemma `bijective_sym`:

```
Lemma bijective_sym: forall {A B} (f: A -> B) (invf: B -> A),
    bijective f invf -> bijective invf f.
```

Then we define our most generic graph isomorphism predicate in Figure 3.4. The definition of `pregraph_isomorphism_explicit` looks long and compli-

```
Record pregraph_isomorphism_explicit
       ‘(g: @PreGraph V E EV EE)
       ‘(g’: @PreGraph V’ E’ EV’ EE’)
       (vmap: V -> V’) (vmap’: V’ -> V)
       (emap: E -> E’) (emap’: E’ -> E): Prop :=
  { vertex_bij: bijective vmap vmap’;
    edge_bij: bijective emap emap’;
    vvalid_bij: forall v, vvalid g v -> vvalid g’ (vmap v);
    vvalid_bij_inv: forall v’, vvalid g’ v’ ->
                               vvalid g (vmap’ v’);
    evalid_bij: forall e, evalid g e -> evalid g’ (emap e);
    evalid_bij_inv: forall e’, evalid g’ e’ ->
                               evalid g (emap’ e’);
    src_bij: forall e, evalid g e ->
                       vmap (src g e) = src g’ (emap e);
    dst_bij: forall e, evalid g e ->
                       vmap (dst g e) = dst g’ (emap e); }.
```

Figure 3.4: The Definition of Isomorphism Between `PreGraphs`

cated but it is actually very intuitive. The first two conditions establish the type level bijections. The next four properties, from `valid_bij` to `evalid_bij_inv`, establish the bijection between valid vertices and edges. The last two, `src_bij` and `dst_bij`, say that the corresponding vertices in two graphs are connected in the same way. This is exactly what Definition 3.3 says. The conditions `vvalid_bij` and `vvalid_bij_inv` together are equivalent to `forall v, vvalid g v <-> vvalid g’ (vmap v)`. So are `evalid_bij` and `evalid_bij_inv`. We justify our definition of graph isomorphism by proving that it is an equivalence relation. It satisfies the reflexive law, the symmetric law and the transitive law. Unlike the subgraph relation, we also define the isomorphism between two `LabeledGraphs`. This isomorphism is the isomorphism between `PreGraphs` with additional equiv-

```
Record label_preserving_graph_isomorphism_explicit
       `(g: @LabeledGraph V E EV EE DV DE DG)
       (g': @LabeledGraph V E EV EE DV DE DG)
       (vmap vmap': V -> V) (emap emap': E -> E): Prop :=
  { lp_pregraph_iso:
    pregraph_isomorphism_explicit g g' vmap vmap' emap emap';
    vlabel_iso: forall v, vvalid g v ->
                  vlabel g v = vlabel g' (vmap v);
    elabel_iso: forall e, evalid g e ->
                  elabel g e = elabel g' (emap e); }.
```

Figure 3.5: The Definition of Isomorphism Between `LabeledGraph`s

alence between corresponding labels of vertices and edges. It is less generic than isomorphism between `PreGraph`s because in the definition above, the types of vertices, edges and labels are all the same. It is still reasonable because when we consider the equivalence between labels, it is more often about the graphs before and after some modification. The types are not changed in that case. Of course, we proved that the isomorphism between `LabeledGraph`s is an equivalence relation too.

For some applications, `pregraph_isomorphism_explicit` is cumbersome. The types of vertices and edges are the same and the bijections between vertices and edges are simply identity functions. So we defined a simplified version of isomorphism between `PreGraph`s in Figure 3.6. This definition is widely used in the verification of algorithms which do not change the whole or part of the structure of graphs. We even defined a notation `g1 ~=~ g2` as a shortcut due to its frequent appearance. Moreover, we defined a simplified version of isomorphism between `LabeledGraph`s. As usual, we proved independently that both simplified versions of isomorphism are equivalence relations.

```
Definition labeled_graph_equiv (g1 g2: Graph) :=
  g1 ~=~ g2 /\ (forall v, vvalid g1 v -> vvalid g2 v ->
    Definition structurally_identical
               (g1 g2: PreGraph Vertex Edge): Prop :=
      (forall v: Vertex, (vvalid g1 v <-> vvalid g2 v)) /\
      (forall e: Edge, (evalid g1 e <-> evalid g2 e)) /\
      (forall e: Edge, evalid g1 e -> evalid g2 e ->
                                       src g1 e = src g2 e) /\
      (forall e: Edge, evalid g1 e -> evalid g2 e ->
                                       dst g1 e = dst g2 e).
    Notation ”g1 ’~=~’ g2” := (structurally_identical g1 g2).
```

Figure 3.6: Definition of `PreGraph` Isomorphism

```
                        vlabel g1 v = vlabel g2 v) /\
  (forall e, evalid g1 e -> evalid g2 e ->
                          elabel g1 e = elabel g2 e).
```

3.1.4 Operations of graphs

In computer programming, there is an acronym CRUD which describes the four basic kinds of operations of persistent storage: create, read, update, and delete. We can classify our defined operations of graphs in a similar way. Note that Gallina—the programming language we used to define entities—is purely functional. So the operations shown here do not modify the graph in place but return the possibly changed graph.

All operations discussed here are about `PreGraph` and `LabeledGraph`. We do not consider general operations of `GeneralGraph` because the soundness condition is about the underlying `LabeledGraph`. We need to prove that the modified `LabeledGraph` still satisfies the soundness condition if we want to get a modified `GeneralGraph`. The proof depends on the concrete definition of the soundness condition. Sometimes the change of the graph breaks the soundness condition in the intermediate steps of certain algorithms.

Instead of defining each of these operations from scratch separately, we defined some generic helper functions first because we found many similarities between `vvalid` and `evalid`, `src` and `dst`.

```
Definition addValidFunc {T: Type} (v: T)
           (validFunc: Ensemble T): Ensemble T :=
           fun n => validFunc n \/ n = v.
Definition removeValidFunc {T: Type} (v: T)
           (validFunc: Ensemble T): Ensemble T :=
           fun n => validFunc n /\ n <> v.
Definition updateEdgeFunc (edgeFunc: E -> V) (e: E) (v: V):
  E -> V := fun n => if equiv_dec e n then v else edgeFunc n.
```

The three definitions above are all so-called higher-order functions because they take functions as arguments and return functions applying the function parameters. From the definition we can see the an element `n` of type `T` satisfies the new predicate generated by `addValidFunc` if and only if `validFunc n` or `n` equals `v`. In other words, the `addValidFunc` returns an enlarged validity predicate by adding an element `v`. Also, an element `n` satisfies the new predicate generated by `removeValidFunc` if and only if it is not `v` and `validFunc n` holds. It means that `removeValidFunc` returns a shrunk predicate by removing `v`. The function `updateEdgeFunc` updates the edge function `edgeFunc`, no matter it is `src` or `dst`. For any edge `n`, if it equals `e` then the generated function returns the updated vertex `v`, otherwise it returns the same vertex returned by `edgeFunc`.

Create. We defined operations which add a vertex or an edge to an existing graph as follows. The first line just means `Graph` is a shortcut of `PreGraph V E`.

```
Notation Graph := (PreGraph V E).
Definition pregraph_add_vertex (g: Graph) (v: V): Graph :=
  @Build_PreGraph V E EV EE (addValidFunc v (vvalid g))
                            (evalid g) (src g) (dst g).
Definition pregraph_add_edge (g: Graph) (e: E) (o t: V) :=
  @Build_PreGraph V E EV EE (vvalid g)
  (addValidFunc e (evalid g))
  (updateEdgeFunc (src g) e o) (updateEdgeFunc (dst g) e t).
Definition pregraph_add_whole_edge (g: Graph) (e: E)
  (s t: V) := Build_PreGraph _ _ (addValidFunc t (vvalid g))
  (addValidFunc e (evalid g))
  (updateEdgeFunc (src g) e s) (updateEdgeFunc (dst g) e t).
```

Adding a single vertex is quite simple: only the `vvalid` predicate needs to be changed. When adding a new edge, there are different cases. If we only need to link two existing vertices, we can use `pregraph_add_edge`. It adds a new edge by modify `evalid` and specifies two end-vertices by updating `src` and `dst`. If the end-vertices of the new edge are also new, one can use `pregraph_add_vertex` to add vertices first before adding edges. In applications, we found that it is more common that the destination vertex is new. So we provide the function `pregraph_add_whole_edge` to add a new edge and its new vertex together. The graph illustrated in Figure 3.1 can be constructed through operations introduced here.

Read. Some inquiries about the components of graphs are parts of the definitions of graphs themselves. They are `src`, `dst`, `vlabel`, `elabel` and etc. Besides these simple inquiries, there are other kinds of operations which can also be classified as reading operations, which are actually filters to retrieve a subgraph from the original graph. Their definitions along with the helper functions are listed below.

```
Definition predicate_vvalid (g: Graph) (p: V -> Prop)
   : Ensemble V := fun n => vvalid g n /\ p n.
Definition predicate_evalid (g: Graph) (p: V -> Prop)
   : Ensemble E := fun e => evalid g e /\
                             p (src g e) /\ p (dst g e).
Definition predicate_weak_evalid (g: Graph) (p: V -> Prop)
   : Ensemble E := fun e => evalid g e /\ p (src g e).
Definition predicate_subgraph (g: Graph) (p: V -> Prop)
   : Graph := Build_PreGraph EV EE (predicate_vvalid g p)
               (predicate_evalid g p) (src g) (dst g).
Definition predicate_partialgraph (g: Graph) (p: V -> Prop)
   : Graph := Build_PreGraph EV EE (predicate_vvalid g p)
               (predicate_weak_evalid g p) (src g) (dst g).
```

The first three of the definitions above are helper functions which enhance the validity predicates by requiring additional property `p` for vertices. According to the definition of `predicate_subgraph`, the valid vertices in the returned graph are valid vertices which also satisfy `p` in the original graph. Also, the valid edges in the returned graph are valid edges whose both end-vertices satisfy `p` in the original graph. The only difference between `predicate_subgraph` and `predicate_partialgraph` is that a valid edge is also valid in the graph returned by `predicate_partialgraph` if its source, not both ends, is valid in the original graph. Both definitions only modify the validity predicates. Here we specify a mathematical notation ↑ to denote the concept of `predicate_subgraph`:

$$\gamma \uparrow P \overset{\text{def}}{=} \texttt{predicate\_subgraph}\ \gamma\ P \tag{3.1}$$

When the functions for `PreGraph` is defined, it is easy to get their corresponding version for `LabeledGraph`:

```
Definition predicate_sub_labeledgraph (g: Graph)
  (p: V -> Prop) :=
      Build_LabeledGraph _ _ _ (predicate_subgraph g p)
      (vlabel g) (elabel g) (glabel g).
Definition predicate_partial_labeledgraph (g: Graph)
  (p: V -> Prop) :=
      Build_LabeledGraph _ _ _ (predicate_partialgraph g p)
      (vlabel g) (elabel g) (glabel g).
```

Update. So far our creating and reading operations are only about the plain structure of a graph, i.e., PreGraph. When considering the update of graphs, we can change the source or destination of edge and labels of graphs. When updating labels, we need to deal with LabeledGraph. So we have the following definitions.

```
Definition pregraph_gen_dst (g: Graph) (e: E) (t: V) :=
  @Build_PreGraph V E EV EE (vvalid g) (evalid g) (src g)
  (updateEdgeFunc (dst g) e t).
Definition update_vlabel (vlabel: V -> DV) (x: V) (d: DV) :=
  fun v => if equiv_dec x v then d else vlabel v.
Definition update_elabel (elabel: E -> DE) (e0: E) (d: DE) :=
  fun e => if equiv_dec e0 e then d else elabel e.
```

The function pregraph_gen_dst changes the destination of an edge e to t. The remaining two functions update the labels of a vertex and an edge respectively. Their definitions are almost the same as updateEdgeFunc, except for the types of parameters. Then we can define the functions which modify the labels of a LabeledGraph as follows. Now the notation Graph represents LabeledGraph. We also use the coercion mechanism of Coq to omit the conversion from LabeledGraph to PreGraph when necessary.

With the help of `update_vlabel` and `update_elabel`, the definitions are straightforward.

```
Notation Graph := (LabeledGraph V E DV DE DG).
Local Coercion pg_lg: LabeledGraph >-> PreGraph.
Definition labeledgraph_vgen (g: Graph) (x: V) (a: DV)
  : Graph := Build_LabeledGraph _ _ _ g
           (update_vlabel (vlabel g) x a)
           (elabel g) (glabel g).
Definition labeledgraph_egen (g: Graph) (e: E) (d: DE)
  : Graph := Build_LabeledGraph _ _ _ g (vlabel g)
           (update_elabel (elabel g) e d) (glabel g).
```

Delete. Similarly, we defined operations which remove a vertex or an edge from an existing graph. They are named `pregraph_remove_vertex` and `pregraph_remove_edge`, which are inverses of our adding operations. We omit their definitions here because they can be simply defined by replacing the function `addValidFunc` with `removeValidFunc` in their corresponding adding functions `pregraph_add_vertex` and `pregraph_add_edge`.

3.1.5 Predicates specifying graphs

Recall that in our definition of `GeneralGraph` at §3.1.1, there is a bound predicate `P` called the soundness condition to describe the unique property of the graph involved for each program under investigation. In principle we can always use the ingredients of the graph theory library to compose the soundness condition. Moreover, we distilled several representative features of a graph as independent soundness conditions. They can be composed flexibly into one large soundness condition in concrete applications. In other words, typical predicates can be shared among verification of different programs. So that we can reuse the supporting theorems about these predicates to greatly save effort and time.

A very intuitive predicate about graphs are about their finiteness. Theoretically, there is no limitation on the vertex or edge type of `PreGraph`. These types can have infinite elements, which could lead to a graph with infinite size[2]. But the graphs handled by real programs are always finite because they are stored in some physical storage devices and the capacities of such devices are finite. It is nearly an implicit property of all graphs in graph-manipulating programs. Sometimes this property is contained in some other stronger properties so that the verification does not need the finiteness explicitly. But sometimes we found that it is crucial to have it in reasoning. We define it as follows:

```
Definition Enumerable U (A: Ensemble U) :=
          {l: list U | NoDup l /\ forall x, In x l <-> A x}.
Class FiniteGraph (pg: PreGraph V E) := {
  finiteV: Enumerable V (vvalid pg);
  finiteE: Enumerable E (evalid pg) }.
```

`Enumerable` defines a dependent sum type, which is composed by a list of type `U` and a proof saying that the list is non-duplicative and an element is in the list if and only if it satisfies `A`. Since a list only contains finite number of elements by definition in Coq, `Enumerable U A` means the elements of type `U` with property `A` are finite. In the definition of `FiniteGraph`, the predicate `A` is instantiated as the validity predicates about vertices and edges respectively. So `finiteV` and `finiteE` together mean that the valid vertices and valid edges in a graph `pg` are all finite. Sometimes, `FiniteGraph` is too strong a condition. We also defined a weakened predicate called `LocalFiniteGraph`:

[2]For example, (`V`=`nat`, `E`=`nat`, `vvalid` v=`True`, `evalid` e=`True`, `src` i=i, `dst` i=$i+1$) represents a graph with infinite vertices and edges.

```
Definition out_edges (pg: PreGraph V E) x: Ensemble E :=
           fun e => evalid pg e /\ src pg e = x.
Class LocalFiniteGraph (pg: PreGraph V E) := {
  local_enumerable: forall x, Enumerable E (out_edges pg x)}.
```

The predicate `out_edges pg x` specifies all edges from vertex `x` in graph `pg`. So the definition of `LocalFiniteGraph` means every vertex has a finite number of out edges. It is a weakened version because we proved that a graph satisfying `FiniteGraph` also satisfies `LocalFiniteGraph`.

Another common predicate of graphs comes from the structure with pointers in C-like programming languages. In a typical implementation of a linked list, each element is represented as a `struct` with a pointer pointing to the next, linked element. The pointer of the last element is a null pointer to indicate the end of the list. When one uses pointers to represent edges to other vertices in a graph implementation, the same situation would happen again: null pointers are needed to show the boundary of a graph. This property is abstracted as `MathGraph`.

```
Definition DecidablePred (A: Type): Type :=
           {P: A -> Prop & forall a, {P a} + {~ P a}}.
Class MathGraph (pg: PreGraph V E) (is_null: DecidablePred V)
     : Prop := {
  weak_valid: V -> Prop := fun p => is_null p \/ vvalid pg p;
  valid_graph: forall e, evalid pg e ->
             vvalid pg (src pg e) /\ weak_valid (dst pg e);
  valid_not_null: forall x, vvalid pg x ->
                                 is_null x -> False }.
```

Here `DecidablePred` is a wrapper of a predicate `P` by emphasizing that whether `P a` holds or not for arbitrary `a` is decidable. The parameter `is_null` is such a predicate to judge and state whether a vertex is null because we need its decidability in both programs and proofs. A graph satisfying MathGraph must satisfy two conditions. One is valid_graph, which says that for any valid edge, its source vertex must be valid and its destination vertex must be either valid or null (so-called weak_valid). The other is valid_not_null, which says that any valid vertex cannot be null. It captures the attribute of null pointers in practice.

We also defined an uncommon predicate BiGraph to describe one feature of graphs in our illustrative verification programs: each vertex has at most two out-edges. We call it binary graph, just like the binary tree.

```
Class BiGraph (pg: PreGraph V E)
              (left_out_edge right_out_edge: V -> E): Prop :=
{ bi_consist: forall x, vvalid pg x ->
                        left_out_edge x <> right_out_edge x;
  only_two_edges: forall x e, vvalid pg x ->
                  (src pg e = x /\ evalid pg e <->
           e = left_out_edge x \/ e = right_out_edge x) }.
```

Besides the parameter pg which refers to the graph, BiGraph has other two parameters representing functions that derive the left and right edges for a given vertex. The two requirements of BiGraph are quite simple. One is bi_consist, which says the for any valid vertex, its left edge and right edge are different. The other is only_two_edges. As the name indicates, it says that for any valid vertex x and any edge e, the source of e is x and e is valid, if and only if e is one of the two designated edges of e.

Each of the three predicates defined before can serve as the soundness condition of a GeneralGraph. More importantly, we can combine the three predicates and other not-yet-defined predicates together to build a larger predicate to precisely describe the graph needed in a concrete verification. For example, the soundness condition used in a graph-marking program introduced in §5 is defined as follows:

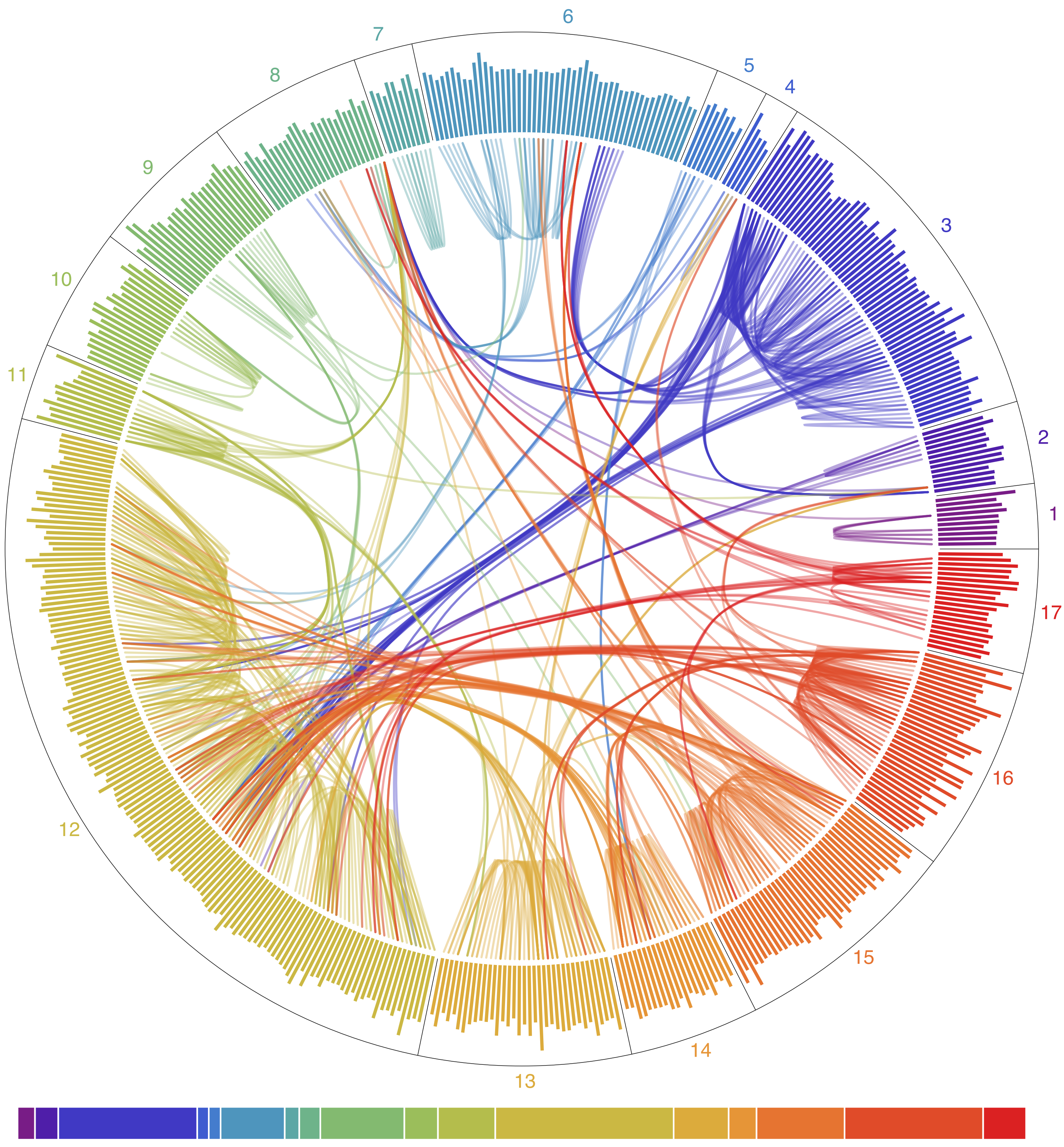
```
Inductive LR := | L | R.
Class BiMaFin (g: PreGraph addr (addr * LR)) := {
  bi: BiGraph g (fun x => (x, L)) (fun x => (x, R));
  ma: MathGraph g is_null;
  fin: FiniteGraph g }.
```

In this definition, the edge type is product of the vertex type `addr` and an indicator type `LR` to imply left and right edges. We omit the definition of decidable predicate `is_null` because it is basically a function to judge whether the given pointer is the null pointer.

3.2 Theorems in the Graph Library

The entities, relations, operations, and predicates shown in §3.1 are a small portion of the many concepts that are important and worth mentioning. Our library of graph theory contains more than 200 definitions of such concepts. They together construct the skeleton of the whole library. It is the *theorems* around those concepts that make the library valuable. There are more than 500 theorems which can be seen as the muscles of the library. These theorems are universal. They do not include the specialized lemmas which are only used in inferring facts about concrete programs.

§3.2.1 gives a global view of all theorems in the graph library. In §3.2.2, we put one theorem `path_shorten` under the microscope. Through the detailed exposure, we can see how complicated a formal proof could be, even for an intuitively straightforward theorem. §3.2.3 contains a brief overview of some other important theorems in the library.



■ 1: graph/BiGraph.v
■ 2: graph/FiniteGraph.v
■ 3: graph/GraphAsList.v
■ 4: graph/MathGraph.v
■ 5: graph/dag.v
■ 6: graph/graph_gen.v
■ 7: graph/graph_isomorphism.v
■ 8: graph/graph_model.v
■ 9: graph/graph_morphism.v
■ 10: graph/graph_relation.v
■ 11: graph/list_model.v
■ 12: graph/path_lemmas.v
■ 13: graph/reachable_computable.v
■ 14: graph/reachable_ind.v
■ 15: graph/subgraph2.v
■ 16: graph/tree_model.v
■ 17: graph/weak_mark_lemmas.v

Figure 3.7: Overview of the Theorems in the Graph Library

3.2.1 Overview of the theorems

To some extent, the theorems in our graph library are unstructured since the various concepts involved in those theorems are organized in a flattened way. But we can still grasp the big picture through their dependency. Figure 3.7 on page 61 gives an overview of the theorems of the whole graph library. All 522 theorems spread in 17 files are represented as line segments surrounding a circle. The color of a line segment indicates the file in which the theorem lives. The length of a line segment implies the size, counted in characters and represented in log scale, of a theorem with its proof. The curves connecting the line segments reveal the dependency among the theorems. If a theorem A needs another theorem B in its proof, there is a curve connecting A and B, where the color of the curve is the same as the color of A, the caller. So if there are many curves with many different colors connecting to a certain sector, it means that the theorems in that file are widely used as the foundation of other theorems. The file graph/graph_model.v in the lower left part of the circle is such an example. The files are not arranged chronologically, but alphabetically by their names, because theorems in the same file are not proved in the same time. The bar under the circle indicates the percentages of sizes of the files. Note that the percentages may not be the same as those presented in the circle because the circle is divided by the number of theorems in each file.

A lot of information can be read from Figure 3.7. For example, there is a significant concentration of curves among the line segments with the same color, which means that theorems in the same file are closely related. This is totally expectable because we usually prove theorems around the similar concepts in the same file. Sector 12 has the most theorems, which belong to graph/path_lemmas.v. From the colorful curves nearby we know that these path-related theorems are widely used. By contrast, the curves connecting graph/GraphAsList.v in sector 3 all have the same color as the theorems. This means the theorems in GraphAsList.v are not supporting lemmas for other theorems in this library. We can also notice that some theorems are not connected to any other theorems. This means they do not depend on other theorems and are not required by any other theorems. But it does not mean that they are useless. They are just not used by theorems listed here. They might be called in other theorems outside the graph library.

Each file or sector in Figure 3.7 is composed of the definitions of concepts and related theorems. The three files BiGraph.v, FiniteGraph.v and MathGraph.v (sectors 1, 2 and 4) contain the three predicates of graph `BiGraph`, `FiniteGraph`, `MathGrah` mentioned in §3.1.5 respectively. The two files GraphAsList.v and list_model.v (sectors 3 and 11) contain special treatment of list-shaped `PreGraph`s. In GraphAsList.v we mainly defined conversion functions between a list of vertex-and-edge pairs and a list-shaped `PreGraph`. We proved the two functions are inverse functions of each other. In list_model.v we defined the isomorphism between a list and a list-shaped `PreGraph` and proved for any list-shaped graph we can always find a list which is isomorphic to the graph. Analogously we have special treatment for tree-shaped graph in tree_model.v (sector 16) which provides the isomorphism between a tree and a tree-shaped graph and related theorems. A special kind of `PreGraph`—the directed acyclic graph—is specified through the predicate `Dag` in dag.v (sector 5). The file graph_gen.v (sector 6) contains functions of graph mentioned in §3.1.4. Graph isomorphism mentioned in §3.1.3 is defined in graph_isomorphism.v (sector 7). The basic definitions such as `PreGraph`, `LabeledGraph` and `GeneralGraph` are defined in graph_model.v (sector 8). The file graph_morphism.v (sector 9) contains definitions of partial and guarded isomorphisms between graphs. Relations other than isomorphism are defined in graph_relation.v (sector 10). The file path_lemmas.v (sector 12) contains definitions of path and reachability mentioned in §3.1.2. The file reachable_computable.v (sector 13) mainly proved a useful theorem: if a `PreGraph` is both a `MathGraph` and a `FiniteGraph`, then for any two vertices a and b, there is a decision procedure to judge whether b is reachable from a. We will discuss this theorem briefly in §3.2.3. The file reachable_ind.v (sector 14) is a helper library which provides some relations and lemmas to help proving reachability-related theorems through induction more easily. The file subgraph2.v (sector 15) contains predicates like `predicate_subgraph` discussed in §3.1.4. The last file, weak_mark_lemmas.v (sector 17) defined an abstract property named "mark" to represent the common marking operation in various algorithms and proved several theorems about it. We can see its application in §5. The 17 files together form our library of formal graph theory.

Even though there is no hierarchical organization for the theorems in our graph library, we can still categorize them into two layers according to whether or not they rely on other theorems. The theorems in the lower layer do not depend on others. They are basic facts derived directly from the concepts defined in the library. Usually they are self-evident: we just need to unfold the definitions, and then it is natural to get the conclusion we want for those basic facts. For example, the following theorem is a basic fact about the function `pregraph_add_vertex`:

```
Lemma addVertex_add_vvalid: forall (g: PreGraph V E) (v: V),
      vvalid (pregraph_add_vertex g v) v.
```

It says that for any vertex `v`, it is valid in the resulting graph if it is added to a graph. This conclusion comes from the definition of `addValidFunc` used in `pregraph_add_vertex`: `fun n => validFunc n \/ n = v`. The right side of the disjunction becomes `v = v` when it is applied to `v`.

Another example is about the function `pfoot` on `path`:

```
Lemma pfoot_head_irrel: forall l (g: PreGraph V E) v1 v2 n,
        pfoot g (v1, n :: l) = pfoot g (v2, n :: l).
```

It says that for any path containing at least one edge, the ending vertex is irrelevant to the starting vertex, which also comes from the definition of `pfoot`. It is proved by structural induction on `l`.

For any concept C in the library, such basic facts about C can be seen as the inherent properties of C. When there are enough proved facts surrounding a concept C, they actually form a "theorem interface" of C so that most proofs of later theorems involving C do not need to drill down to the definition of C. In a large formal library containing hundreds of concepts, such "interfaces" are absolutely necessary because they keep the complexity of proofs manageable. In our development of the graph library, we intended to prove as many basic facts about a certain concept as possible, which greatly reduces the burden of proving theorems in the upper layer.

In contrast to the theorems in lower layer, if a theorem is the collaboration of several complicated concepts, or if it cannot be proved instantly from the definitions of the concepts involved, it is categorized as the theorem in the upper layer. The proofs of the upper layer theorems adopt the lower layer theorems. They are usually more difficult and interesting to prove. In the remainder of the section, we will discuss some influential theorems in the upper layer according to the concepts involved.

3.2.2 Case study: lemma `path_shorten`

Here we explore a very interesting theorem thoroughly about the concept `path`: `path_shorten`. Through this sample we will give a taste of the theorems and proofs in the graph library. We will also show that the formal proof of a theorem could be rather complicated even when the idea of the proof is simple and intuitive.

The theorem `path_shorten` says that if there is a valid path connecting `n1` to `n2` with duplicated vertices, we can always find a strictly shorter and valid sub-path of the original path, which is still from `n1` to `n2`. It is formulated as follows:

```
Lemma path_shorten: forall (g: PreGraph V E) (p: path) n1 n2,
    path_ends g p n1 n2 -> valid_path g p ->
    Dup (epath_to_vpath g p) ->
    exists p', length (snd p') < length (snd p) /\
                 Subpath g p' p /\
                 path_ends g p' n1 n2 /\ valid_path g p'.
```

Recall that a `path` is defined as a pair consisting of a starting vertex and an edge list. That is why we use `length (snd p)` to express the length of a path. The definition is also the reason to define a function `epath_to_vpath` to retrieve the vertices along a path.

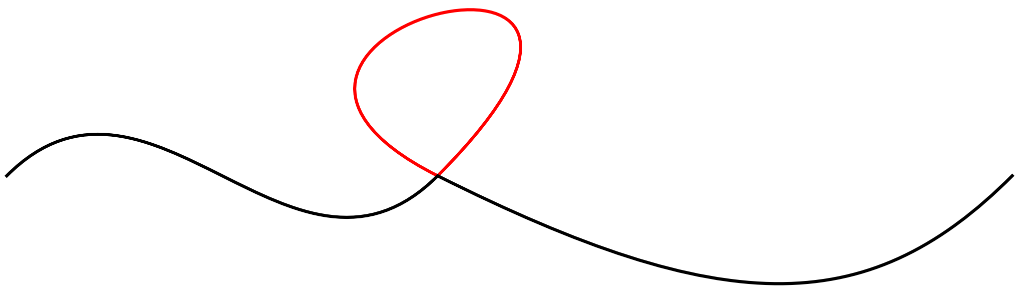


Figure 3.8: Intuition Behind the Lemma `path_shorten`

As illustrated in Figure 3.8, the lemma `path_shorten` is quite understandable. We can cut the “red” loop from the whole path. The remaining black path is still valid and it connects the original starting and ending vertex.

The informal proof of this lemma is also short and intuitive. We first notice that the vertices of the path `p` can possibly be split into three segments separated by the two duplicated vertices.

$$a_1, a_2, \ldots, a_m, d, b_1, b_2, \ldots, b_n, d, c_1, c_2, \ldots, c_l$$

where a_i, b_i, c_i and d are all vertices which are connected by consecutive edges. The numbers m, n and l could be zero. Then we can see that d, b_1, b_2, $\ldots$, b_n, d form a loop which can be cut from the `p`. The remaining vertices:

$$a_1, a_2, \ldots, a_m, d, c_1, c_2, \ldots, c_l$$

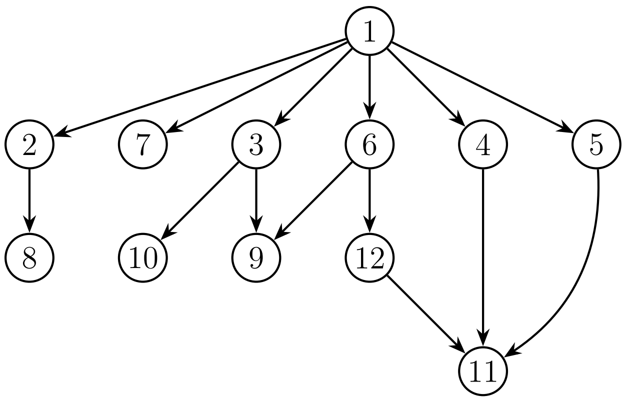
still form a shortened path without changing the starting and ending vertices. This is the path `p'` required in the conclusion.

Unlike the informal proof, the formal proof written in Coq is much longer, even though the proof idea is the same as the informal one, but every step needs to be justified via rigorous inference. The more concepts are involved, the more supporting theorems are needed. As we can tell, the statement of the `path_shorten` lemma includes two functions which are `length` and `epath_to_vpath`, and four predicates: `path_ends`, `valid_path`, `Dup` and `Subpath`. The proof of this theorem depends on 11 other theorems, directly or indirectly. The dependency among the 12 theorems is shown in Figure 3.9.

In the formal proof, the first step relies on the following helper lemma:

```
Lemma Dup_cyclic: forall l: list A,
      Dup l -> exists (a: A) (l1 l2 l3: list A),
                l = l1 ++ (a :: l2) ++ a :: l3.
```

It says exactly that a list `l` of duplicated elements can be split in to three segments `l1`, `l2` and `l3` joined by `a`. We do structural induction on `l` to prove this lemma. This helper lemma also depends on another basic fact about `Dup`:



1: `path_shorten`
2: `Dup_cyclic`
3: `epath_to_vpath_split`
4: `epath_to_vpath_pfoot`
5: `pfoot_app_cons`
6: `valid_path_merge`
7: `epath_to_vpath_phead`
8: `Dup_unfold`
9: `valid_path_cons`
10: `epath_to_vpath_cons`
11: `pfoot_head_irrel`
12: `paths_meet_cons`

Figure 3.9: Theorems On Which `path_shorten` Depends

```
Lemma Dup_unfold: forall {A} (a: A) (l: list A),
      Dup (a :: l) -> In a l \/ Dup l.
```

The lemma `Dup_unfold` says that if a list with the head `a` is duplicated, then either `a` is also in the rest of the list `l` or `l` has duplicated elements.

The next step is the cutting-vertices procedure. In the informal proof this step blurs the distinction between a list of vertices and a path. In the formal proof it needs another lemma twice to construct the sub-path from the lists of vertices:

```
Lemma epath_to_vpath_split: forall (g : PreGraph V E) (n : V)
      (l1 l2 : list V) (p : path), valid_path g p ->
       epath_to_vpath g p = l1 ++ n :: l2 ->
       exists p1 p2 : path,
         p = p1 +++ p2 /\ valid_path g p1 /\
         valid_path g p2 /\ epath_to_vpath g p1 = l1 +:: n /\
         epath_to_vpath g p2 = n :: l2.
```

This lemma says that if there is a vertex `n` in a path, we can always split the path into two paths, one ending at `n` and the other starting at `n`. The notation `p1 +++ p2` represents the merge function of two paths. This lemma builds the connection between a path composed by edges and the vertices along the path. It is proved by natural induction on `l1`. From Figure 3.9 we can see the proof adopts two other lemmas `valid_path_cons` and `epath_to_vpath_cons` in the lower layer:

```
Lemma valid_path_cons: forall g v e p,
       valid_path g (v, e :: p) -> valid_path g (dst g e, p).
Lemma epath_to_vpath_cons: forall g v e p a l,
      epath_to_vpath g (v, e :: p) = a :: l ->
      epath_to_vpath g (dst g e, p) = l.
```

After applying `Dup_cyclic` we get the decomposition of the vertices of the path `p`: `epath_to_vpath g p = L1 ++ a :: L2 ++ a :: L3`. Then we apply `epath_to_vpath_split` to decompose `p` into two paths `p1` and `p2` satisfying `p = p1 +++ p2` and other properties:

```
epath_to_vpath g p1 = L1 +:: a
epath_to_vpath g p2 = a :: L2 ++ a :: L3
```

Then we can apply `epath_to_vpath_split` again to decompose `p2` into two paths `p3` and `p4` satisfying `p2 = p3 +++ p4` and two more properties:

```
epath_to_vpath g p3 = a :: L2 +:: a
epath_to_vpath g p4 = a :: L3
```

Now we merge `p1` and `p4`, the new path `p1+++p4` is the `p'` required by `path_shorten`. So far it is just half of the formal proof. The remaining proof obligation is to prove that `p'` satisfies all four conjunctions to complete `path_shorten`. These steps rely on some other helper lemmas listed in Figure 3.9. For example, we need `valid_path_merge` to prove the constructed path by merging two paths is valid. In the interest of brevity we omit the details of these steps.

In our graph library, the lemma `path_shorten` is quite typical, i.e. it is neither the most complicated nor the simplest. The 11 supporting theorems involved were not all proved during the proving of `path_shorten`. Most of them were developed naturally and independently before, when proving other lemmas that were obliquely related. When we actually started writing the proof of `path_shorten`, we found many supporting theorems were ready. This was thanks to our general library, where we emphasize general lemmas over specialized facts, and thanks to our careful organization. So the intellectual effort of this proof was much lower than proving 12 theorems all at once.

The lemma `path_shorten` is not only a caller but also a callee. From the statement we know that it cuts at least one loop of a path with duplicated vertices. A typical valid path may contain multiple loops. We proved the following lemma, which calls `path_shorten` to guarantee that if there is a path from `n1` to `n2`, there always exists a path without any loops connecting `n1` to `n2`:

```
Lemma valid_path_acyclic:
   forall (g: PreGraph V E) (p: path) n1 n2,
   path_ends g p n1 n2 -> valid_path g p ->
   exists p', Subpath g p' p /\ path_ends g p' n1 n2 /\
              NoDup (epath_to_vpath g p') /\ valid_path g p'.
```

This lemma is proved by induction on the length of the path `p`. When the length is zero, the path contains only one vertex so that the lemma is trivially true. In the induction step, the induction hypothesis is that, for any path whose length is less than or equal to n, there is a path without loops. Now the path `p` has length $n+1$. If `p` contains no loops, then `p'` is `p`. Otherwise, we apply the lemma `path_shorten` to get a strictly shorter path `p1` from `p`. Then we can use the induction hypothesis to get the required `p'` from `p1`. This lemma is useful in dealing with properties relating to reachability.

3.2.3 Other influential theorems

In the total 525 theorems proved in our graph library, 180 of them are basic facts in the lower layer. They do not cite any other theorems. Even in the remaining 345 theorems, most of them rely on only one or two other theorems. Figure 3.10 gives the distribution of the 345 theorems according to the number of theorems they depend on in the graph library. Theorems in the standard library of Coq are not counted as dependencies.

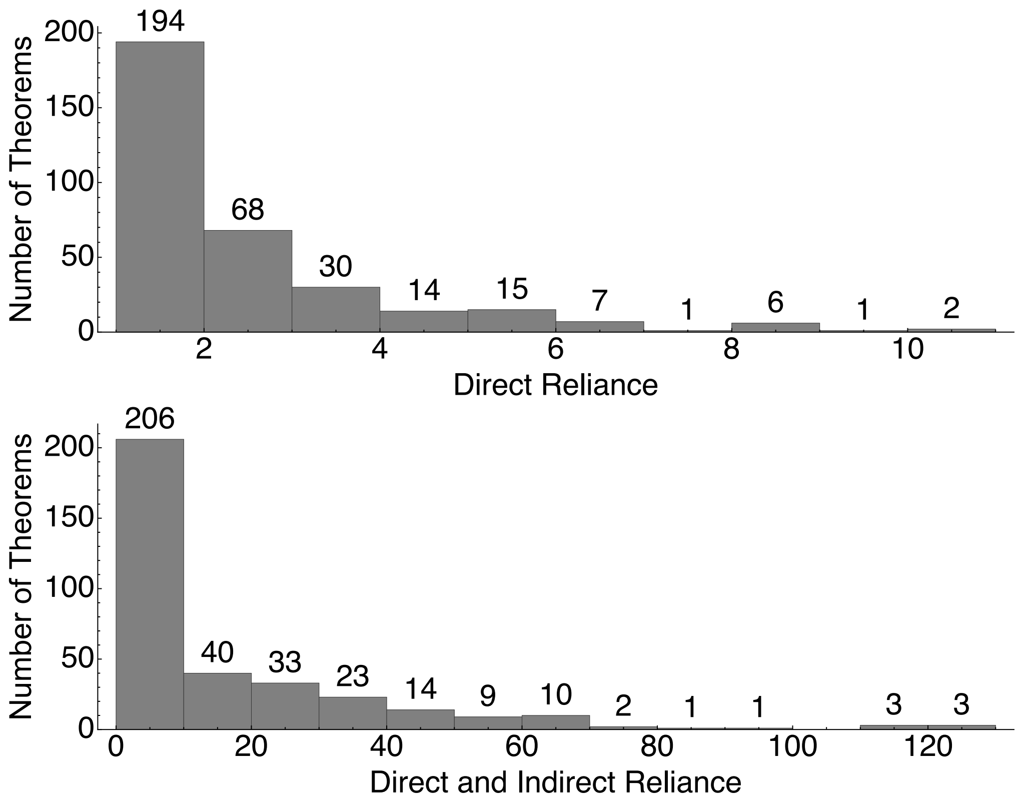


Figure 3.10: Theorem Dependencies in the Math Graph Library

There are two histograms in Figure 3.10. The top one is based on the number of directly supporting theorems and the other is based on all supporting theorems, directly and indirectly. For example, in Figure 3.9, the neighbors of node 1, which are nodes 2, 3, 4, 5, 6 and 7, represent the 6 directly supporting theorems of `path_shorten`. Other nodes except node 1 itself are indirectly supporting ones because they are not cited in the proof of `path_shorten` explicitly.

In general, the more theorems a proof cites, the more complicated the proof is. From both histograms we can conclude that nearly 70% of the 525 theorems in the graph library are relatively simple: they are either atomic or only depend on one or two theorems. For the other theorems we think counting both directly- and indirectly-supporting theorems is a more accurate measurement of their complexity because it reflects the accumulated effort to achieve the conclusion of a theorem.

It is worth mentioning two interesting theorems with certain complexity in the graph library. The first one is `reachable_by_idempotent` in subgraph2.v.

```
Lemma reachable_by_idempotent:
      forall (g: PreGraph V E) (P: V -> Prop) (n: V),
      Same_set (reachable_by g n P)
               (reachable_by g n (reachable_by g n P)).
```

The relation `Same_set P Q` is just a shortcut of `forall x, P x <-> Q x`. So the theorem says that for any vertex `n` in a graph `g` and any vertex property `P`, the following two statements are equivalent:

1. A vertex `v` is reachable from `n` and each vertex in the reaching path from `n` to `v` satisfies `P`.

2. A vertex `v` is reachable from `n` and any vertex `m` in the reaching path from `n` to `v` is also reachable from `n`, and each vertex in the reaching path from `n` to `m` satisfies `P`.

This theorem looks obviously true, but its formal proof is surprisingly complicated. Figure 3.11 gives a dependency graph of this theorem. Each vertex represents a theorem and each directed edge from a

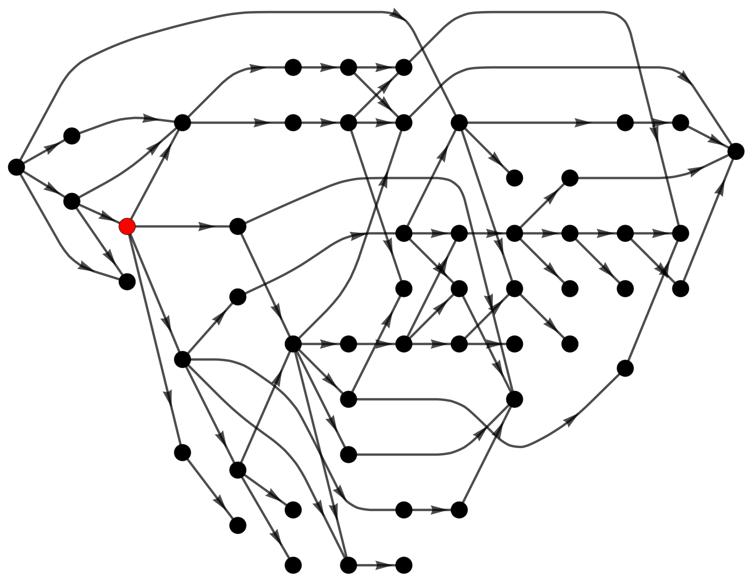


Figure 3.11: Dependency Graph of `reachable_by_idempotent`

to b means that theorem a depends on theorem b. The leftmost node is `reachable_by_idempotent`. There are 52 vertices in total. From the figure, it only cites 4 theorems directly. The largest number of dependencies comes via the red node, which represents the following theorem:

```
Lemma reachable_partialgraph_reachable_equiv:
      forall (g: PreGraph V E) (P: V -> Prop) (n: V),
      Included (reachable g n) P ->
      Same_set (reachable g n)
               (reachable (predicate_partialgraph g P) n).
```

It says that if `reachable g n` can be inferred from the property `P`, then the property "reachable from `n` in graph `g`" is the same as "reachable from `n` in a partial graph which only contains nodes satisfying `P`".

The other interesting theorem is `reachable_by_decidable`:

```
Lemma reachable_by_decidable: forall (G: PreGraph V E)
      {is_null: DecidablePred V} {MA: MathGraph G is_null}
      {LF: LocalFiniteGraph G} (p : DecidablePred V) x,
      {vvalid G x} + {~ vvalid G x} ->
      EnumCovered V (reachable G x) -> ReachDecidable G x p.
```

The predicates involved in this theorem is listed as follows:

```
Definition EnumCovered U (A: Ensemble U) :=
           {l: list U | NoDup l /\ forall x, A x -> In x l}.
Definition ReachDecidable (g: PreGraph V E)
           (x : V) (P : V -> Prop) :=
   forall y, Decidable (g |= x ~o~> y satisfying P).
```

Here `EnumCovered` is a weakened version of `Enumerable`: the list provided in `EnumCovered` contains all `x` satisfying `A` but it may contain other extraneous elements. So the only information it provides is that the number of elements satisfying `A` is finite. The proposition `ReachDecidable gx P` means there is a decision procedure to judge whether any vertex `y` is reachable from `x` in `g` along a path in which each vertex satisfies `P`. The whole theorem `reachable_by_decidable` says that in a `PreGraph G` satisfying both `MathGraph` and `LocalFiniteGraph`, for any vertex `x`, if the vertices reachable from `x` are finite, then we can judge whether a certain vertex is reachable from `x`.

The proof of this theorem is highly nontrivial but the idea is still simple. Since `G` is `LocalFiniteGraph`, we can always find all neighbors of a certain vertex `v`. Since `G` is `MathGraph`, we can always remove the null vertices from those neighbors. The remaining vertices are all valid. According the two clues here, we wrote a Breadth-First-Search program to find all vertices reachable from `x`. Since there are finitely many reachable nodes, this program must terminate. We proved that the program is correctly implemented. So the decision procedure just compares the vertex with all reachable vertices found by the BFS program.

Figure 3.12 gives a dependency graph of this theorem. The topmost

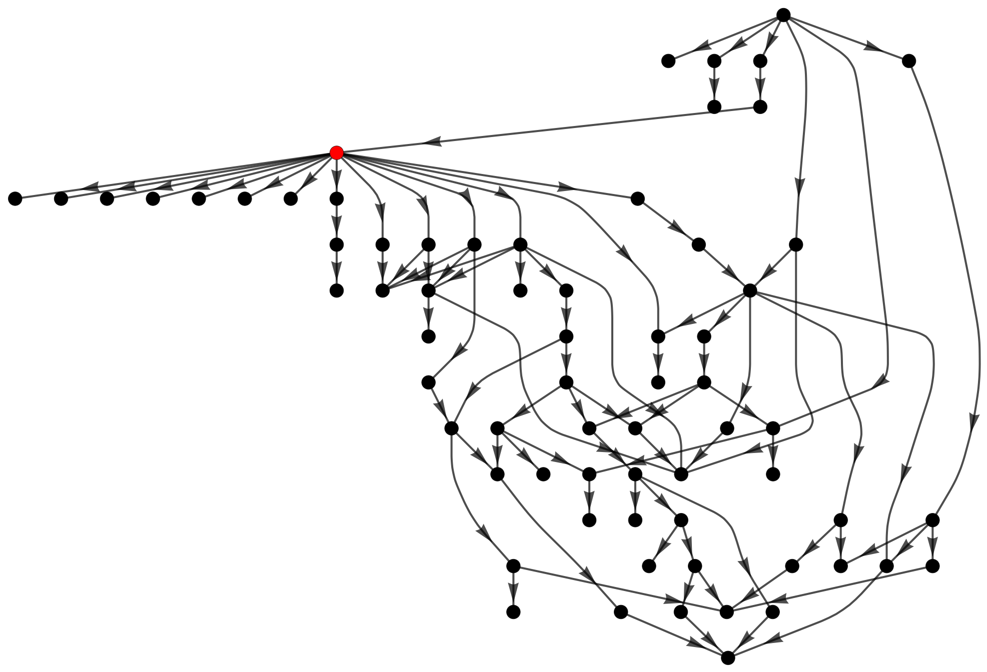


Figure 3.12: Dependency Graph of `reachable_by_decidable`

node represents `reachable_by_decidable`. It contains 67 supporting theorems beneath it. The red node represents a key theorem in proving it. It is named `finite_reachable_computable'`:

```
Lemma finite_reachable_computable': forall (G: PreGraph V E)
      {is_null: DecidablePred V} {MA: MathGraph G is_null}
      {LF: LocalFiniteGraph G} x
      (X: EnumCovered V (reachable G x)) l', vvalid G x ->
      l' = construct_reachable
                (length (proj1_sig X), x :: nil, @nil V) ->
      reachable_list G x l' /\ NoDup l'.
```

Here the function `construct_reachable` is the BFS program. We use the technique called well-founded recursion discussed in §2.3 to finish its definition. The predicate `reachable_list` means the list `l'` contains exactly all the vertices reachable from `x`, and nothing more. This theorem basically says that the result `l'` produced by the function is the reachable set from vertex `x`. It is indeed a complicated proof because the number of directly supporting theorems reaches 14 already in Figure 3.12.

3.3 Related Work

Other people have also tried to develop mechanized graph libraries, and here we provide a literature review.

The most famous graph related theorem which has been mechanically verified is the four-color theorem: Any planar map can be colored with only four colors. In 2004, Georges Gonthier formalized a proof of the theorem [Gonthier, 2008] inside Coq. It is very easy and natural to rephrase the problem in graph theory: by taking regions as nodes and connecting each pair of adjacent regions as edges, coloring the map is equivalent to coloring the graph obtained. However, they used a different kind of combinatorial structure, known as hypermaps, instead of graphs. Basically, a hypermap is a type "dart" with several functions mapping dart to dart. The combinatorial and geometrical properties are encoded as certain permutation properties of those functions. It is a substantially different structure from graph. So their formal library provides little inspiration.

Lars Noschinski built a formalized graph library for the Isabelle/HOL proof assistant and verified a method of checking Kuratowski subgraphs used in the LEDA library. It supports general infinite directed graphs with labeled and parallel arcs [Noschinski, 2015a]. His definition of graph is similar to our PreGraph except he uses vertex/edge set instead of validity functions. Besides, Noschinski's library also covers basic graph related concepts such as reachable component and spanning tree.

Benedikt Nordhoff and Peter Lammich formalized and proved Dijkstra's algorithm in Isabelle [Nordhoff and Lammich, 2012]. Their graph is defined as vertex and edge sets where the edge is a triple (source, label, destination).

They only defined what they needed for the algorithm, and so their work does not lend itself to reuse.

Working in HOL, Wai Wong expressed a small portion of the conventional graph theory, which is mainly used to model the railway track network and applied in signaling systems [Wong, 1991]. It does not contain many graph property-related theorems.

Ching-Tsun Chou formalized theory of undirected graphs in HOL that emphasizes the notion and important properties of trees [Chou, 1994]. He applied this library to verify a distributed algorithm named PIF (Propagation of Information with Feedback) [Chou, 1995].

Jean Duprat formalized graph in an inductive way in Coq [Duprat, 2001]. Only some basic properties are proved in it. To our knowledge, no application is built on it.

In the work of Mitsuharu Yamamoto et al., a formalization of planar graph is inductively defined in HOL [Yamamoto et al., 1995]. They use it to prove Euler's formula as an application. Tetsuo Tamai tackled the same problem [Tamai, 2000]. But his purpose was just giving a formal specification in CafeOBJ, so this graph library only contains formal definitions.

In 1998, Mitsuharu Yamamoto et al. formalized directed graph based on Wong's work [Yamamoto et al., 1998]. They proved the correctness of the abstract A* algorithm based on graph and semi-lattice.

NASA's graph theory library is written in PVS (Prototype Verification System) [Butler and Sjogren, 1998]. It is restricted to finite graphs only and does not support multi-edge graphs. They use the library to prove Ramsey's Theorem and Menger's Theorem.

Gertrud Bauer and Tobias Nipkow inherited the inductive approach of Mitsuharu Yamamoto et al. for the formalization of planar graph theory. They formally proved the five-color theorem using graph theory and triangulations [Bauer and Nipkow, 2002].

In the work of Tobias Nipkow et al., a finite, undirected, planar graph is formalized as a list of faces and faces as lists of vertices [Nipkow et al., 2006]. That library is mainly used to prove the completeness of the enumeration of tame graphs. This is the first step of Thomas Hales' proof of Kepler Conjecture [Hales et al., 2017].

Tom Ridge also mechanized graphs and trees in Isabelle/HOL [Ridge, 2005]. It is close to Wong's work [Wong, 1991] with the following difference: the edges are represented as sets of vertices instead of atomic objects.

In 2015, Lars Noschinski presented a graph library in Isabelle/HOL to reason about graphs and implemented a verified decision procedure for combinatorial planarity of graphs [Noschinski, 2015b]. In this thesis, he also verified checkers for both the planarity and the non-planarity certificates emitted by the LEDA library. Both the implementation and verification of the checker are written in the abstract language of AutoCorres.

In 2015, Dubois et al. [2015] proposed a formalization of graphs without multiple edges. A formally verified auditor was developed to certify the result of a function that calculates a maximum cardinality matching. The executable code of the auditor is extracted from Coq directly.

Sergey et al. [2015] proposed a formal framework which is also embedded in Coq for mechanized verification of full functional correctness of fine-grained concurrent programs through Hoare-style reasoning. To deal with the possible interactions between concurrent threads on shared resources, they employed a state-transition system called concurroid as the base of their specification model. In one of the examples in the paper, they also defined a spatial graph as a predicate about the shared heap part of a concurroid. A graph is described as a bunch of pointers which are connected through pointing to each other. This definition, which does not rely on a particular mathematical graph, has advantages in reasoning but also loses some expressiveness.

Chapter 4

Spatial Representation and Inference

The formalization of mathematical graph theory is just the beginning of the story. Our goal, which sets us apart from many other purely-formal graph libraries discussed in §3.3, is to verify real graph-manipulating programs. So the ability to represent a graph in the memory model is a natural requirement. We discuss our effort and final decision in finding a proper representation in §4.1. In §4.2 we show our LOCALIZE rule, an improvement of the RAMIFY rule introduced in §2.1.3. It is a very powerful inference rule to reason about the global consequences of local actions on data structures like graphs. We pack the concepts in constructing spatial representations of graphs, the theorems about this LOCALIZE rule, the infrastructure to adapt to the formal library of separation logic in MSL [Appel et al., 2014], and some common ramification theorems which are premises of the LOCALIZE rule together to form a library about spatial graphs. We discuss the concrete organization of this library in §4.3.

It should be noted that our spatial library does not include concrete settings or theorems for special kinds of graphs in certain programs. But if a concept or theorem is applicable for multiple programs, we will classify it to the spatial library to achieve good modularity.

4.1 Definitions of Spatial Graph Predicates

The function of the spatial representation of a graph is twofold. One is to precisely describe the data layout in the memory and the other is to build a bridge connecting the abstract mathematical graph with the concrete data structure in the memory. A program can be seen as a sequence of actions on data structures and our verification is a proof of certain propositions about the consequence of manipulations on mathematical graphs. There is a gap between actions on data structures and manipulations on graphs. The spatial representation fills this gap, and so its role is crucial. We use it to reflect manipulations on graphs from actions on data structures. In §4.1.1 we show that the most obvious recursive definition of spatial representations, inspired by classical examples in separation logic, is inadequate. In §4.1.2 we give a flat spatial representation of graphs used in all our verification projects.

4.1.1 Flaw of recursive definitions

It is not a fresh new idea at all to introduce mathematical entities into spatial representations. In two papers about separation logic [O'Hearn et al., 2001; O'Hearn, 2012], that both aim to give a more precise specification of a tree copy program, the spatial representation of a binary tree is defined recursively as a predicate with an additional parameter, the mathematical tree τ:

$$\begin{aligned}\mathsf{tree}(x,\tau) \overset{\text{def}}{=} {} & (x = 0 \wedge \mathsf{isatom}(\tau) \wedge \mathsf{emp}) \vee \\ & \Big(\exists\, l, r, \tau_1, \tau_2.\ \tau = \langle \tau_1, \tau_2 \rangle \wedge \\ & (x \mapsto l, r) * \mathsf{tree}(l, \tau_1) * \mathsf{tree}(r, \tau_2)\Big)\end{aligned} \tag{4.1}$$

Here $x \mapsto l, r$ means x points to two adjacent memory locations which hold values l and r. It can be seen as a shorthand of $x \mapsto l * (x+1) \mapsto r$. The pair l, r can be even extended to arbitrary tuples. In definition (4.1), the mathematical tree τ is defined as an expression to encode the shape of a binary tree. It is either an atom or a pair of tree encodings. For example, $\langle \mathsf{atom}, \langle\langle \mathsf{atom}, \mathsf{atom}\rangle, \mathsf{atom}\rangle\rangle$ is a valid encoding. We can see that τ describes the shape precisely. The other parts of (4.1) are self-evident. A $\mathsf{tree}(x, \tau)$ is either an emp or $(x \mapsto l, r) * \mathsf{tree}(l, \tau_1) * \mathsf{tree}(r, \tau_2)$ where τ_1 and τ_2 are the left and right subtrees of τ. If τ_1 is atom then a must be null pointer and $\mathsf{tree}(l, \tau_1)$ must be emp. Otherwise $\mathsf{tree}(l, \tau_1)$ can be further expanded like $\mathsf{tree}(x, \tau)$, and so on. Thus the data layout matches τ (the shape of a tree) exactly. In the verification of a tree copy program, the precondition is $\mathsf{tree}(x, \tau)$ and the postcondition is $\mathsf{tree}(x, \tau) * \mathsf{tree}(y, \tau)$. The same τ in the specification indicates that the program creates a exact clone of the original tree, not an arbitrary shaped tree.

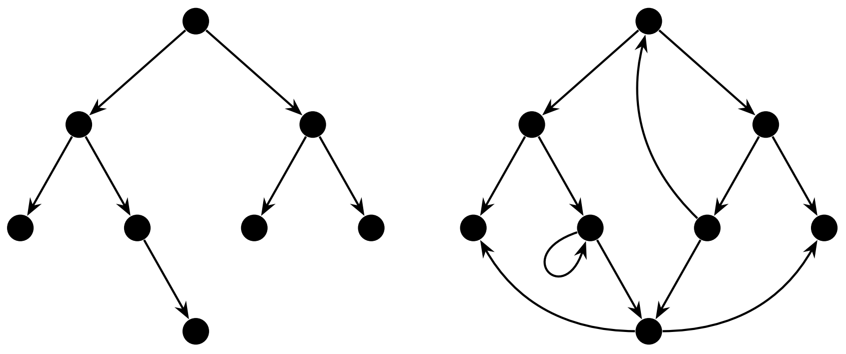


Figure 4.1: A Binary Tree and a Binary Graph

When it turns to graph, things become more complicated. If we restrict the number of out edges to at most two, we get a special kind of graph which can be called binary graph, just like binary tree. Figure 4.1 is an illustration of a binary tree on the left and a binary graph on the right. From the figure we can clearly figure out the key difference between a tree and a graph. For a binary tree, its left branch and right branch are disjoint.

So we can use $*$ to separate them in definition (4.1). But for a typical binary graph in Figure 4.1, its left branch and right branch are overlapped. So the separating conjunction is not applicable here if we want to define a similar recursive predicate. According to Reynolds [2003], Richard Bornat proposed a new connective $\uplus\!\!\ast$ called the "relevance conjunction" to address this situation. Hobor and Villard [2013] exploited this concept further. They named $\uplus\!\!\ast$ the overlapping conjunction whose formal definition is:

$$\begin{aligned} h \models P \uplus\!\!\ast\, Q \overset{\text{def}}{=} \exists\, h_1, h_2, h_3, h_{12}, h_{23} \text{ s.t. } h_1 \oplus h_2 = h_{12} \wedge \\ h_2 \oplus h_3 = h_{23} \wedge h_{12} \oplus h_3 = h \wedge h_{12} \models P \wedge h_{23} \models Q \end{aligned} \tag{4.2}$$

As illustrated in Figure 4.2, an assertion $P \uplus\!\!\ast\, Q$ is true of a heap if it can be split into three disjoint heaplets h_1, h_2 and h_3. The heap joined by first two heaplets h_1 and h_2 makes P true and the heap joined by last two heaplets h_2 and h_3 makes Q true. In other words, the heaps on which P and Q holds have an unspecified shared portion.

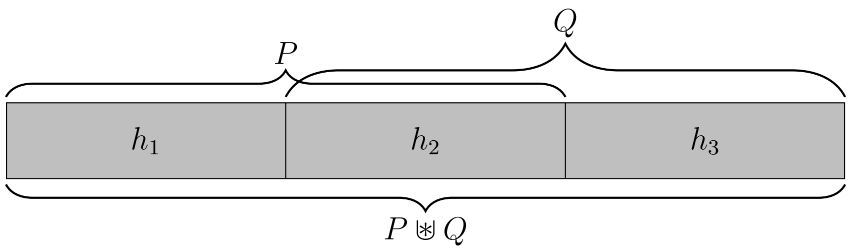


Figure 4.2: Overlapping Conjunction

With the help of overlapping conjunction $\uplus\!\!\ast$, it is possible to define the spatial representation of a graph recursively. Consider the following data structure written in C. It is used in a graph marking program:

```
struct Node {int m; struct Node* l; struct Node* r;};      (4.3)
```

It is the same as the standard binary tree structure. However, the pointers `l` and `r` can point to the same piece of memory. So this data structure can represent a binary graph. In fact, this binary graph also satisfies two additional graph predicates discussed in §3.1.5: `MathGraph` and `BiGraph`. It satisfies `BiGraph` because it has exactly two out edges and it satisfies `MathGraph` because the pointers `l` and `r` could be null. Hobor and Villard [2013] defined the following spatial predicate (a predicate on some part of the memory) for this data structure of a binary graph:

$$\begin{aligned}\mathsf{graph}(x,\gamma) \stackrel{\text{def}}{=} {} & (x = 0 \wedge \mathsf{emp}) \vee \\ & \Big(\exists m, l, r, \text{ s.t. } \gamma(x) = (m, l, r) \wedge \\ & x \mapsto m, l, r \uplus\!\!\!\!\ast\ \mathsf{graph}(l,\gamma) \uplus\!\!\!\!\ast\ \mathsf{graph}(r,\gamma)\Big)\end{aligned} \tag{4.4}$$

Here γ is a mathematical graph. Besides the distinction between $*$ and $\uplus\!\!\!\!\ast$, there are several other differences between definition (4.1) and (4.4). It is also overloaded to represent a function $\gamma(x)$, which provides information (vertex label: m, destinations of left edge: l, and destinations of right edge: r) for a vertex x in graph γ. Unlike the τ in (4.1), γ used in (4.4) contains not only the shape information but also the data field stored in each `Node`. Another difference is that the recursive part in definition (4.4) such as $\mathsf{graph}(l,\gamma)$ still has the whole γ as its parameter but $\mathsf{tree}(l,\tau_1)$ only uses the left branch τ_1. This is because a graph is a structure with intrinsic sharing. It is difficult and unnecessary to use the left and right parts of the graph as the parameters. The two parts provides no information: they may have common vertices and edges or even worse, they could be the same. It is more clear to use γ consistently. Because of this difference, the semantics of $\mathsf{tree}(x,\tau)$ and $\mathsf{graph}(x,\gamma)$ are different. The predicate $\mathsf{tree}(x,\tau)$ is a spatial representation of the whole tree τ but from the definition $\mathsf{graph}(x,\gamma)$ is just a spatial representation of the reachable part from vertex x of the graph γ.

Initially we construct a recursive predicate according to definition (4.4) through Tarski's fixed-point mechanism mentioned in §2.4.2. However, we

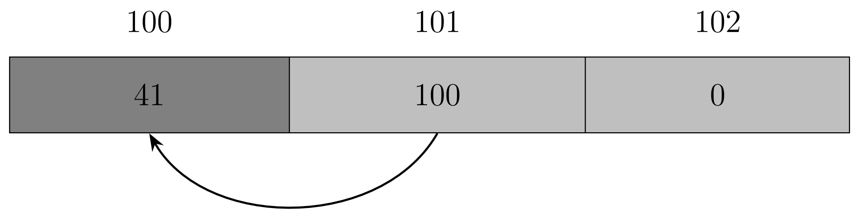


Figure 4.3: A Self-Pointing Graph

soon found that the result is hard to use. Consider the memory m which is composed of three heaplets illustrated in Figure 4.3. Clearly we have $m \models 100 \mapsto 41, 100, 0$. It is also clear that this memory represents a one-vertex cyclic graph, i.e. $\mathsf{graph}(100, \hat{\gamma})$ where $\hat{\gamma}(100) = (41, 100, 0)$. In other words, we should be able to prove $m \models \mathsf{graph}(100, \hat{\gamma})$ from the hypothesis $m \models 100 \mapsto 41, 100, 0$. We can use backward reasoning to explore this proof goal.

We will shortly be using so-called "backwards reasoning" and so will briefly digress to review it here. Ordinarily a forward proof starts from the hypotheses, performs some manipulations, and reaches the conclusion. But backward reasoning starts from the conclusion, collects what would imply the conclusion, attempts to make the goal look more like the hypotheses or some known results. For example, suppose the hypothesis is A, the conclusion is C and there are two known theorems: $A \rightarrow B$ and $B \leftrightarrow C$. In forward reasoning, we start from A, use $A \rightarrow B$ to get B first. Then we use $B \leftrightarrow C$ to get the conclusion C. While in backward reasoning, we start from C. Since we have $B \leftrightarrow C$, we can change the current proof goal to B. The reason is that if we can prove B, then immediately we can get C by using $B \leftrightarrow C$. This step can be seen as the backward application of $B \leftrightarrow C$. Now for the goal B, we can backwardly apply $A \rightarrow B$ to change the goal to A. A is the hypothesis, so the proof is complete.

In our case, the current goal is the conclusion $m \models \mathsf{graph}(100, \hat{\gamma})$. We can unfold $\textsf{graph}$ according to definition (4.4). The proof goal becomes

$$m \models (100 = 0 \wedge \textsf{emp}) \vee \Big(\exists m, l, r, \text{ s.t. } \hat{\gamma}(100) = (a, l, r) \wedge \\ 100 \mapsto a, l, r \uplus \textsf{graph}(l, \hat{\gamma}) \uplus \textsf{graph}(r, \hat{\gamma})\Big).$$

For goal like $m \models A \vee B$, either $m \models A$ or $m \models B$ could imply it. It is impossible to prove $100 = 0$, so the only choice is the right branch. We know that $\hat{\gamma}(100) = (41, 100, 0)$. So the existential variables a, l, r must be $41, 100, 0$. So the current proof goal becomes

$$m \models \hat{\gamma}(100) = (41, 100, 0) \wedge \\ 100 \mapsto 41, 100, 0 \uplus \textsf{graph}(100, \hat{\gamma}) \uplus \textsf{graph}(0, \hat{\gamma}).$$

For proof goal like $m \models A \wedge B$, we need to prove both $m \models A$ and $m \models B$ to imply it. In our case, we know that $\hat{\gamma}(100) = (41, 100, 0)$ and it is a pure proposition. So no matter what m is, $m \models \hat{\gamma}(100) = (41, 100, 0)$ is always true. For the other branch, according to definition (4.4) we know $\textsf{graph}(0, \hat{\gamma}) = \textsf{emp}$ and we proved $P \uplus \textsf{emp} = P$. So the proof goal becomes

$$m \models 100 \mapsto 41, 100, 0 \uplus \textsf{graph}(100, \hat{\gamma}).$$

Recall the definition of $\uplus$ in (4.2). We can unfold $\uplus$ to get the new proof goal as follows:

$$\exists\, h_1, h_2, h_3, h_{12}, h_{23} \text{ s.t. } h_1 \oplus h_2 = h_{12} \wedge h_2 \oplus h_3 = h_{23} \wedge h_{12} \oplus h_3 = h \wedge \\ h_{12} \models 100 \mapsto 41, 100, 0 \wedge h_{23} \models \textsf{graph}(100, \hat{\gamma}).$$

Obviously $100 \mapsto 41, 100, 0$ and $\textsf{graph}(100, \hat{\gamma})$ overlap completely. This means h_1 and h_3 are the null heap and h_2 is m. In such a situation h_{12} and h_{23} are both m and all joining relations with the form $p \oplus q = r$ are satisfied. The remaining current proof goal is

$$m \models 100 \mapsto 41, 100, 0 \wedge m \models \mathsf{graph}(100, \hat{\gamma}).$$

For goal like $A \wedge B$, we need to prove A and B respectively to imply it. $m \models 100 \mapsto 41, 100, 0$ is exactly the hypothesis. So it is done. The remaining current proof goal is

$$m \models \mathsf{graph}(100, \hat{\gamma}).$$

But this is the conclusion we wanted to prove at the very beginning! Each step of our backward reasoning is solid and looks reasonable. But we have taken a useless detour and hit a dead end. This is partially because the expansion of the recursive part does not interact well with $\uplus$. If the expansion is about $*$, we could get a strictly smaller sub-heap. Then an induction on the finite memory would prove what we want. Unfortunately the overlapping conjunction $\uplus$ does not have the "strictly smaller" property.

In fact we do not know whether it is possible to prove $m \models \mathsf{graph}(100, \hat{\gamma})$ from $m \models 100 \mapsto 41, 100, 0$. However, the difficulty encountered in proving such an "obvious" entailment suggests that it may not be a good idea to define spatial predicates for graphs recursively.

4.1.2 Flat representations of graphs

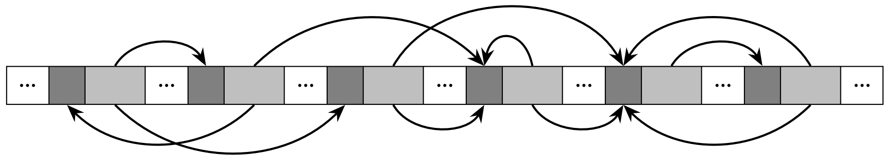


Figure 4.4: Graph Vertices in a Heap

Figure 4.4 illustrates how vertices of a binary graph could be distributed in a heap. Each vertex could be thought as a `struct Node` defined in (4.3). The dark gray parts are data fields and the lights gray parts are pointer fields. The arrows in the figure show how these vertices are connected. There are at most two arrows from each node. From the viewpoint of the heap, the graph is flattened as a series of its individual vertices.

Inspired by Figure 4.4 we could define the spatial representation of a graph as a list of vertex representations separated by $*$. This is not a recursive definition and it does not need $\uplus$ anymore. This idea is not original to us. Sergey et al. [2015] have defined graphs in separation logic in a similar way. The difference is that their definition does not contain a mathematical graph so its expressiveness is limited.

Before giving a formal definition of the spatial predicate of a graph, we need to define a more general concept: the iterated separating conjunction or "big star" over a list:

$$\mathop{\huge\ast}_{\{l_1,\ldots,l_n\}} P \overset{\text{def}}{=} P(l_1) * P(l_2) * \cdots * P(l_n). \tag{4.5}$$

In our spatial library, the big star is formalized and named `iter_sepcon`:

```
Fixpoint iter_sepcon (l: list B) (p: B -> A): A :=
  match l with
    | nil => emp
    | x :: xl => p x * iter_sepcon xl p
  end.
```

This is a normal recursive definition over the list `l`. Since the list always has a finite length, the expansion will always terminate. In the definition, `B` is an arbitrary type without any restriction to achieve the most generality. By contrast, type `A` must be an instance of two type classes `NatDed` and `SepLog` defined in MSL. The `NatDed` contains natural deduction laws and `SepLog` contains the notation of $*$, `emp`, and related laws in separation logic. In other words, `A` must be the type of spatial predicates in separation logic. The notation `*` used in `iter_sepcon` is exactly the familiar separating conjunction $*$ and the `emp` is exactly the empty-heap assertion emp.

We further extend the "big star" to arbitrary predicate S, not a list:

$$\mathop{\Huge *}_{S} P \overset{\text{def}}{=} \exists L\,.\,(\forall x\,.\,x \,\mathsf{in}\, L \leftrightarrow S(x)) \wedge (\mathsf{NoDup}\ L) \wedge \mathop{\Huge *}_{L} P \tag{4.6}$$

The formal definition in Coq is almost the same as (4.6):

```
Definition pred_sepcon (P: B -> Prop) (p: B -> A): A :=
  EX l: list B, !! (forall x, In x l <-> P x) &&
                !! NoDup l && iter_sepcon l p.
```

Here we need to explain the notations used in the definition above. Compared with (4.6), the notation `EX` should mean the existential quantifier $\exists$ and `&&` should mean the conjunction $\wedge$. They are different from the standard keyword `exists` and notation `/\` used in Coq. This is because, in the formalization of separation logic in MSL, separation logic is provided as a formal system (or, a logic) to seal the rather intricate underlying model. To distinguish this formal logic from the meta logic of Coq, MSL provides a set of new notations such as `EX`, `&&` and related laws, like "entails" $\vdash$, a high-level version of implication $\rightarrow$ defined in (2.4). In fact, the type of propositions in meta logic of Coq is `Prop` but the type of spatial predicates, `A`, can be thought of as $\mathsf{memory} \rightarrow$ `Prop`. In this sense the formalized separation logic is a high-order logic. The propositions of type `Prop` are called *pure*. In comparison, the propositions of type `A` in high-order separation logic are called *spatial*. If we want to present a pure proposition `P` in spatial propositions, we need to use another notation `!!` to "lift" it to type `A`, just like `(forall x, In x l <-> P x)` and `NoDup l` in the definition of `pred_sepcon`.

Now we are ready to give the spatial representation of a graph via a series of vertex representations. The vertex information function $\gamma(x)$ appeared in definition (4.4) is quite useful here. For a vertex v in graph γ, its vertex representation could simply be $v \mapsto \gamma(v)$ if v is also a memory location. Roughly speaking, we can define the spatial predicate for a binary graph starting from x as follows:

$$\mathsf{graph}(x, \gamma) \overset{\text{def}}{=} \mathop{\Huge *}_{\mathtt{reachable}\,\gamma\,x\,v} v \mapsto \gamma(v) \tag{4.7}$$

where `reachable` is the predicate about reachability defined in Figure 3.3. Recall that $\mathtt{reachable}\,\gamma\,x\,y$ means vertex y is reachable from x in graph γ. So $\mathsf{graph}(x, \gamma)$ represents the reachable part of γ from x in memory. Furthermore, the graph γ here is not necessary a binary graph. For graphs with unspecified number of out edges, we can define other information functions $\gamma(x)$ accordingly. Thus the spatial predicate $\mathsf{graph}(x, \gamma)$ defined in (4.7) can be a generic definition of spatial representation of graphs which have a "starting point".

Our definition of $\mathsf{graph}(x, \gamma)$ is flat, in that we cannot follow the link structure recursively via any obvious way. However, we successfully proved a general recursive fold/unfold theorem when $x \mapsto \gamma(x)$ and the mathematical graph γ have certain necessary properties:

$$\mathsf{graph}(x, \gamma) \dashv\vdash x \mapsto \gamma(x) \uplus\!\!\!* \left(\mathop{\biguplus\!\!\!\!*}_{n \in \mathsf{neighbors}(\gamma, x)} \mathsf{graph}(n, \gamma) \right) \tag{4.8}$$

$$\mathop{\biguplus\!\!\!\!*}_{l_1, l_2, \ldots, l_n} P \overset{\text{def}}{=} P(l_1) \uplus\!\!\!* P(l_2) \uplus\!\!\!* \cdots \uplus\!\!\!* P(l_n) \tag{4.9}$$

The proof of (4.8) can be split into two directions. The "$\vdash$" direction is not very hard, but the "$\dashv$" direction demands special care. The major obstacle is that if two vertex predicates $x \mapsto \gamma(x)$ and $y \mapsto \gamma(y)$ are partially overlapped, i.e. some—but not all–of x's memory cells are shared with y's, then the $*$ on the left hand side of $\dashv$ cannot separate them. To avoid such a situation we require that the spatial predicate $x \mapsto \gamma(x)$ be precise and joinable whose definitions are listed below.

$$\begin{aligned}\textsf{precise}\, P &\overset{\text{def}}{=} \forall h, h_1, h_2 \,.\, h_1 \models P \rightarrow h_2 \models P \rightarrow (\exists h_1' \,.\, h_1 \oplus h_1' = h) \rightarrow \\ &\qquad (\exists h_2' \,.\, h_2 \oplus h_2' = h) \rightarrow h_1 = h_2 \qquad (4.10)\\ \textsf{joinable}\, P &\overset{\text{def}}{=} \forall x, y \,.\, x \neq y \rightarrow P(x) \uplus\!\!\!\!\ast\; P(y) \vdash P(x) * P(y) \qquad (4.11)\end{aligned}$$

The definition of precise P means that if P is satisfied on a sub-heap, that sub-heap must be unique. The definition of joinable P means that for different arguments x and y, the spatial predicates $P(x)$ and $P(y)$ do not interfere at all. In a Java-like memory model this property is always true because pointers in such a model always point to the root/beginning of an object. In contrast, in a C-like memory model, this property is not always satisfied because pointers can point anywhere. In such a model, the joinable property is most easily enforced by storing graph vertices at addresses that are multiples of an appropriate size.

However, a graph with a "starting point" like graph(x, γ) is only one special case of possible graph representations. In real programs, the graph involved may not be limited to the reachable component of a certain vertex. To handle various possible graph representations in memory, we build a framework for all such pointwise representations and remain the possibility to support other forms such as an array. It is an intermediate layer to connect our mathematical graph library discussed in §3 and the separation logic library in MSL. It not only provides necessary infrastructure and interfaces to build customized spatial predicates for different graphs in various programs but also proves many spatial theorems under the assumptions of those interfaces. Once the interfaces are implemented in different applications, those theorems can be applied directly.

Figure 4.5 illustrates the interfaces in the spatial graph library and the

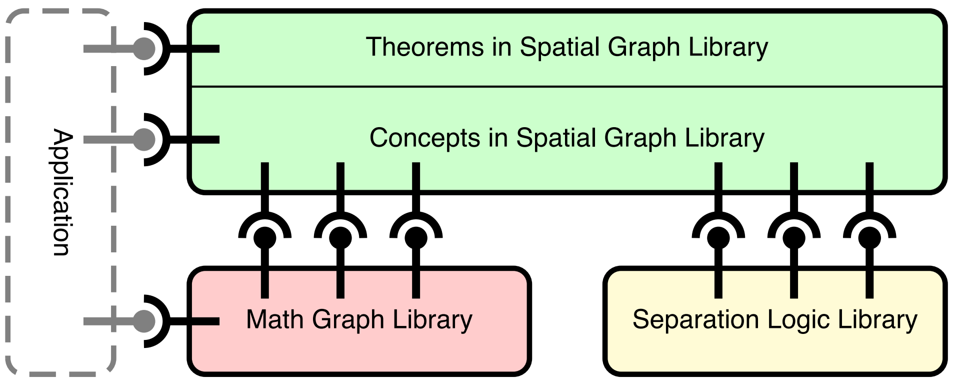

Theorems in Spatial Graph Library
Concepts in Spatial Graph Library
Application
Math Graph Library
Separation Logic Library

Figure 4.5: Interfaces of the Spatial Graph Library

relations to other libraries. The gadget —(represents the interfaces and another gadget —• represents implementations of certain interfaces. In the verification of concrete programs, all interfaces must have implementations. Otherwise the concepts and theorems in the library would have the unimplemented interfaces as additional parameters and hypotheses respectively. We design so many interfaces in the spatial graph library because we want to make it relatively independent from concrete memory models and particular definitions of graphs. It is a loosely coupled design. At the same time, as we can see in Figure 4.5, many interfaces have corresponding implementations in other libraries already. When we use this library, only a few interfaces need to be implemented for each program. They are about the unique features of graphs which are hard to estimate and account for until we embark on a concrete verification. Such a design achieves a good balance between independence and usability of the spatial graph library.

The interfaces are formulated as a set of layered type classes in Coq, in the file msl_applications/Graph.v:

```
Class PointwiseGraph (V E: Type) (GV GE: Type): Type...

Class PointwiseGraphPred (V E GV GE Pred: Type): Type...

Class PointwiseGraphBasicAssum (V E: Type): Type...

Class PointwiseGraphAssum {V E GV GE Pred: Type}

      (SGP: PointwiseGraphPred V E GV GE Pred)
      {SGBA: PointwiseGraphBasicAssum V E}: Type...
Class PointwiseGraphAssum_vs {V E GV GE Pred: Type}
      (SGP: PointwiseGraphPred V E GV GE Pred)
      {SGBA: PointwiseGraphBasicAssum V E}
      {SGA: PointwiseGraphAssum SGP}: Type...
Class PointwiseGraphAssum_vn...
Class ...
```

Correspondingly, the implementations of interfaces are formulated as instances of type classes. In these type classes, the latter ones take former ones as parameters. The graph information function $\gamma(x)$ is named `vgamma` in `PointwiseGraph`. The vertex representation $v \mapsto \gamma(v)$ is named `vertex_at` in `PointwiseGraphPred`. The instances of these two classes are application-dependent. The elements in `PointwiseGraphBasicAssum` have instances in the math graph library. The elements in `PointwiseGraphAssum` have instances in MSL. Other classes contain spatial properties of vertex representations, intermediate constructions to help build $\gamma(x)$, etc.

We can take the binary graph defined through `struct Node` in (4.3) as an example to see how to define a spatial predicate of a graph formally in Coq. Other spatial predicates used in our verification will be discussed in §5. In Figure 4.5 we can see there are gaps (unimplemented interfaces) for applications in the mathematical graph library. Our first step is filling those gaps, i.e. determining the types of vertices, edges, labels and the soundness condition of a graph. To be more specific, we need to provide the parameters `Vertex`, `Edge` for `PreGraph`; `DV`, `DE`, `DG` for `LabeledGraph`; and `Sound` for `GeneralGraph`. Since we will use the spatial predicate in the verification of C programs, it is better to instantiate the type of vertices as pointers in C. Here we use `pointer_val`—the formalized C pointer—defined in another external library called Verified Software Toolchain (VST) as the type `Vertex`. VST, which is build upon MSL, provides a program logic for the C programming language. The higher-order impredicative separation logic given in VST is proved sound with respect to the operational semantics of CompCert C. CompCert and VST will be discussed further in §5. Table 4.1 gives all determined types for the binary graph. The

Vertex	Edge	DV	DE	DG	Sound
point_val	point_val * LR	bool	unit	unit	BiMaFin

Table 4.1: Instantiated Types of a Binary Graph

definitions of `LR` and `BiMaFin` were introduced in §3.1.5. Any vertex `v` of type `point_val` has two out edges: `(v, L)` and `(v, R)`. Recall that the type class `BiGraph` contained in `BiMaFin` requires two additional parameters to derive the only two out edges from a certain vertex. It is the type of edge that makes the definitions of these two parameters possible. The type `unit` of edge labels and global labels can be seen as just a placeholder because there is no information attached to edges or the whole graph in this case. Besides the pointers, there is data field `m` of `int` in `struct Node`. Since the graph is used in a marking algorithm, `m` has only two possible values: 0 for "unmarked" and 1 for "marked". So we use `bool` for vertex labels.

The second step is define the graph information function $\gamma(x)$. It is defined as the following function in Coq:

$$\gamma(x) \overset{\text{def}}{=} \big(\texttt{vlabel}\,\gamma\,x,\ \texttt{dst}\,\gamma\,(x,\texttt{L}),\ \texttt{dst}\,\gamma\,(x,\texttt{R})\big) \tag{4.12}$$

In this definition, $\texttt{dst}\,\gamma\,(x,\texttt{L})$ and $\texttt{dst}\,\gamma\,(x,\texttt{R})$ are the destination of x's left and right edges respectively. The $\texttt{vlabel}\,\gamma\,x$ is the label of vertex x. This function returns a triple.

The third and last step is defining the vertex representation $x \mapsto \gamma(x)$. It is defined as the function `trinode` in Figure 4.6. This definition employs $\textsf{data\_at}\,\pi\,\tau\,v\,p$ defined in VST to represent $p \mapsto v$ with read/write

```
Definition trinode (sh: share) p dlr: mpred :=
  match dlr with | (d, l, r) => data_at sh node_type
        (Vint (Int.repr (if d then 1 else 0)),
         (pointer_val_val l, pointer_val_val r))
         (pointer_val_val p)
  end.
```

Figure 4.6: Coq Definition of $x \mapsto \gamma(x)$ for a Binary Graph

permission π. The τ is the type of v. It is used to calculate the size and memory layout. The parameter `sh` is the permission π. The `mpred` is the type of the spatial predicate in VST. It is an instance of `NatDed` and `SepLog`. Recall that we assumed a type `A` in the definition of `iter_sepcon` in our spatial graph library. When the library is used with VST, `A` is instantiated as `mpred`, just like the situation illustrated in Figure 4.5. The function `pointer_val_val` converts the pointer to the generic value type in C. The `node_type` actually represent the type `struct Node` in C. It will discussed further in §5. The most interesting part is treatment of vertex label `d` of type `bool`. It is encoded into integer C type in the vertex representation. It should be noted that sometimes the step 2 and 3 can be combined.

Once all three steps are done, we immediately get the spatial predicate of the binary graph `reachable_vertices_at` $x\,\gamma$ through a series of predefined functions listed below: These four definitions are fixed and many theorems about them are proved in the spatial graph library. For different applications with pointwise graph layout, we only need to finish the three steps before to fill the gaps. Then we get a whole toolkit for the concrete application for free.

```
Definition graph_vcell (g: Graph) (v : V) : Pred :=
           vertex_at v (vgamma g v).
Definition vertices_at (P: V -> Prop) (g: Graph): Pred :=
           pred_sepcon P (graph_vcell g).
Definition reachable_vertices_at (x : V) (g: Graph): Pred :=
           vertices_at (reachable g x) g.
Definition full_vertices_at (g: Graph): Pred :=
           vertices_at (vvalid g) g.
```

Figure 4.7: Generic Definitions for Spatial Representations of a Graph

4.2 Localize Rule and Spatial Inference

After defining the spatial predicates which can accurately describe the memory layout of graphs, it is convenient to use them in verification of programs, which is basically an inference based on the change of shapes of graphs. However, in the mechanized context we found that the RAMIFY rule in (2.5) is inadequate, especially in handling modified program variables and existential variables in the postconditions. To conquer the inadequacy of the RAMIFY rule, we propose a new inference rule called LOCALIZE rule and develop a notion of *localization block* that enables modular reasoning by using this rule in §4.2.1. Furthermore, we show that our LOCALIZE rule and the classic FRAME rule are co-derivable. In §4.2.1 we illustrate that the LOCALIZE rule can robustly handle modified local program variables through a detailed example. In §4.2.3 we discussed how to handle existential variables in the postconditions of localization blocks via the LOCALIZE rule. The existential variables occur frequently when using relations in specifications. The LOCALIZE rule and related theorems are essential in the verification of graph-manipulating programs.

Handling modified program variables is not a new problem. Hobor and Villard already realized that their RAMIFY rule is not applicable when its side condition is not satisfied, i.e. involving non-free modified program variables [Hobor and Villard, 2013]. They proposed two solutions. One is to use variables as resource by adding "variable contexts" to assertions in Hoare logic [Bornat et al., 2006], which actually introduces extra complications. The other is making local program transformations which substitute modified program variables with fresh new variables for once, and then later substitute them back. Both solutions are sufficient and reasonable in a pen-and-paper context, but are deficient in a mechanized context. When we started to verify real programs with the existing toolset, we found that neither of the two solutions is viable. Most mechanized verification systems including VST do not use variables as resource, and further do not support reasoning about program equivalence after local transformations [Beckert et al., 2007; Distefano and Parkinson, 2008; Bengtson et al., 2012]. We do not want to reinvent large wheels such as VST and CompCert, which roughly contain about 840k lines of code. Our new LOCALIZE rule respects these design decisions, and this makes it a more pragmatic solution when it comes to integration into a large existing toolset.

4.2.1 Localize rule and localization blocks

Here we give our improved inference rule, the LOCALIZE rule:

$$\textsc{Localize}\quad \frac{\{L_1\}\,C\,\{\exists x\,.\,L_2\} \qquad G_1 \vdash L_1 * R \qquad R \vdash \forall x\,.\,(L_2 \mathrel{-\!\!*} G_2)}{\{G_1\}\,C\,\{\exists x\,.\,G_2\}} \tag{4.13}$$

$$\text{where } \mathsf{FreeVar}(R) \cap \mathsf{ModVar}(C) = \emptyset$$

The LOCALIZE rule can be seen as an enhanced version of the RAMIFY rule in (2.5), which connects the "local" effect of a piece of program C, i.e. from L_1 to $\exists x\,.\,L_2$, with the "global" consequence, i.e. from G_1 to $\exists x\,.\,G_2$. The major differences from the RAMIFY rule are the existential variable $\exists x$ in postconditions and the *ramification frame* R as an extra level of indirection. If L_2 and G_2 do not contain free occurrence of x and we set $R = L_2 \mathrel{-\!\!*} G_2$, then the LOCALIZE rule degenerates to the RAMIFY rule. If we further let $G_i = L_i * F$ $(i = 1, 2)$ for some frame F which is untouched by C, then the LOCALIZE rule further degenerates to the well-known FRAME rule.

We formally proved that the LOCALIZE rule is sound in our spatial graph library. Here we give a sketch proof to show that both rules are co-derivable. There are two directions. We prove one direction first, from LOCALIZE to FRAME.

First, by introducing a fresh variable x_f, we have the following entailment by the CONSEQUENCE rule discussed in §2.1.1 of Hoare logic:

$$\frac{\{P\}\,C\,\{Q\} \qquad Q \vdash \exists x_f\,.\,Q}{\{P\}\,C\,\{\exists x_f\,.\,Q\}} \tag{4.14}$$

Secondly, from the definition of $\mathrel{-\!\!*}$ in (2.2), it is easy to prove the tautology $F \vdash \forall x_f\,.\,\Big(Q \mathrel{-\!\!*} (Q * F)\Big)$ for a disjoint frame F. By combining this tautology with the conclusion of (4.14), we can apply the LOCALIZE rule:

$$\frac{\{P\}\,C\,\{\exists x_f\,.\,Q\} \qquad P * F \vdash P * F \qquad F \vdash \forall x_f\,.\,\Big(Q \mathrel{-\!\!*} (Q * F)\Big)}{\{P * F\}\,C\,\{\exists x_f\,.\,(Q * F)\}} \tag{4.15}$$

Thirdly, we can apply the CONSEQUENCE rule again:

$$\frac{\{P * F\}\,C\,\{\exists x_f\,.\,(Q * F)\} \qquad \exists x_f\,.\,(Q * F) \vdash Q * F}{\{P * F\}\,C\,\{Q * F\}} \tag{4.16}$$

Thus, starting from $\{P\}\,C\,\{Q\}$, by using the LOCALIZE rule, we get the conclusion of the FRAME rule $\{P * F\}\,C\,\{Q * F\}$. This finishes the proof of the direction from LOCALIZE to FRAME. The side condition of the FRAME rule, $\mathsf{FreeVar}(F) \cap \mathsf{ModVar}(C)$, is directly inherited from the side condition required in (4.15) since we set $R \overset{\text{def}}{=} F$ there.

To prove the other direction, we need to derive $\{G_1\}\,C\,\{\exists x\,.\,G_2\}$ from three hypotheses: $\{L_1\}\,C\,\{\exists x\,.\,L_2\}$, $G_1 \vdash L_1 * R$, and $R \vdash \forall x\,.\,(L_2 \mathrel{-\!\!*} G_2)$. First, we can apply the FRAME rule to $\{L_1\}\,C\,\{\exists x\,.\,L_2\}$:

$$\frac{\{L_1\}\,C\,\{\exists x\,.\,L_2\}}{\{L_1 * R\}\,C\,\{(\exists x\,.\,L_2) * R\}} \tag{4.17}$$

Secondly, taking a little care with quantifiers, we can prove the tautology $(\exists x\,.\,L_2) * \Big(\forall x\,.\,(L_2 \mathrel{-\!\!*} G_2)\Big) \vdash \exists x\,.\,G_2$ according to (2.3) on page 23. By combining this tautology together with the hypothesis $R \vdash \forall x\,.\,(L_2 \mathrel{-\!\!*} G_2)$, it is not difficult to make the following entailment:

$$\frac{R \vdash \forall x\,.\,(L_2 \mathrel{-\!\!*} G_2) \qquad (\exists x\,.\,L_2) * \Big(\forall x\,.\,(L_2 \mathrel{-\!\!*} G_2)\Big) \vdash \exists x\,.\,G_2}{(\exists x\,.\,L_2) * R \vdash \exists x\,.\,G_2} \tag{4.18}$$

Thirdly, we can apply the CONSEQUENCE rule to the premise $G_1 \vdash L_1 * R$ and the conclusions of (4.17) and (4.18):

$$\frac{G_1 \vdash L_1 * R \qquad \{L_1 * R\}\,C\,\{(\exists x\,.\,L_2) * R\} \qquad (\exists x\,.\,L_2) * R \vdash \exists x\,.\,G_2}{\{G_1\}\,C\,\{\exists x\,.\,G_2\}}$$

This finishes the proof of the direction from FRAME to LOCALIZE. The side condition of the LOCALIZE rule, $\mathsf{FreeVar}(R) \cap \mathsf{ModVar}(C) = \emptyset$, is directly inherited from the side condition required in (4.17). Since both directions are proved, we establish the equivalence between the LOCALIZE rule and the FRAME rule, which means our LOCALIZE rule is sound in any separation logic. Our feet are solidly planted on the ground.

To better illustrate the application of the LOCALIZE rule, we propose a

```
// $\{G_1\}$
// $\searrow \{L_1\}$
...; x = f(...); ...
// $\swarrow \{\exists x \,.\, L_2\}$
// $\{\exists x \,.\, G_2\}$
```

Figure 4.8: A Typical Localization Block

notion called "localization blocks" appeared as comments before and after a piece of code C instead of the inference rule in (4.13). A typical localization block is shown in Figure 4.8. The existential quantifier $\exists x$ in postconditions is not always necessary since we have $\forall P \,.\, (P \dashv\vdash \exists x \,.\, P)$ when P does not contain free x. We do not really insert $\{G_1\}$, $\searrow \{L_1\}$, etc., into the program as comments. The localization block in Figure 4.8 just indicates that we are applying the Localize rule to the Hoare triple $\{L_1\}\, C\, \{\exists x \,.\, L_2\}$ in order to get $\{G_1\}\, C\, \{\exists x \,.\, G_2\}$. It gives us an intuition that we are allowed to zoom in from a larger "global" context $\{G_1\}$ to a smaller "local" one $\{L_1\}$, and, after verifying a program C locally and arriving at a local postcondition $\{\exists x \,.\, L_2\}$, to zoom back out to the global context $\{\exists x \,.\, G_2\}$. If we want to save vertical space, the context in Figure 4.8 can be written as $\{G_1\} \searrow \{L_1\}$ and $\{\exists x \,.\, G_2\} \swarrow \{\exists x \,.\, L_2\}$.

Since the Localize rule has three hypotheses, there are two proof obligations omitted in Figure 4.8 besides verifying $\{L_1\}\, C\, \{\exists x \,.\, L_2\}$. They will be discussed in §4.2.2 and §4.2.3 to demonstrate the power of the Localize rule.

4.2.2 Handling modified program variables

Consider using the Localize rule to verify the code snippet in Figure 4.9:

Since the localization block is overkill for a single assignment, we can suppose that the elided part in line 2 of the program makes it desirable.

```
// {x = 5 ∧ K} ↘ {x = 5 ∧ L}
...; x = x + 1; ...
// {x = 6 ∧ N} ↙ {x = 6 ∧ M}
```

Figure 4.9: Verification of a Program Involving Assignment

We further assume that only x is modified in the Figure 4.9, none of K, L, M, and N contains free x, and the local variable issue is the only problem pending to be solved. In other words, we have $K \vdash L * (M \mathrel{-\!\!*} N)$ and the Hoare triple $\{\mathsf{x} = 5 \wedge L\}$`...; x = x + 1; ...`$\{\mathsf{x} = 6 \wedge M\}$ holds.

To finish the whole entailment, i.e. the application of the LOCALIZE rule (4.13) in Figure 4.9, we have to specify a proper R to prove the remaining two proof obligations: $G_1 \vdash L_1 * R$ and $R \vdash L_2 \mathrel{-\!\!*} G_2$. If we set $R \stackrel{\text{def}}{=} L_2 \mathrel{-\!\!*} G_2$, then the latter obligation is trivial but the program variable x will appear in all four places in the former entailment:

$$\overbrace{(\mathsf{x} = 5 \wedge K)}^{G_1} \vdash \overbrace{(\mathsf{x} = 5 \wedge L)}^{L_1} * \Big(\overbrace{(\mathsf{x} = 6 \wedge M)}^{L_2} \mathrel{-\!\!*} \overbrace{(\mathsf{x} = 6 \wedge N)}^{G_2} \Big) \tag{4.19}$$

The side condition of the LOCALIZE rule, $\mathsf{FreeVar}(R) \cap \mathsf{ModVar}(C) = \emptyset$ is not satisfied any more. In this case, since $R = L_2 \mathrel{-\!\!*} G_2$, we have $\mathsf{FreeVar}(L_2 \mathrel{-\!\!*} G_2) = \{\mathsf{x}\}$ and x is the modified variable in the code C.

To resolve the issue, we need a little trick to untangle the connection between x and $L_2 \mathrel{-\!\!*} G_2$. First, we define $\hat{L}_2(x_f) \stackrel{\text{def}}{=} (x_f = 6 \wedge M)$ and $\hat{G}_2(x_f) \stackrel{\text{def}}{=} (x_f = 6 \wedge N)$ where x_f is a harmless fresh meta-variable which replaces the troublesome program variable x in L_2 and G_2. Next, with this carefully chose existential quantifier, we can still express the original L_2 and isolate the troublesome program variable x. The decorated program is then transformed into the new form in Figure 4.10.

We can further define $\tilde{L}_2(x_f) \stackrel{\text{def}}{=} (x_f = \mathsf{x}) \wedge \hat{L}_2(x_f)$, i.e. $(x_f = \mathsf{x}) \wedge (x_f = 6 \wedge M)$ and similarly corresponding $\tilde{G}_2(x_f)$. Thus line 4 and 5 are exactly in the form $\exists x_f \,.\, \tilde{L}_2(x_f)$ and $\exists x_f \,.\, \tilde{G}_2(x_f)$, which are both permitted by

```
1 // $\{\mathsf{x} = 5 \wedge K\} \searrow \{\mathsf{x} = 5 \wedge L\}$
2 ...; x = x + 1; ...;
3 // $\{\mathsf{x} = 6 \wedge M\}$
4 // $\swarrow \{\exists x_f \,.\, (x_f = \mathsf{x}) \wedge (x_f = 6 \wedge M)\}$
5 // $\{\exists x_f \,.\, (x_f = \mathsf{x}) \wedge (x_f = 6 \wedge N)\}\}$
6 // $\{\mathsf{x} = 6 \wedge N\}\}$
```

Figure 4.10: Isolate Program Variable via a New Existential Variable

the LOCALIZE rule. Reasoning from line 3 to line 4 and from line 5 to line 6 are trivial. After such a shift, the localization block in Figure 4.10 is still an application of the LOCALIZE rule. But this time we can set $R \overset{\text{def}}{=} \forall x_f \,.\, \hat{L}_2(x_f) \mathrel{-\!\!*} \hat{G}_2(x_f)$, i.e. $\forall x_f \,.\, (x_f = 6 \wedge M) \mathrel{-\!\!*} (x_f = 6 \wedge N)$. By construction, R is free from all program variables modified by the code snippet, so the side condition of LOCALIZE is satisfied. Still we only need to consider the rest two proof obligations, $G_1 \vdash L_1 * R$ and $R \vdash \forall x \,.\, (\tilde{L}_2 \mathrel{-\!\!*} \tilde{G}_2)$. We can unfold the definition of $\tilde{L}_2$ and $\tilde{G}_2$ of the second one to get this:

$$\left(\forall x_f \,.\, \hat{L}_2(x_f) \mathrel{-\!\!*} \hat{G}_2(x_f)\right) \vdash \\ \forall x_f \,.\, \left((x_f = \mathsf{x}) \wedge \hat{L}_2(x_f)\right) \mathrel{-\!\!*} \left((x_f = \mathsf{x}) \wedge \hat{G}_2(x_f)\right)$$

This turns out to be just a verbose tautology which can be proved automatically in Coq.

If we unfold the definition in the first obligation, it would look like:

$$(\mathsf{x} = 5 \wedge K) \vdash (\mathsf{x} = 5 \wedge L) * \left(\forall x_f \,.\, (x_f = 6 \wedge M) \mathrel{-\!\!*} (x_f = 6 \wedge N)\right)$$

This can be decomposed into a "variable-related" part $(\mathsf{x} = 5) \vdash (\mathsf{x} = 5) * \left(\forall x_f \,.\, (x_f = 6) \mathrel{-\!\!*} (x_f = 6)\right)$, which is also an easily provable tautology, and a "spatial" part $K \vdash L * (M \mathrel{-\!\!*} N)$, which is true by assumption.

With careful engineering, we made such modified-variable tricks fully automatic in our spatial library. These details are completely hidden to end-users. They only need to handle the spatial part $K \vdash L * (M \mathrel{-\!\!*} N)$

which actually captures the key action of the localization block. We call the entailment of this form *ramification entailment* and theorems involving it *ramification theorems*. Intuitively, $K \vdash L * (M \mathrel{-\!\!*} N)$ says that given a large shape K that contains a small shape L within it, we can change L into a new small shape M. This substitution causes the large "global" shape to become N. To solve these in practice, we observed some common patterns and proved several generic ramification theorems which are discussed in §4.3.

4.2.3 Handling existential variables

We have seen that the Localize rule can smoothly handle modified program variables. However, there is another important application scenario in which the Localize rule can play a central role. It is when we cannot precisely determine the value of a program variable but have to introduce the existential variable anyway. Consider the localization block in Figure 4.11.

```
1 // $\{K\} \searrow \{L\}$
2 ...; x = rand();
3 if (is_prime(x)) y = 0; else y = 1; ...;
4 // $\swarrow \{\big(\mathsf{x} =?\wedge(\mathsf{y}=1\vee\mathsf{y}=0)\big) * M\}$
5 // $\{\big(\mathsf{x} =?\wedge(\mathsf{y}=1\vee\mathsf{y}=0)\big) * N\}$
```

Figure 4.11: Non-Deterministic Behavior in a Code Snippet

Within the localization block in Figure 4.11 we call the function `rand` which generates pseudo-random integers and use the program variable `y` as a flag to keep track of whether the generated `x` is a prime number. We make necessary assumptions just like §4.2.2 to ensure the program variable issue with `x` and `y` is the only problem.

If we call the postconditions in line 4 and 5 as L_2 and G_2, then we have a similar problem as in §4.2.2: the modified program variables `x` and `y` appeared in L_2 and G_2, which violates the side condition of LOCALIZE. However, a more obvious and important problem is the question mark "?" in line 4 and 5. Unlike §4.2.2, we do not know the concrete value of `x` this time. In this sense L_2 and G_2 are not valid propositions at all. The solution to this problem is the same as when solving the issue of the modified program variable: introducing existential quantifiers.

We proceed as follows. First, we define $\hat{L}_2(y_f) \overset{\text{def}}{=} \Big((y_f = 1 \vee y_f = 0) * M\Big)$ and $\hat{G}_2(y_f) \overset{\text{def}}{=} \Big((y_f = 1 \vee y_f = 0) * N\Big)$ where y_f is a harmless fresh meta-variable which replaces the program variable `y` in L_2 and G_2. Next, we further define $\tilde{L}_2(x_f, y_f) \overset{\text{def}}{=} (x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge \hat{L}_2(y_f)$, i.e. $(x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge (y_f = 1 \vee y_f = 0) * M$ and similarly corresponding $\tilde{G}_2(x_f, y_f) \overset{\text{def}}{=} (x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge \hat{G}_2(y_f)$, i.e. $(x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge (y_f = 1 \vee y_f = 0) * N$ where x_f is another harmless fresh meta-variable which represents the unknown value of the program variable `x`. Thus line 4 and 5 can be transformed to the form $\exists x_f, y_f \,.\, \tilde{L}_2(x_f, y_f)$ and $\exists x_f, y_f \,.\, \tilde{G}_2(x_f, y_f)$, which is shown in Figure 4.12.

```
1 // $\{K\} \searrow \{L\}$
2 ...; x = rand();
3 if (is_prime(x)) y = 0; else y = 1; ...;
4 // $\swarrow \{\exists x_f, y_f \,.\, (x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge (y_f = 1 \vee y_f = 0) * M\}$
5 // $\{\exists x_f, y_f \,.\, (x_f = \mathtt{x} \wedge y_f = \mathtt{y}) \wedge (y_f = 1 \vee y_f = 0) * N\}$
```

Figure 4.12: Introducing Existential Quantifier in Postconditions

As an application of the LOCALIZE rule (4.13), we need to find an R so that $K \vdash L * R$ and $R \vdash \forall x_f, y_f \,.\, \Big(\tilde{L}_2(x_f, y_f) \mathrel{-\!\!*} \tilde{G}_2(x_f, y_f)\Big)$. This time we can set $R \overset{\text{def}}{=} \forall y_f \,.\, \hat{L}_2(y_f) \mathrel{-\!\!*} \hat{G}_2(y_f)$. After unfolding, the first proof obligation becomes

$$K \vdash L * \forall y_f \,.\, \Big((y_f = 1 \vee y_f = 0) * M\Big) \mathrel{-\!\!*} \Big((y_f = 1 \vee y_f = 0) * N\Big)$$

whose spatial part $K \vdash L * (M \mathrel{-\!\!*} N)$ is the assumption and the variable related part $\mathsf{emp} \vdash \mathsf{emp} * \Big(\forall y_f \,.\, (y_f = 1 \vee y_f = 0) \mathrel{-\!\!*} (y_f = 1 \vee y_f = 0)\Big)$ is a tautology. For the second proof obligation, we unfold the definitions of $\tilde{L}_2(x_f, y_f)$ and $\tilde{G}_2(x_f, y_f)$ to get the following:

$$\Big(\forall y_f \,.\, \hat{L}_2(y_f) \mathrel{-\!\!*} \hat{G}_2(y_f)\Big) \vdash \forall x_f, y_f \,.\, \Big((x_f = \mathsf{x} \wedge y_f = \mathsf{y}) \wedge \hat{L}_2(y_f)\Big) \mathrel{-\!\!*} \Big((x_f = \mathsf{x} \wedge y_f = \mathsf{y}) \wedge \hat{G}_2(y_f)\Big).$$

This turns out again to be a verbose tautology. Therefore we successfully verify the localization block in Figure 4.12. In this case, the introduction of y_f is the same trick played in §4.2.2, but the introduction of x_f can be seen as a natural requirement proposed by the non-deterministic nature of certain programs. Through this example we can see that the LOCALIZE rule can handle it properly.

4.3 Library of Spatial Graph and Inference

In §4.1 we discussed our design decision about the spatial representation of graphs and take binary graph as an example to show how to formally define a spatial predicate by implementing the structural interfaces of the spatial library. The definition of $\circledast$ and various interfaces form the core component in the spatial graph library. In §4.2 we demonstrate the effectiveness of our LOCALIZE rule which can smoothly handle the modified program variables and existential quantifiers. The LOCALIZE rule is the crucial inference rule to reason about the spatial properties in verifying graph-manipulating programs through separation logic. So basically §4.1 explored the core concepts of the spatial library and §4.2 explained the most critical theorem of the spatial library in great detail. As pointed out in the beginning of this chapter, the spatial graph library also contains the infrastructure to adapt the formal library of separation logic and some common ramification theorems which can ease the application of the LOCALIZE rule. In the rest of this chapter, we discuss these components from the whole perspective.

4.3.1 Architecture of the spatial graph library

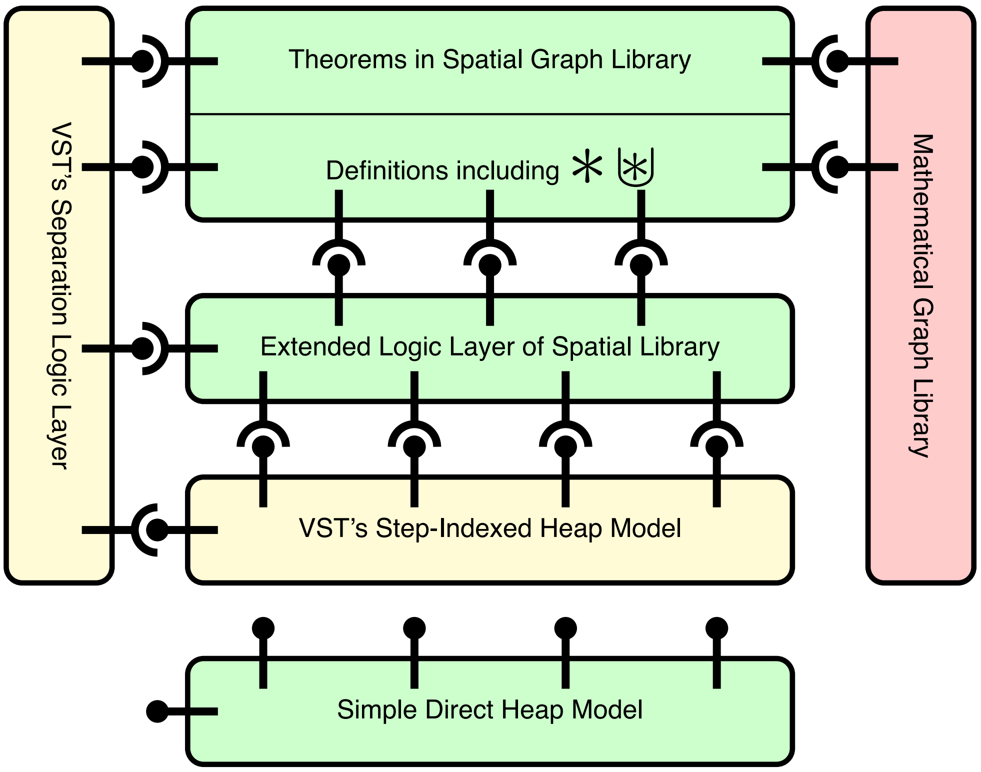

Theorems in Spatial Graph Library
Definitions including $*$ $\uplus$
Extended Logic Layer of Spatial Library
VST's Step-Indexed Heap Model
Simple Direct Heap Model
VST's Separation Logic Layer
Mathematical Graph Library

Figure 4.13: Architecture of the Spatial Graph Library

Figure 4.13 illustrates the overall architecture of our spatial graph library and its relations with the mathematical graph library and part of the Verified Software Toolchain (VST). It can be seen as a more detailed version of Figure 4.5. The three light green blocks together constitute the spatial graph library. The two light yellow blocks belong to the VST. The pink box is our mathematical graph library discussed in §3.

As pointed out in §4.1.2, we intend to have a loosely coupled design. To be more specific, the relative independence of the spatial graph library is achieved via logic layers that isolate the concrete heap models and various spatial predicates. VST constructs an unusually complex step-indexed heap model to support an unusually rich program logic [Appel et al., 2014]. But for representing spatial predicates and inference in separation logic, such a complex model is not necessary. There is a component called Mechanized Semantic Library (MSL) in VST which provides a separation logic layer. Instead of giving definitions of $*$ and $-\!\!*$ such as (2.1) and (2.2), MSL gives a set of logic rules about $*$ and $-\!\!*$ without touching the definitions of them. Figure 4.14 gives several inference rules in separation logic. They are

$$\textsc{SepconWand}\ \dfrac{P * Q \vdash R}{P \vdash Q -\!\!* R} \qquad \textsc{WandSepcon}\ \dfrac{P \vdash Q -\!\!* R}{P * Q \vdash R} \qquad \textsc{SepconDerives}\ \dfrac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

$$\textsc{SepconAssoc}\ \dfrac{}{(P * Q) * R \dashv\vdash P * (Q * R)} \qquad \textsc{SepconComm}\ \dfrac{}{P * Q \dashv\vdash Q * P}$$

Figure 4.14: Part of Inference Rules of Separation Logic

actually packed into a Coq class called `SepLog` provided in MSL. According to the step-indexed heap model, VST also provides instances of `SepLog`, which gives concrete definitions of $*$, $-\!\!*$ and proves that these inference rules holds in their model. This is the only connection between the model and separation logic in VST. All other separation logic related definitions depend on the notations (e.g. `emp`, $*$, $-\!\!*$) provided in `SepLog` and all related theorems are proved by using the inference rules in `SepLog`. They are unaware of the concrete heap models.

Inspired by the logic layer in VST, we also gives our own logic layer to accommodate the notations of $\mathbin{\cup\!\!\!\!\ast}$, `precise`, etc. and related inference rules. We call this the extended logic layer because it relies on the separation logic layer in VST, i.e. it also employs $*$ and $-\!\!*$. Figure 4.15 gives several inference rules involving $\mathbin{\cup\!\!\!\!\ast}$ and `precise`. The graph fold/unfold theorem (4.8) is proved according to these rules. We cannot dig into the model

$$\textsc{PreciseSepcon}\ \frac{\mathsf{precise}\ P \qquad \mathsf{precise}\ Q}{\mathsf{precise}\ (P * Q)} \qquad \textsc{DerivesPrecise}\ \frac{P \vdash Q \qquad \mathsf{precise}\ Q}{\mathsf{precise}\ P} \qquad \textsc{AndOcon}\ \frac{P \wedge Q}{P \uplus Q}$$

$$\textsc{SepOcon}\ \frac{P * Q}{P \uplus Q} \qquad \textsc{OconDerives}\ \frac{P \vdash P' \qquad Q \vdash Q'}{P \uplus Q \vdash P' \uplus Q'} \qquad \textsc{PreciseOcon}\ \frac{\mathsf{precise}\ P \qquad \mathsf{precise}\ Q}{\mathsf{precise}\ (P \uplus Q)}$$

$$\textsc{OconAssoc}\ \frac{}{(P \uplus Q) \uplus R \dashv\vdash P \uplus (Q \uplus R)} \qquad \textsc{OconComm}\ \frac{}{P \uplus Q \dashv\vdash Q \uplus P}$$

$$\textsc{OconWand}\ \frac{}{(R \multimap P) * (R \multimap Q) * R \vdash P \uplus Q}$$

Figure 4.15: Part of Inference Rules in Extended Logic Layer

level by unfolding the definitions of $\mathsf{precise}$ in (4.10) or $\uplus$ in (4.2) to prove it because the logic layer may have multiple models with very different definitions. In our spatial graph library, the notations of $\uplus$ and $\mathsf{precise}$ together with the inference rules partially enumerated in Figure 4.15 are packed into several Coq classes such as `PreciseSepLog` and `OverlapSepLog`. These classes compose the "Extended Logic Layer of Spatial Library" in Figure 4.13. To prove the soundness of the inference rules proposed in `PreciseSepLog`, `OverlapSepLog` and other classes in the extended logic layer, we instantiate each class using basic definitions in VST's step-indexed heap model. VST's separation logic layer and our extended logic layer provide a good abstraction of $*$, $\uplus$, $\mathsf{precise}$, etc. which are necessary for definitions and theorems discussed in §4.1 and §4.2.

To validate the certain independence of the two logic layers, we even define another simple direct heap model implementing all interfaces (i.e. instantiating all classes) in VST's separation logic layer and our extended logic layer. In this model, the memory heap is simply defined as a finite partial map from address to address, where an address is simply defined as a natural number. In heap_model_direct/SeparationAlgebra.v we give the definition of heap and prove that it can be viewed as an instance of the separation algebra discussed in §2.4.2 with properly-defined `join` relation. In heap_model_direct/SeparationLogic.v we prove that the spatial predicates on this heap definition do satisfy all rules of `NatDed`, `SepLog`, `PreciseSepLog`, `OverlapSepLog` and etc. In heap_model_direct/mapsto.v we give the definition $x \mapsto v$ on our heap model and prove that it satisfies some other additional classes of the extended logic layer. These three files compose the isolated "Simple Direct Heap Model" in Figure 4.13. But the shape and number of "stubs" on it indicate that it can entirely replace "VST's Step-Indexed Heap Model" as the underlying model. The successful definitions and implementations of interfaces of logic layers prove that the separation logic layer and our extended logic layer are generic. They do not depend on certain special features of the underlying model. Thus our spatial predicates and theorems built on the two logic layers are generic and independent of concrete heap models.

On top of the "Extended Logic Layers of Spatial Library" is the upper layer definitions and theorems in the spatial graph library. They include the iterated separating conjunction $\mathop{\Huge\ast}$, the iterated overlapping conjunction $\mathop{\cup\!\!\!\!\!\ast}$, and the interfaces which help constructing the spatial representations of graphs discussed in great detail in §4.1.2. We introduce and discuss theorems of the spatial graph library in §4.3.2.

4.3.2 Theorems in the spatial graph library

Figure 4.16 on page 109 gives an overview of the theorems in the spatial graph library. All 400 theorems spread over 23 files surround a big circle. Just like Figure 3.7, the length of a line segment indicates the length of a theorem in log scale and the connecting curves indicate dependencies.

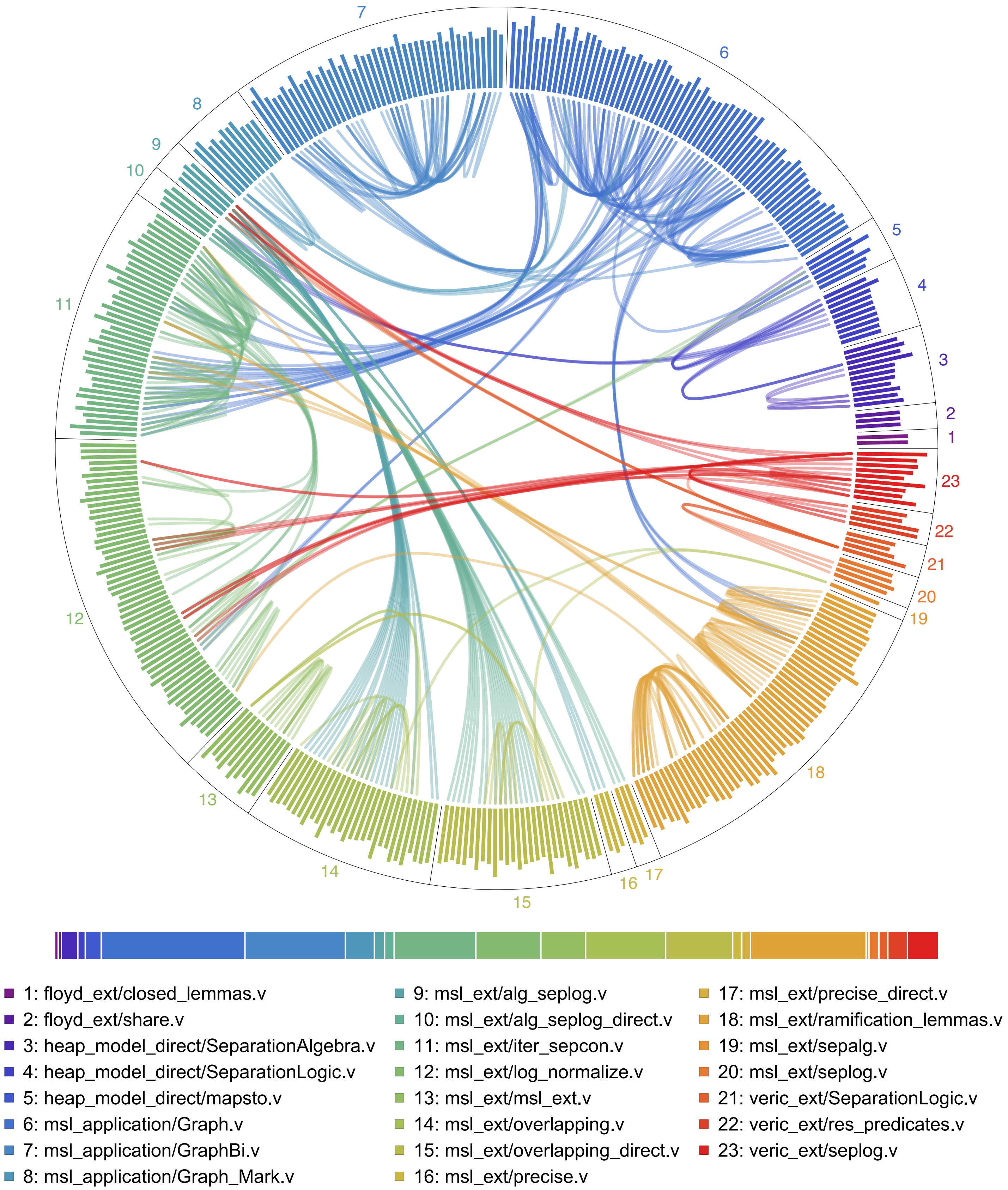


Figure 4.16: The Overview of the Theorems in the Spatial Library

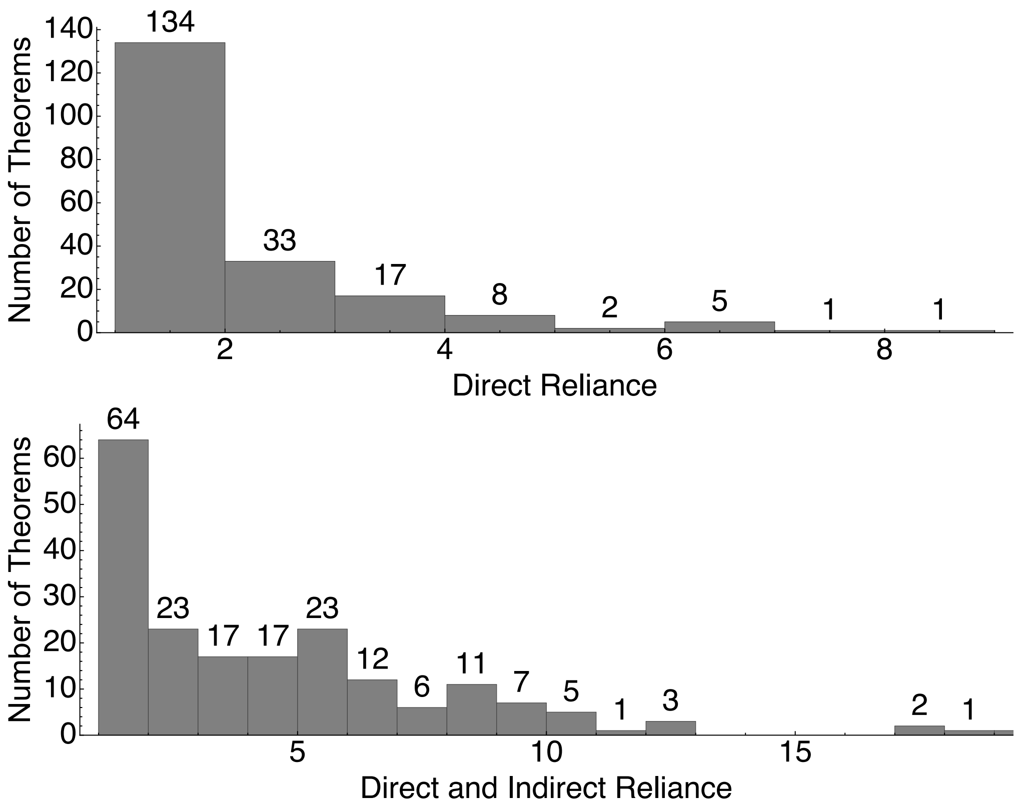


Figure 4.17: Theorem Dependencies in Spatial Graph Library

Compared to Figure 3.7, there are fewer connections in Figure 4.16. The statistic about the theorems further confirms this impression, although it is because there are fewer theorems in the spatial graph library than in the mathematical graph library. Among the 400 theorems, 197 of them are basic facts which do not cite any other theorem. In the remaining 200 theorems, more than half of them rely on only one or two other theorems. Figure 4.17 gives the distribution of the rest 203 theorems according to the extent of reliance. We do not count the theorems outside the spatial graph library, i.e. the theorems in the standard library of Coq or in the mathematical graph library. It is interesting to see that in the lower histogram of Figure 4.17, the distribution is not as extreme as the upper one. A greater number of theorems have more than two supporting theorems in total. This distinction between Figure 4.17 and Figure 3.10 suggests that the theorems in the spatial graph library are more closely connected than theorems in the mathematical graph library.

From now we focus on Figure 4.16, explore each file in the spatial graph library, and explain some important theorems in it when necessary.

Sector 1 and 2 (floyd_ext/closed_lemmas.v and floyd_ext/share.v) are infrastructure to adapt to the VST library. The two theorems in sector 1 handle the issue of variables and $-\!\!*$ automatically for VST since the spatial library uses $-\!\!*$ extensively. In another file floyd_ext/share.v we define two specialized permission entities which can ease the verification. The permission required by VST is used in our final spatial representation of vertex `trinode` discussed in §4.1.2. In VST, $p \mapsto v$ is formalized as data_at $\pi\, \tau\, v\, p$ where π is the read/write permission. This infrastructure is later used in our verification and the theorems are used implicitly as a automatic mechanism, so there is no explicit reference (curve) to them.

The next three files (sectors 3, 4 and 5) belong to the "Simple Direct Heap Model" layer of the spatial library, which are discussed in §4.3.1. As discussed in §2.4.2, separation algebra can be seen as an abstraction layer between separation logic and underlying concrete model. The theorems in SeparationAlgebra.v and SeparationLogic.v are instances of VST's classes of separation algebra and separation logic respectively. They together claim that our direct model fulfills all inference rules of separation logic. Part of the rules are listed in Figure 4.14.

Three files Graph.v, GraphBi.v and Graph_Mark.v under msl_application (sectors 6, 7 and 8) contains definitions and theorems about the spatial representation of graphs. Graph.v contains various definitions and interfaces explained in §4.1. GraphBi.v is a partial implementation of the interfaces defined in Graph.v, which focuses on the binary graph. Since we verify two programs about the binary graph which share the same data structure, we classify this file to the spatial graph library. Graph_Mark.v contains several generic spatial theorems about the abstract marking relation among graphs. They are also used in verifying two programs, so we also classify this file to the spatial graph library.

In Graph.v, we prove several generic ramification theorems which are crucial for further verification. One of them is:

$$\frac{\forall v\,.\,\Big(v \neq x \to v \neq x' \to \gamma(v) = \gamma'(v)\Big) \wedge \Big(P(v) \wedge v \neq x \leftrightarrow P'(v) \wedge v \neq x'\Big)}{\texttt{vertices\_at}\, P\, \gamma \vdash x \mapsto \gamma(x) * \Big(x' \mapsto \gamma'(x') \mathrel{-\!\!*} \texttt{vertices\_at}\, P'\, \gamma'\Big)} \tag{4.20}$$

The spatial predicate $\texttt{vertices\_at}\, P\, \gamma$ is defined in Figure 4.7. The different definitions of the vertex predicate P could derive different spatial representations of graphs based on `vertices_at`. This theorem involves two vertex predicates P, P' and two graphs γ, γ'. If they satisfy a rather complex condition above the line, we have the conclusion below the line. The condition basically says that the two predicates P, P' are almost equivalent and the two graphs γ, γ' are almost the same, except on vertices x and x'. The conclusion is exactly the ramification entailment discussed in the end of §4.2.2, which is the proof obligation of the LOCALIZE rule. In the verification of graph-manipulating programs, when a single vertex of graph γ is modified, we get a slightly changed new graph γ'. The theorem (4.20) can be used in this case. Since the P, P', γ and γ' are all generic, the theorem itself is widely applicable.

When more vertices of a graph are changed, theorem (4.20) is not applicable any more. But Graph.v provides the following theorem:

$$\frac{\begin{array}{c}\forall v\,.\,\Big(\neg L(v) \to \neg L'(v) \to \gamma(v) = \gamma'(v)\Big) \wedge \\ \Big(G(v) \wedge \neg L(v) \leftrightarrow G'(v) \wedge \neg L'(v)\Big)\end{array}}{\begin{array}{l}\texttt{vertices\_at}\, G\, \gamma \vdash \texttt{vertices\_at}\, L\, \gamma\, * \\ \qquad\qquad \big(\texttt{vertices\_at}\, L'\, \gamma' \mathrel{-\!\!*} \texttt{vertices\_at}\, G'\, \gamma'\big)\end{array}} \tag{4.21}$$

where G, G', L and L' are all vertex predicates. The condition just says that, except the changed part (L and L'), the graphs γ, γ' are the same everywhere and the predicates G, G' are equivalent. The conclusion is another typical ramification theorem. The graph fold/unfold theorem (4.8) is also in Graph.v. The theorems in GraphBi.v can be seen as the specialized version of the theorems in Graph.v for binary graphs.

Sectors 9 and 10 are two files containing the instances ($-\!\!\bullet$) of classes ($-\!\!($) in the "Extended Logic Layer of Spatial Library". Every notation in a class needs to be defined and every property claimed in a class needs to be proved in a certain model. The concrete definitions and theorems are contained in sectors 14 to 17. So we can see many curves connecting them in Figure 4.16. Sectors 14 and 15 contain the definition of $\cup\!\!\!\!\ast$ and sectors 16 and 17 contain the definition of precise. Among the six sectors, the files whose names with suffix "_direct" are based on the simple direct heap model and the rest are based on the step-indexed heap model. Some of the theorems in these files are listed in Figure 4.15.

Sector 11 (msl_ext/iter_sepcon.v) contains the definitions of $\mathop{\Large\ast}$ (4.5, 4.6) and $\bigcup\!\!\!\!\!\ast$ (4.9) and related theorems. The theorems are about the basic properties of $\mathop{\Large\ast}$ and $\bigcup\!\!\!\!\!\ast$. We list some of them in Figure 4.18. Among these theorems, the theorem PredSepSep is particularly interesting. It says that if the spatial predicate P has a certain property about uniqueness (i.e. $P(x) * P(x) \vdash \bot$), then from the separating conjunction of $\mathop{\Large\ast}\limits_{M} P$ and $\mathop{\Large\ast}\limits_{N} P$, we can infer that the pure logic predicate are disjointed (i.e. an element x cannot satisfy M and N simultaneously). This theorem enables us to infer pure logic facts from spatial facts. Since the spatial representations of graphs adopt $\mathop{\Large\ast}$ in their definitions and the graph fold/unfold theorem (4.8) involves $\bigcup\!\!\!\!\!\ast$, it is no surprise that there are a lot of curves growing from sector 6 (msl_application/Graph.v) to the related theorems in sector 11.

Sector 12 (msl_ext/log_normalize.v) collects many separation logic related theorems directly derived from the logic layers. The inference rules about

$$\textsc{IterSepApp}\quad \frac{}{\mathop{\Huge\ast}\limits_{L_1+L_2} P = \mathop{\Huge\ast}\limits_{L_1} P \ast \mathop{\Huge\ast}\limits_{L_2} P} \qquad \textsc{IterOconApp}\quad \frac{}{\mathop{\bigcup\!\!\!\!\ast}\limits_{L_1+L_2} P = \mathop{\bigcup\!\!\!\!\ast}\limits_{L_1} P \uplus\!\!\!\!\ast\ \mathop{\bigcup\!\!\!\!\ast}\limits_{L_2} P}$$

$$\textsc{IterSepComm}\quad \frac{}{\mathop{\Huge\ast}\limits_{L_1+L_2} P = \mathop{\Huge\ast}\limits_{L_2+L_1} P} \qquad \textsc{IterSepNil}\quad \frac{}{\mathop{\Huge\ast}\limits_{\mathsf{nil}} P = \mathsf{emp}} \qquad \textsc{IterOconSep}\quad \frac{}{\mathop{\Huge\ast}\limits_{L} P \vdash \mathop{\bigcup\!\!\!\!\ast}\limits_{L} P}$$

$$\textsc{IterSepFunc}\quad \frac{\forall x\,.\,P(x) = Q(x)}{\mathop{\Huge\ast}\limits_{L} P = \mathop{\Huge\ast}\limits_{L} Q} \qquad \textsc{PredSepJoin}\quad \frac{\forall x\,.\,M(x) \to N(x) \to \bot}{\mathop{\Huge\ast}\limits_{M} P \ast \mathop{\Huge\ast}\limits_{N} P = \mathop{\Huge\ast}\limits_{M \wedge N} P}$$

$$\textsc{PredSepSep}\quad \frac{\forall x\,.\,P(x) \ast P(x) \vdash \bot}{\mathop{\Huge\ast}\limits_{M} P \ast \mathop{\Huge\ast}\limits_{N} P \vdash \forall x\,.\,M(x) \to N(x) \to \bot}$$

Figure 4.18: Basic Facts of `iter_sepcon`, `pred_sepcon` and `iter_ocon`

$\ast$, $\uplus\!\!\!\!\ast$, precise, etc. in VST's separation logic layer and our extended logic layer can be seen as a minimum set for reasoning about separation logic and related concepts. In practice, we find that some other inference rules not in the minimum set are also quite generic and useful. They can be proved using existing inference rules. From Figure 4.16 we can see that there are plenty of them. The curves connecting it and other sectors (5, 6, 11 and 18) show that these theorems are widely used in the spatial graph library. Some of the theorems are listed in Figure 4.19.

Sectors 13 and 19 (msl_ext/msl_ext.v and msl_ext/sepalg.v) contain certain low-level components which contain several derived theorems about separation algebra. Since separation algebra is independent of concrete heap models, we can see the theorems are used in theorems relating to both the direct and the indirect model (overlapping.v and overlapping_direct.v).

Sector 18 (msl_ext/ramification_lemmas.v) contains various ramification theorems (i.e. involving the form $K \vdash L \ast (M \mathrel{-\!\!\ast} N)$) about `iter_sepcon` and `pred_sepcon` (i.e. $\mathop{\Huge\ast}$) which form the bedrock of the ramification theo-

$$\textsc{PrcsAndLft}\ \frac{\textsf{precise}\ P}{\textsf{precise}\ (P \wedge Q)} \qquad \textsc{PrcsAndRht}\ \frac{\textsf{precise}\ Q}{\textsf{precise}\ (P \wedge Q)} \qquad \textsc{PrcsSepAnd}\ \frac{\textsf{precise}\ P}{(P * Q) \wedge (P * R) \vdash P * (Q \wedge R)}$$

$$\textsc{OconSepTrue}\ \frac{}{P \uplus Q \vdash P * \top} \qquad \textsc{SepWandWandSep}\ \frac{}{(P \mathrel{-\!\!*} Q) * R \vdash P \mathrel{-\!\!*} (Q * R)} \qquad \textsc{OconEmp}\ \frac{}{\textsf{emp} \uplus P = P}$$

$$\textsc{OconSelf}\ \frac{}{P \vdash P \uplus P} \qquad \textsc{OconSepCancel}\ \frac{}{P * Q \vdash P \uplus (P * Q)} \qquad \textsc{TriSepOcon}\ \frac{}{P * Q * R \vdash (P * Q) \uplus (Q * R)}$$

Figure 4.19: Some Theorems in msl_ext/log_normalize.v

rems in Graph.v such as (4.20) and (4.21), since `vertices_at` is defined via `pred_sepcon` and `pred_sepcon` is defined via `iter_sepcon`. Although there are only four curves in Figure 4.16 from Graph.v to cite the four theorems in sector 18, it does not mean that other theorems are useless or irrelevant. The rest are either supporting theorems or used in other files outside the spatial graph library. We list some of these ramification theorems as follows.

$$\frac{\exists F\,.\begin{pmatrix}\texttt{Permutation}\,G\,(L+F) \wedge \texttt{Permutation}\,G'\,(L'+F) \wedge \\ \forall x\,.\,x \in F \to P(x) = P'(x)\end{pmatrix}}{\mathop{\Large *}_{G} P \vdash \mathop{\Large *}_{L} P * \left(\mathop{\Large *}_{L'} P' \mathrel{-\!\!*} \mathop{\Large *}_{G'} P'\right)} \tag{4.22}$$

In the theorem (4.22), G, G', L, L', F are lists and P, P' are spatial predicates. The expression $L_1 + L_2$ represents the concatenation of two lists L_1 and L_2. The relation $\texttt{Permutation}\,L_1\,L_2$ means that two lists L_1 and L_2 are permutations of each other. The premise says that L/L' is a sublist of G/G', the complement of L in G is the same as the complement of L' in G', and P, P' are the same on the common complementary list. The conclusion is our familiar ramification entailment. This is the crucial theorem in proving (4.20) and (4.21) with similar forms. Theorem (4.22) can be generalized with a pure logic predicate as an additional condition:

$$\frac{\exists F\,.\left(\begin{array}{c}\texttt{Permutation}\,G\,(L+F)\wedge\\ \Big(\forall a\,.\,M(a)\rightarrow\texttt{Permutation}\,G'(a)\,(L'(a)+F)\Big)\wedge\\ \forall a\,.\,M(a)\rightarrow\forall x\,.\,x\in F\rightarrow P(x)=P'(a,x)\end{array}\right)}{\mathop{\Huge\ast}_{G}P\vdash\mathop{\Huge\ast}_{L}P*\forall a\,.\,M(a)\rightarrow\Big(\mathop{\Huge\ast}_{L'(a)}P'(a)\mathrel{-\!\!*}\mathop{\Huge\ast}_{G'(a)}P'(a)\Big)} \tag{4.23}$$

In theorem (4.23), G, L, F are still lists while G' and L' are single-parameter functions that return lists. Similarly, P is a spatial predicate while P' is a function returning a spatial predicate. Another big difference between (4.23) and (4.22) is that (4.23) contains a pure logic predicate M. The application of this theorem is discussed in §5. Although we use the same symbol $\Huge\ast$ for `iter_sepcon` and `pred_sepcon`, theorem (4.22) and (4.23) are both about `iter_sepcon`. In ramification_lemmas.v we also provide the corresponding theorems about `pred_sepcon` with almost the same forms. They are listed as theorem (4.24) and (4.25) where G, G', L, L', F are pure logic predicates and `Permutation` is replaced by a ternary relation `PropJoin` defined in (4.26).

$$\frac{\exists F\,.\left(\begin{array}{c}\texttt{PropJoin}\,L\,F\,G\wedge\texttt{PropJoin}\,L'\,F\,G'\wedge\\ \forall x\,.\,F(x)\rightarrow P(x)=P'(x)\end{array}\right)}{\mathop{\Huge\ast}_{G}P\vdash\mathop{\Huge\ast}_{L}P*\Big(\mathop{\Huge\ast}_{L'}P'\mathrel{-\!\!*}\mathop{\Huge\ast}_{G'}P'\Big)} \tag{4.24}$$

$$\frac{\exists F\,.\left(\begin{array}{c}\texttt{PropJoin}\,L\,F\,G\wedge\\ \Big(\forall a\,.\,M(a)\rightarrow\texttt{PropJoin}\,L'(a)\,F\,G'(a)\Big)\wedge\\ \forall a\,.\,M(a)\rightarrow\forall x\,.\,F(x)\rightarrow P(x)=P'(a,x)\end{array}\right)}{\mathop{\Huge\ast}_{G}P\vdash\mathop{\Huge\ast}_{L}P*\forall a\,.\,M(a)\rightarrow\Big(\mathop{\Huge\ast}_{L'(a)}P'(a)\mathrel{-\!\!*}\mathop{\Huge\ast}_{G'(a)}P'(a)\Big)} \tag{4.25}$$

$$\begin{aligned}\texttt{PropJoin}\,X\,Y\,Z\overset{\text{def}}{=}{}&\Big(\forall a\,.\,Z(a)\leftrightarrow X(a)\vee Y(a)\Big)\wedge\\ &(\forall a\,.\,X(a)\rightarrow Y(a)\rightarrow\bot)\end{aligned} \tag{4.26}$$

It should be noted that the various theorems proved in ramification_lemmas.v also rely on a bunch of theorems about basic facts of the ramification entailment $K \vdash L * (M \twoheadrightarrow N)$ which were first proposed by Hobor and Villard [2013], formalized by us, and further integrated into VST. Some

$$\textsc{RamifSolve}\ \frac{G \vdash L * F \qquad F * L' \vdash G'}{G * L * (L' \twoheadrightarrow G')} \qquad \textsc{RamifFrame}\ \frac{G \vdash L * (L' \twoheadrightarrow G')}{G * F \vdash L * (L' \twoheadrightarrow G' * F)}$$

$$\textsc{RamifTrans}\ \frac{G \vdash M * (M' \twoheadrightarrow G') \qquad M \vdash L * (L' \twoheadrightarrow M')}{G \vdash L * (L' \twoheadrightarrow G')}$$

$$\textsc{RamifSplit}\ \frac{G_1 \vdash L_1 * (L_1' \twoheadrightarrow G_1') \qquad G_2 \vdash L_2 * (L_2' \twoheadrightarrow G_2')}{G_1 * G_2 \vdash (L_1 * L_2) * (L_1' * L_2' \twoheadrightarrow G_1' * G_2')}$$

$$\textsc{RamifFramePre}\ \frac{G \vdash L * (L' \twoheadrightarrow G')}{G \vdash L * (L' * F \twoheadrightarrow G' * F)} \qquad \textsc{RamifFramePost}\ \frac{G \vdash L * (L' \twoheadrightarrow G')}{G * F \vdash (L * F) * (L' \twoheadrightarrow G')}$$

Figure 4.20: Some Theorems in the Ramification Library

of them are listed in Figure 4.20. With the help of these theorems, we can break large ramification entailments into more manageable pieces in a compositional way.

Sector 20 (msl_ext/seplog.v) contains classes which compose the "Extended Logic Layer" of the spatial graph library in Figure 4.13.

The remaining three sectors 21, 22 and 23 under directory veric_ext are infrastructure which glues the VST library and our spatial library. From Figure 4.16 we can see that many theorems in veric_ext/seplog.v heavily rely on theorems in sector 9 and 12.

To summarize, we give a brief overview of the files in the spatial graph library through Figure 4.16 and introduce the theorems in each file and their relationship among these files. We list several representative theorems in compact, mathematical forms. This concise introduction expands the explanation of the architecture of the spatial library illustrated in Figure 4.13. Admittedly the theorems about $\uplus$, precise, direct model, etc. are not used in our verification of programs so far. They are currently included mostly for completeness, but do make our library more general. The spatial graph library together with the mathematical graph library enable us to prove the full functional correctness of real graph-manipulating programs. We demonstrate various verification in §5.

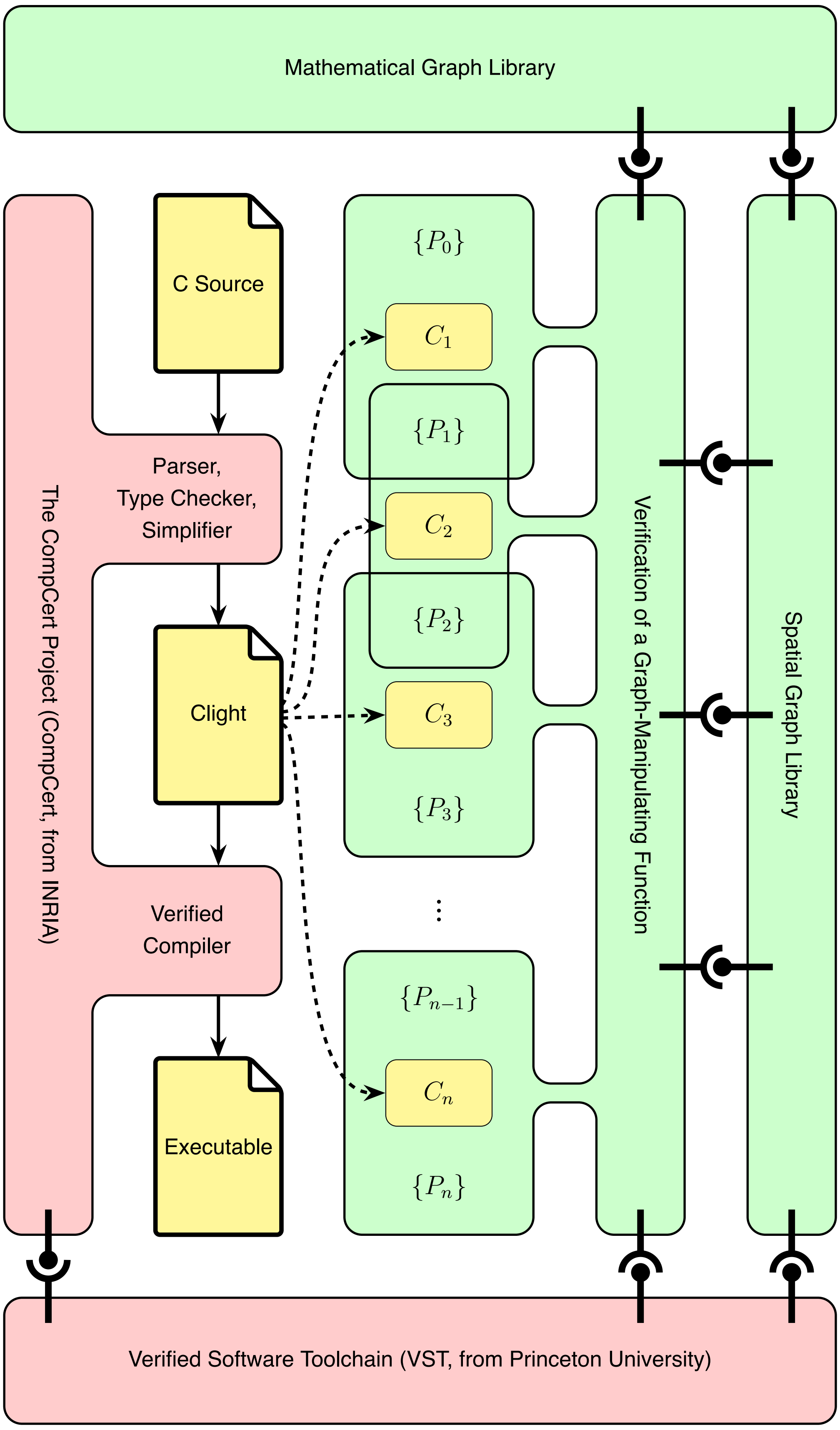
Chapter 5

Verification of Graph-manipulating Programs

In this chapter we introduce the application of our framework: the verification of full functional correctness of real C programs. In §5.1 we give an overall workflow of the verification. In §5.2 we illustrate the verification of a graph marking program. In §5.3 we focus on the verification of a spanning tree program. Unlike graph marking, this algorithm changes the shape of the graph, which makes the verification more difficult. In §5.4 we discuss verification of the classic union-find algorithm with two different data structures: one using pointers and the other using an array. In §5.5 we discuss our flagship application, the verification of a rather complex generational garbage collector. All these proofs are machine checked in Coq.

5.1 Workflow of Verification for C Programs

Figure 5.1 illustrates the workflow of the verification for a C function which manipulates graphs. Besides the mathematical graph library and spatial graph library that we described in previous sections, the verification (col-



Mathematical Graph Library
C Source
Parser,
Type Checker,
Simplifier
Clight
Verified
Compiler
Executable
The CompCert Project (CompCert, from INRIA)
$\{P_0\}$
C_1
$\{P_1\}$
C_2
$\{P_2\}$
C_3
$\{P_3\}$
$\{P_{n-1}\}$
C_n
$\{P_n\}$
Verification of a Graph-Manipulating Function
Spatial Graph Library
Verified Software Toolchain (VST, from Princeton University)

Figure 5.1: Workflow of Our Verification of C Programs

ored green) is also built on two external projects (colored pink): the CompCert project from INRIA [Leroy et al., 2012] and VST from Princeton University [Appel et al., 2014]. The workflow can be described as follows.

We start with a source file written in C, which is then parsed by CompCert's parser and saved as a Clight file after some type checking and simplification. The Clight file is a valid Coq file written in GALLINA by translating C expressions and C statements into corresponding GALLINA expressions of pre-defined, inductive types `expr` and `statement`. It can be seen as an intermediate representation of the C program. The translation is almost identical with respect to the abstract syntax tree. For example, the C conditional statement `if (...) {...} else {...}` is translated into an `Sifthenelse ...` expression of type `statement` listed below:

```
Inductive statement : Type :=
  | Sskip : statement
  | Sassign : expr -> expr -> statement
  | Sset : ident -> expr -> statement
  | Scall: option ident -> expr -> list expr -> statement
  | Ssequence : statement -> statement -> statement
  | Sifthenelse : expr -> statement -> statement -> statement
  ...
```

The only difference from C code is that all expressions are pure and assignments and function calls are statements, not expressions. In other words, side effects are pulled out of expressions in Clight. Although it is not necessary in our verification, the Clight file can be further processed by a multiple-stage compiler of CompCert and compiled to the executable machine code.

CompCert claims that the compilation is formally verified, i.e. free from miscompilation issues. There is a formal proof finished by the CompCert team saying that the executable code it produces behaves exactly as specified by the semantics of the source C program. As part of the proof, CompCert defines the formal semantics (i.e. relation between programs and their possible behaviors) for the C language and the assembly language for each of the supported target platforms (PowerPC, ARM, RISC-V and x86). However, this proof "just" guarantees that the observable behavior of the C program will be the same as that of the compiled assembly code. It does not tell one how to characterize the observable behavior of those C programs.

The Verified Software Toolchain (VST) project from Princeton gives a solution. It provides *Verifiable C*, a program logic for the C programming language. It is a higher-order impredicative separation logic, proved sound with respect to the operational semantics of CompCert C. In Figure 5.1, the link —(●— between the leftmost CompCert block and the bottom VST block indicates VST's adoption of the semantics of C defined in CompCert. With the help of verifiable C and certain tools in VST, one can give a specification written in separation logic formula for a C function and prove that the C function behaves exactly as the specification describes. As a whole, it is proved that [Appel et al., 2019],

> Whatever observable property about a C program you prove using the Verifiable C program logic, that property will actually hold on the assembly-language program that comes out of the C compiler.

Our verification is targeting the Clight file. Every C function in the Clight file is just a big expression of type `function` in Coq. After importing the Clight file, we give the specification (i.e. precondition and postcondition) of each function first. Then we can start the verification with the help of another component of VST, the Floyd program verification system.

Floyd provides a set of tactics to enable reasoning in forward style. As illustrated in Figure 5.1, a function is decomposed into a sequence of statements $C_1; C_2; C_3; \ldots; C_{n-1}; C_n$. We inspect these statements step by step. At each step, we are given a Hoare triple: a description of the current program state in the form of a separation logic formula $\{P_{i-1}\}$, the rest of the statement sequence $C_i; C_{i+1}; \ldots; C_n$, and the postcondition $\{P_n\}$. In the beginning, the current program state is the precondition $\{P_0\}$. For each statement C_i, we need to apply a tactic defined in Floyd to go through it. In principle, the tactic can be seen as the application of inference rules such as in Figure 2.1 and Figure 2.4 but under more delicate semantics. For many kinds of statements (assignments, return, break, continue), a `forward` tactic would automatically choose proper rules and derive the strongest postcondition $\{P_i\}$ from $\{P_{i-1}\}$. For the loop statement, the `forward_loop` tactic needs user to provide the loop invariant and postcondition. Sometimes due to the complex semantics of C, one may have to transform the current program state to another form through the entailment, so as to fulfill the requirement of certain tactics. Sometimes the application of a tactic would generate multiple proof goals. For example, one subgoal after applying the `forward_loop` tactic is proving that the postcondition of the loop body implies the loop invariant. Either the transformation of current program state or the entailment as subgoal may need theorems derived from or proved in our spatial graph library and mathematical graph library. That is why we can see the links among the corresponding blocks in Figure 5.1.

By progressively applying proper tactics and solving potential entailments, every Hoare triple $\{P_{i-1}\}\, C_i\, \{P_i\}$ would be proved. The overlapping of the Hoare triple block in Figure 5.1 implies that the description of the intermediate state $\{P_i\}$ must be chosen very carefully. It should be strong enough to prove the formula that succeeds it but still be weak enough to be derived from the formula that precedes it. The whole process can be seen as a repeated application of the COMPOSITION rule in Figure 2.1, which proves the specification of the whole function: $\{P_0\}\,F\,\{P_n\}$ where $F \overset{\text{def}}{=} C_1; C_2; C_3; \ldots; C_{n-1}; C_n$.

A complete verification workflow starts from a C source program and ends in the formal proofs of individual C functions. Since the the formal proof of C functions and the verified compiler in CompCert share the same semantics of C, we can say that the workflow facilitates the end-to-end verification of C programs.

5.2 Verification of Graph Mark

In Figure 5.2 we put the code and proof sketch of the classic `mark` algorithm that visits and colors every reachable vertex in a heap-represented graph. The `mark` program is good to start with because it is complex enough to require some care to verify while being simple enough that the invariants are straightforward.

The code in Figure 5.2 is a complete, valid C source program with the definition of `struct Node` and function `mark`. We put the description $\{P\}$ of the program state after the comment delimiter `//`. This mixed style of C code and program states is an imitation of the actual verification illustrated in Figure 5.2. We call it a paper-format *proof sketch*. All descriptions of the states are extracted from the real Floyd proof in VST, with minor cleanup to aid the presentation.

The `mark` function starts from a root vertex `x`. If `x` is a null pointer or a marked node, it returns. Otherwise, it marks `x` and continues marking its left and right branches recursively. Our task is to verify its correctness.

```
struct Node {
  int m;
  struct Node * l;
  struct Node * r;
};

void mark(struct Node * x) {
  struct Node * l, * r;
  int root_mark;
// $\{\mathsf{graph}(\mathtt{x}, \gamma)\}$
  if (x == 0) return;
// $\{\mathsf{graph}(\mathtt{x}, \gamma) \wedge \exists m, l, r \,.\, \gamma(\mathtt{x}) = (m, l, r)\}$
// $\{\mathsf{graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (m, l, r)\}$
// $\searrow \{\mathtt{x} \mapsto m, l, r\}$
$\lightning$(5.10) root_mark = x -> m;
// $\swarrow \{\mathtt{x} \mapsto m, l, r \wedge m = \mathtt{root\_mark}\}$
// $\{\mathsf{graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (m, l, r) \wedge m = \mathtt{root\_mark}\}$
  if (root_mark == 1) return;
// $\{\mathsf{graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
// $\searrow \{\mathtt{x} \mapsto 0, l, r \wedge \gamma(\mathtt{x}) = (0, l, r)\}$
      l = x -> l;
$\lightning$(5.11) r = x -> r;
      x -> m = 1;
// $\swarrow \{\mathtt{x} \mapsto 1, \mathtt{l}, \mathtt{r} \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \exists \gamma' \,.\, \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\}$
// $\{\exists \gamma' \,.\, \mathsf{graph}(\mathtt{x}, \gamma') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\}$
// $\{\mathsf{graph}(\mathtt{x}, \gamma') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma')\}$
// $\searrow \{\mathsf{graph}(\mathtt{l}, \gamma')\}$
$\lightning$(5.12) mark(l);
// $\swarrow \{\exists \gamma'' \,.\, \mathsf{graph}(\mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\}$
// $\left\{\begin{array}{c} \exists \gamma'' \,.\, \mathsf{graph}(\mathtt{x}, \gamma'') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'') \end{array}\right\}$
// $\{\mathsf{graph}(\mathtt{x}, \gamma'') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'')\}$
// $\searrow \{\mathsf{graph}(\mathtt{r}, \gamma'')\}$
$\lightning$(5.12) mark(r);
// $\swarrow \{\exists \gamma''' \,.\, \mathsf{graph}(\mathtt{r}, \gamma''') \wedge \mathit{mark}(\gamma'', \mathtt{r}, \gamma''')\}$
// $\left\{\begin{array}{c} \exists \gamma''' \,.\, \mathsf{graph}(\mathtt{x}, \gamma''') \wedge \gamma(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma'', \mathtt{r}, \gamma''') \end{array}\right\}$
} // $\{\exists \gamma''' \,.\, \mathsf{graph}(\mathtt{x}, \gamma''') \wedge \mathit{mark}(\gamma, \mathtt{x}, \gamma''')\}$
```

Figure 5.2: Clight Code and Proof Sketch for Graph Mark

5.2.1 The specification of the `mark` function

From the proof sketch we can get the specification we certify (lines 10 and 36 in Figure 5.2):

$$\{\mathsf{graph}(\mathsf{x},\gamma)\} \quad \texttt{mark(x)} \quad \{\exists\gamma'\,.\,\mathsf{graph}(\mathsf{x},\gamma') \wedge \mathit{mark}(\gamma,\mathsf{x},\gamma')\} \tag{5.1}$$

The specification is for fully functional correctness. It is stated using the *mathematical* graph γ, the *spatial* predicate graph describing how the mathematical graph γ is implemented in the heap, and the relation *mark* about the graphs before and after the execution of function `mark`. To understand this specification, one must know the definitions of these three concepts.

It should be noted that the struct `Node` defined in line 1–5 in Figure 5.2 is exactly the same as (4.3) discussed on page 82. The `mark` function runs on the binary graphs which are discussed in §4.1 as a detailed example to show how to represent a graph in heap. So it is easy to recall the definitions of γ and graph. The type of the mathematical graph γ is a specialized `GeneralGraph` whose concrete parameters are listed in Table 4.1 on page 93. The spatial predicate graph is defined in (4.7) on page 89. To see the source code level definition rather than the mathematical form, one can refer to `reachable_vertices_at` in Figure 4.7 of page 95. We should note that the definition of graph only represents in heap *those* parts of γ that are reachable from x, i.e. exactly those parts that the `mark` function operates on. The definitions of $\gamma(x)$ and $x \mapsto \gamma(x)$ involved in the definition of graph can be found in (4.12) and Figure 4.6.

Now we can introduce the definition of the last concept in the specification (5.1): *mark*. Before dive into the concrete definitions, we would like to explain the general principle behind it. It should be noted that the postcondition is specified *relationally*, i.e. $\{\exists\gamma'\,.\,\mathsf{graph}(\mathsf{x},\gamma') \wedge \mathit{mark}(\gamma,\mathsf{x},\gamma')\}$ instead of being specified *functionally*, i.e. $\{\mathsf{graph}\big(\mathsf{x},\mathit{mark}(\gamma,\mathsf{x})\big)\}$. In the first case *mark* is a relation which connects the original graph γ and the result graph γ', whereas in the second *mark* is a function that **computes** the result from the original graph γ. We prefer the relational approach because of its advantages in both theoretical and practical aspects. Theoretically, relation is a more general concept than function. A particular relation or function can be seen as a subset of the Cartesian product of "input" and "output". But relation has fewer constraints. For example, relations allow "inputs" to have no "outputs" (i.e. be partial) or alternatively have many outputs (i.e. be non-deterministic). In this sense, a function is just a special relation. For any function f, we can define a corresponding relation R: the relation $R(x, y)$ holds if and only if $y = f(x)$. On the contrary, there is no such direct and simple way to define a function for an arbitrary relation. Relations are also preferable to functions because they are more compositional. We take advantage of compositionality by using $mark(\gamma, x, \gamma') \wedge \ldots$ to also specify our "spanning tree" algorithm in §5.3, which also marks vertices while carrying out its primary tasks.

Practically, it is very difficult to define computational functions in Coq under certain situations. Writing functions over graphs is such a typical situation. Coq requires that all functions terminate. In our case, it means a nontrivial proof obligation over cyclic structures like graphs, which is overkill. Our verification of function `mark` is only for partial correctness. This is because partial correctness is what is guaranteed by the underlying system (VST). It appears to be challenging to use the step-indexed models used in VST to verify total correctness [Dockins and Hobor, 2012].

To formally define the relation *mark*, we first define two relations for a single vertex. The relation $lmarked(\gamma, v)$ means that vertex v is marked in graph γ and the other $\mathit{lunmarked}(\gamma, v)$ means that v is unmarked.

$$\mathit{lmarked}(\gamma, v) \overset{\text{def}}{=} \texttt{vlabel}\,\gamma\,v = \texttt{True} \tag{5.2}$$

$$\mathit{lunmarked}(\gamma, v) \overset{\text{def}}{=} \texttt{vlabel}\,\gamma\,v = \texttt{False} \tag{5.3}$$

Suppose after the execution of function `mark`, the graph γ becomes γ'. The function `mark` will have marked every vertex reachable from `x`. More formally we can conclude:

$$\begin{aligned} &\forall v \,.\, \mathit{lmarked}(\gamma', v) \leftrightarrow \mathit{lmarked}(\gamma, v) \lor \\ &\qquad\qquad \gamma \,\texttt{|=}\; \texttt{x} \;\texttt{\~{}o\~{}>}\; v \;\texttt{satisfying}\; \mathit{lunmarked}(\gamma) \end{aligned} \tag{5.4}$$

This is saying that for any vertex v which is marked in γ', it is either already marked in γ, or it is reachable from the root vertex `x`, along an unmarked path in γ. The definition of the notation $\gamma\,\texttt{|=}\; v_1 \;\texttt{\~{}o\~{}>}\; v_2 \;\texttt{satisfying}\; P$ can be found in Figure 3.3. Using (5.4) as the specification of the label part of relation *mark* is complete because of the double implication $\leftrightarrow$. However the definition of *mark* is not done yet. We need to point out the specification of the spatial part. One observation is that the function `mark` does not change anything that is unreachable from a unmarked path. In fact it does not change the shape of the structure at all. But for generality, let us just consider a formal statement about the unreachable part first.

$$\begin{aligned} &\gamma \uparrow \big(\lambda\, v. \neg\, \gamma \,\texttt{|=}\; \texttt{x} \;\texttt{\~{}o\~{}>}\; v \;\texttt{satisfying}\; \mathit{lunmarked}(\gamma)\big) \;\texttt{\~{}=\~{}} \\ &\gamma' \uparrow \big(\lambda\, v. \neg\, \gamma \,\texttt{|=}\; \texttt{x} \;\texttt{\~{}o\~{}>}\; v \;\texttt{satisfying}\; \mathit{lunmarked}(\gamma)\big) \end{aligned} \tag{5.5}$$

The notation $\gamma \uparrow P$ means part of a graph γ in which every vertex satisfies P. We discussed its formal definition in (3.1). The notation `~=~`, read as "structurally identical" is defined Figure 3.6. We combine (5.4) and (5.5) together to define a relation called *weak_mark*:

$$\begin{aligned} \mathit{weak\_mark}(\gamma, \mathtt{x}, \gamma') \stackrel{\text{def}}{=} \Big(&\forall v\,.\,\mathit{lmarked}(\gamma', v) \leftrightarrow \ \mathit{lmarked}(\gamma, v) \vee \\ &\gamma \,\mathtt{|=}\ \mathtt{x}\ \mathtt{\sim o\sim>}\ v\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma)\Big) \wedge \\ \Big(&\gamma \uparrow \big(\lambda\, v.\neg\,\gamma \,\mathtt{|=}\ \mathtt{x}\ \mathtt{\sim o\sim>}\ v\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma)\big)\ \mathtt{\sim=\sim} \\ &\gamma' \uparrow \big(\lambda\, v.\neg\,\gamma \,\mathtt{|=}\ \mathtt{x}\ \mathtt{\sim o\sim>}\ v\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma)\big)\Big) \end{aligned} \tag{5.6}$$

The definition of *weak_mark* captures the two key characters of a general marking procedure. It just says the labels of the reachable part are changed and the structure of the unreachable part is the same. It is named “weak” because it does not say anything about the structure of the reachable part, regardless of whether or not it is changed. So theorems about relation *weak_mark* can be used in both scenarios.

For our `mark` function in Figure 5.2, the reachable part from vertex `x` does not change. We could add this fact to define the relation *mark* but we did not. Instead we define *mark* as:

$$\mathit{mark}(\gamma, \mathtt{x}, \gamma') \stackrel{\text{def}}{=} \mathit{weak\_mark}(\gamma, \mathtt{x}, \gamma') \wedge \gamma\ \mathtt{\sim=\sim}\ \gamma' \tag{5.7}$$

In the definition of relation *mark*, we simply use $\gamma\ \mathtt{\sim=\sim}\ \gamma'$ since function `mark` does not change the shape of the graph structure in any way. It makes (5.7) an accurate definition already, even though there is a little redundancy because $\gamma\ \mathtt{\sim=\sim}\ \gamma'$ implies part of *weak_mark*, i.e. the part discussed in (5.5). Here we treat the relation *weak_mark* as a whole because we want to reuse the theorems about it directly in our verification of function `mark`. So far we fully explain the specification of the `mark` function. Next we turn to the proof about the specification.

5.2.2 The proof of the `mark` function

Lines 7–36 of Figure 5.2 give the body of the proof sketch. As illustrated in Figure 5.1, it can be seen as a description of a series of Hoare triples, i.e. state transitions between C statements. Inside the proof sketch, the consecutive descriptions of states represent entailments. For example, from line 12 to line 13, it is just the extraction of existential variables.

The proof sketch in Figure 5.2 omits entailment for simple branches of some conditional statements for simplicity. For example, line 17 is the precondition of line 18 while line 19 is just the postcondition of its "else" branch: the condition does not hold so $m = 0$. For the "then" branch, m should be 1 and the whole function returns. Our proof script does handle this case. Since the code after line 18 would never be executed in this "then" branch, we must entail the postcondition of `mark` directly from the current situation of "then" branch, i.e.

$$\mathsf{graph}(\mathsf{x}, \gamma) \wedge \gamma(\mathsf{x}) = (1, l, r) \vdash \exists \gamma' \,.\, \mathsf{graph}(\mathsf{x}, \gamma') \wedge \mathit{mark}(\gamma, \mathsf{x}, \gamma') \tag{5.8}$$

This is not a difficult entailment. The existential quantifier $\exists\gamma'$ could only be chosen as γ. Then the spatial entailment $\mathsf{graph}(\mathsf{x}, \gamma) \vdash \mathsf{graph}(\mathsf{x}, \gamma)$ is trivial. The pure part $\gamma(\mathsf{x}) = (1, l, r) \to \mathit{mark}(\gamma, \mathsf{x}, \gamma)$ is the only entailment which needs further explanation. According to (5.7), there are two parts in the definition of *mark*. One part is γ `~=~` γ, which is obviously true because `~=~` is reflexive. The other part, $\mathit{weak\_mark}(\gamma, \mathsf{x}, \gamma)$ can be further split into two components according to (5.6). The first, i.e. (5.5), holds again because of the reflexivity. The other (i.e. the only remaining obligation) is

$$\begin{aligned} &\forall v \,.\, \mathit{lmarked}(\gamma, v) \leftrightarrow \mathit{lmarked}(\gamma, v) \vee \\ &\qquad\qquad \gamma\,\texttt{|= x ~o~>}\ v\ \texttt{satisfying}\ \mathit{lunmarked}(\gamma) \end{aligned} \tag{5.9}$$

Since we have $\gamma(\mathsf{x}) = (1, l, r)$ which means x is marked, the right hand side of $\vee$ in (5.9) becomes $\perp$: no vertex is reachable from x along an unmarked path because its beginning has been marked already. Thus (5.9) becomes $\forall v \,.\, \mathit{lmarked}(\gamma, v) \leftrightarrow \mathit{lmarked}(\gamma, v) \vee \perp$ which is a tautology. So we have finished the simple proof of the entailment (5.8).

Other state transitions proved in our proof script may also involve entailments like (5.8), no matter whether they are omitted in Figure 5.2 or not. Various entailments depend on various semantics of C statements (i.e. inference rules like Figure 2.1). Besides lines 17–19 in Figure 5.2, there are 4 major entailments surrounded by the form $\{G_1\} \searrow \{L_1\}$ and $\{G_2\} \swarrow \{L_2\}$ (lines 13–17, lines 19–25, lines 26–30, and lines 31–35), which are localization blocks discussed in §4.2.1. We use the $\lightning$ symbol and the number followed to indicate the theorem number applied in corresponding blocks.

Now let us inspect the first localization block, lines 13–17 in Figure 5.2. Line 15, `root_mark= x -> m`, is just a simple memory load instruction which does not change any thing in the heap. But we still apply our Localize rule here. This is because the formal semantics of this load instruction requires an explicit form $\mathsf{x} \mapsto (\ldots, m, \ldots)$ in the precondition to ensure that the pointer x does point to a piece of memory which contains the field `m`. Our current spatial state is in line 13: $\mathsf{graph}(\mathsf{x}, \gamma)$. It is the Localize rule that allows us to have the local context $\mathsf{x} \mapsto m, l, r$ to go through the load instruction of line 15. As a cost, we have to prove the following theorem:

$$\frac{\mathtt{vvalid}\,\gamma\,\mathsf{x}}{\mathsf{graph}(\mathsf{x}, \gamma) \vdash \mathsf{x} \mapsto \gamma(\mathsf{x}) * \Big(\mathsf{x} \mapsto \gamma(\mathsf{x}) \mathrel{-\!\!*} \mathsf{graph}(\mathsf{x}, \gamma)\Big)} \tag{5.10}$$

The theorem (5.10) is the proof obligation of the Localize rule, which can be easily proved by the theorem (4.20) on page 112. The theorem (5.10) can be seen as a special case of (4.20) when two graphs γ, γ' are the same, two predicates P, P' are the same, and two vertices x, x' are the same.

The second localization block, lines 19–25, does more things than the previous one. Besides the two load instructions in lines 21 and 22, more importantly, line 23 is a memory store instruction which does change the heap. We still apply the LOCALIZE rule here because of the formal semantics of C. But we do not need to apply it for each statement. The statement C in the LOCALIZE rule in (4.13) can be a composition of several statements, just like this block. Under the new local state in line 20, the state transitions between the three instructions are omitted. They may involve some pure or spatial entailments, just like the entailments we explained for the `return` statement before. The proof obligation of this block is the theorem (5.11).

$$\frac{\forall x_0 \neq \mathsf{x} \,.\, \gamma(x_0) = \gamma'(x_0) \qquad \mathsf{neighbors}(\gamma, \mathsf{x}) = \mathsf{neighbors}(\gamma', \mathsf{x})}{\mathsf{graph}(\mathsf{x}, \gamma) \vdash \mathsf{x} \mapsto \gamma(\mathsf{x}) * \big(\mathsf{x} \mapsto \gamma'(\mathsf{x}) \mathrel{-\!\!*} \mathsf{graph}(\mathsf{x}, \gamma')\big)} \tag{5.11}$$

There are two premises for (5.11). The first one says that the graph γ and γ' are almost the same except vertex x. The second one says even for x, their neighbors are the same. This means the two graphs γ and γ' only differ in the data field part of x. This theorem can also be proved by the theorem (4.20). It is just another special case when two predicates P, P' are the same and two vertices x, x' are the same.

The third and fourth localization blocks (lines 26–30 and lines 31–35) are quite similar. They are both the recursive calls of the `mark` function, one for the left branch and the other for the right branch of a binary graph. For a function call, the state should match the precondition of the function specification. In these two cases, it means we need to extract the subgraph $\mathsf{graph}(\mathtt{l}, \gamma')$, $\mathsf{graph}(\mathsf{r}, \gamma'')$ from the whole graph $\mathsf{graph}(\mathsf{x}, \gamma')$,

$\mathsf{graph}(\mathtt{x}, \gamma'')$ respectively. We prove that both blocks can share the same proof obligation when applying the LOCALIZE rule. This is the theorem (5.12).

$$\frac{n \in \mathsf{neighbors}(\gamma, x)}{\mathsf{graph}(x,\gamma) \vdash \mathsf{graph}(n,\gamma) * \forall \gamma' \,.\, \mathit{mark}(\gamma, n, \gamma') \to \big(\mathsf{graph}(n,\gamma') \mathrel{-\!\!*} \mathsf{graph}(x,\gamma')\big)} \tag{5.12}$$

The idea of this theorem is the same as (5.11) but this time it is not the change of a single vertex but a subgraph. The validity of this theorem even depends on the relation *mark*. It is actually proved by the theorem (4.25).

So far we have finished the "spatial" proof of the specification of the `mark` function. But the verification is still not complete. What we proved so far can be explained as follows: a heap representation of graph γ, after a series of C statements, is transformed into a heap representation of graph γ'''. From line 35 we can see the relation between γ and γ''' is:

$$\exists \gamma', \gamma'' \,.\, \mathit{mark1}(\gamma, \mathtt{x}, \gamma') \wedge \mathit{mark}(\gamma', \mathtt{l}, \gamma'') \wedge \mathit{mark}(\gamma'', \mathtt{r}, \gamma''').$$

But the relation we expect is in line 36: $\mathit{mark}(\gamma, \mathtt{x}, \gamma''')$. So to finally finish the fully functional correctness verification of the `mark` function, we have to prove the following, pure mathematical theorem:

$$\frac{\gamma_0(\mathtt{x}) = (0, \mathtt{l}, \mathtt{r}) \quad \mathit{mark1}(\gamma_0, \mathtt{x}, \gamma_1) \quad \mathit{mark}(\gamma_1, \mathtt{l}, \gamma_2) \quad \mathit{mark}(\gamma_2, \mathtt{r}, \gamma_3)}{\mathit{mark}(\gamma_0, \mathtt{x}, \gamma_3)} \tag{5.13}$$

Before the discussion of the theorem 5.13, let us clarify a possible misunderstanding about the whole verification of the mark function. In the beginning the verification, we assume a specification of the `mark` function (we can call it Γ), which is necessary to reason about the recursive call of `mark`. Two premises $\mathit{mark}(\gamma_1, \mathtt{l}, \gamma_2)$ and $\mathit{mark}(\gamma_2, \mathtt{r}, \gamma_3)$ of the theorem

(5.13) come from Γ. When (5.13) is proved, we can claim that we proved Γ. It seems that we are assuming Γ in order to prove Γ, which is circular reasoning. But in fact it is sound. VST provides a rather sophisticated semantic model for the predicates in the heap. We are actually assuming Γ at *approximation level* n in order to prove Γ at approximation level $n + 1$. In the soundness proof of VST's program logic, there is a *Löb rule* (5.14) which is applied to ensure that we can assume the specification Γ in the proof that the function body meets Γ:

$$\frac{\triangleright\Gamma \vdash \Gamma}{\vdash \Gamma} \tag{5.14}$$

where $\triangleright\Gamma$ can be thought as Γ at approximation level n when Γ is thought as at approximation level $n + 1$. VST hides these technical details in the Floyd toolkit. For more information, one could refer to [Appel et al., 2014].

Now we can discuss the final missing piece in the verification of the `mark` function—the theorem (5.13). If γ_0 is a binary tree, then the theorem (5.13) is trivial, because the left and right branch of γ_0 are disjointed. In this situation, $\mathit{mark1}(\gamma_0, \mathtt{x}, \gamma_1)$ means the root `x` is marked, $\mathit{mark}(\gamma_1, \mathtt{l}, \gamma_2)$ means the left branch is marked, and $\mathit{mark}(\gamma_2, \mathtt{r}, \gamma_3)$ means the right branch is marked. Then we have $\mathit{mark}(\gamma_0, \mathtt{x}, \gamma_3)$: every vertex reachable from the root is marked.

But for a graph, the situation is more complex because its left and right branches could overlap. For any vertex v which is reachable from the root `x`, there are 3 situations. If it is the root `x`, then it is marked because we have $\mathit{mark1}(\gamma_0, \mathtt{x}, \gamma_1)$. If it is reachable from `l`, i.e. in the left branch, then it is marked because we have $\mathit{mark}(\gamma_1, \mathtt{l}, \gamma_2)$. If it is reachable from `r`, then there are two cases. Either it is also reachable from `l` so that it is marked already, or it is marked because of $\mathit{mark}(\gamma_2, \mathtt{r}, \gamma_3)$. This is the basic idea

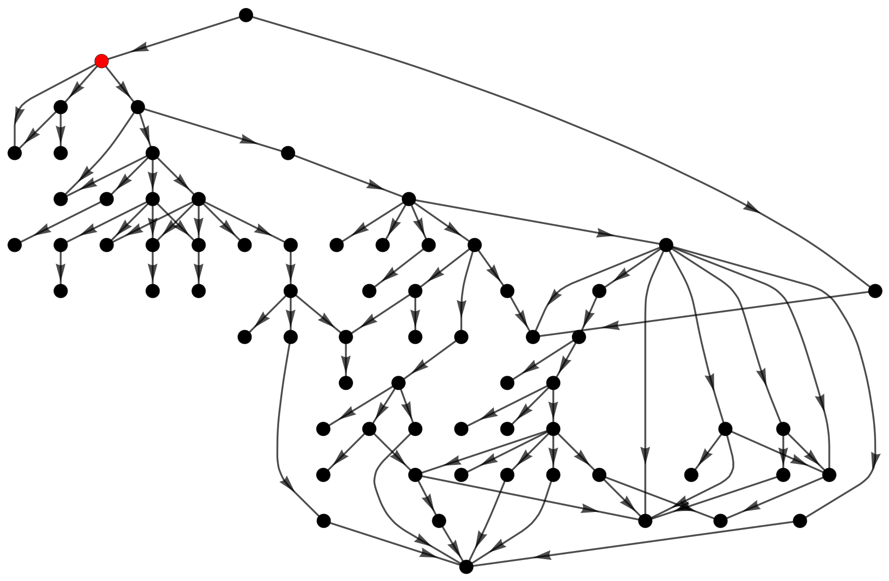


Figure 5.3: Dependency Graph of the Theorem (5.13)

but we must reason rigorously according to the definition of the relation *mark* in (5.7). Figure 5.3 illustrates the dependency graph of the theorem (5.13). There are 68 theorems involved. From the figure we can see that the red node is the key theorem. This is theorem (5.15), a more general version of (5.13):

$$\frac{\gamma_0(\mathsf{x}) = (0, n_1, n_2, \ldots, n_{m_x}) \qquad \mathit{mark1}(\gamma_0, \mathsf{x}, \gamma_1) \qquad \mathit{mark}(\gamma_1, n_1, \gamma_2) \qquad \mathit{mark}(\gamma_2, n_2, \gamma_3) \qquad \ldots \qquad \mathit{mark}(\gamma_{m_x}, n_{m_x}, \gamma_{m_x+1})}{\mathit{mark}(\gamma_0, \mathsf{x}, \gamma_{m_x+1})} \tag{5.15}$$

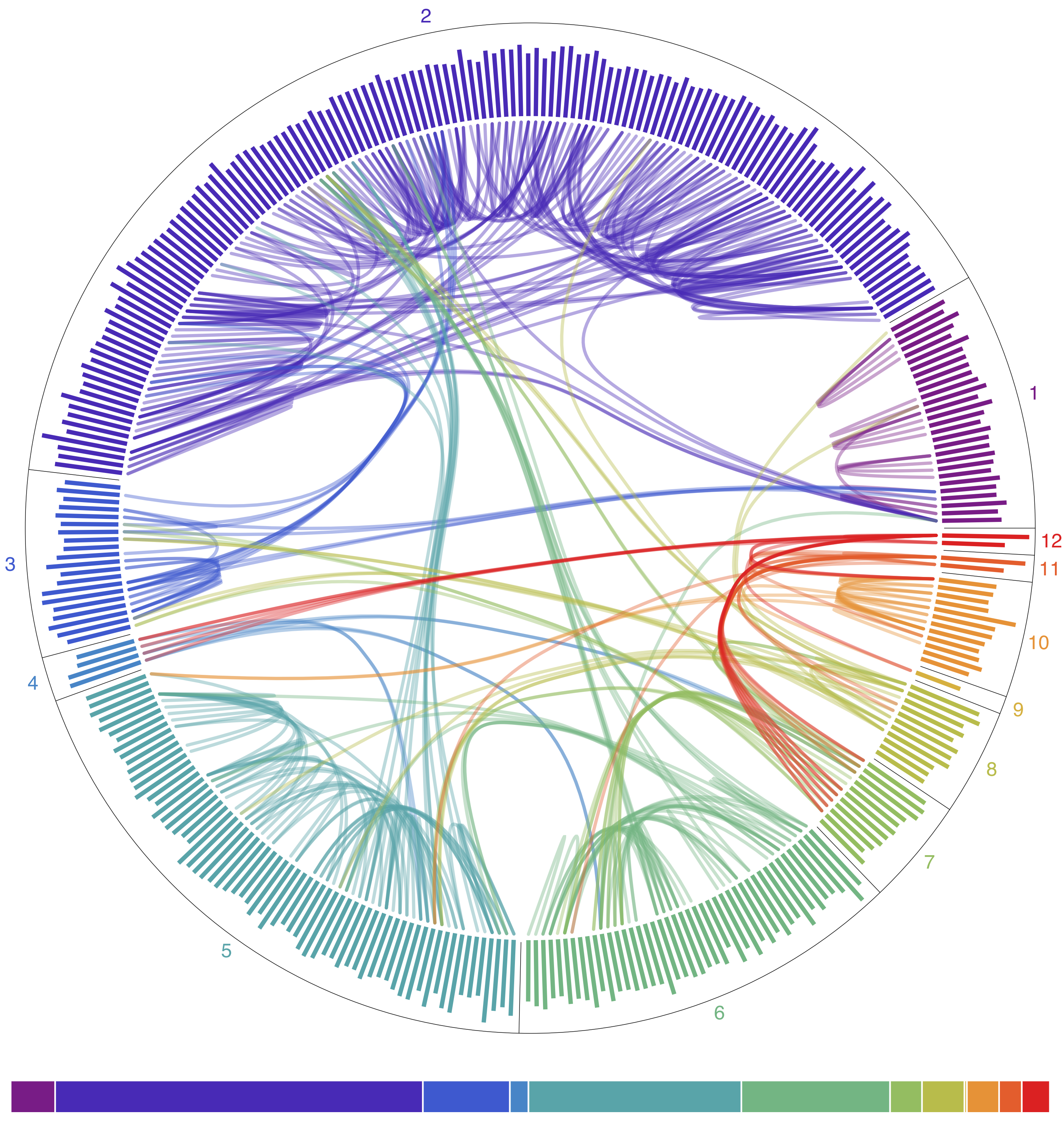
In the theorem (5.15), every vertex of the graph γ_0 can have several neighbors. The numbers of neighbors of vertices are not necessarily the same. This theorem says that we can start marking from every neighbor of the root vertex in turn, then the whole graph (reachable part from root) is marked. As illustrated in Figure 5.3, the formal proof of this theorem is rather complicated, involving many supporting theorems.

5.2.3 Theorems in verifying the `mark` function

Figure 5.4 on page 137 gives an overview of the theorems in the verification of marking programs. Unlike Figure 3.7 or Figure 4.16, it contains some files belonging to libraries that we have discussed previously. For example, sectors 1, 2, and 3 are classified to the mathematical graph library. Sectors 5, 6, and 8 are classified to the spatial library. These files are included in Figure 5.4 to reveal their support in verifying graph marking programs.

The proof script which verifies the `mark` function is in sector 11, the file named sample_mark/verif_mark_bi.v. All entailments we discussed about Figure 5.2 are done in this file. From Figure 5.4 we can see that it depends heavily on sector 7, msl_application/GraphBi_Mark.v, which contains all theorems needed in the verification, like (5.8), (5.10), (5.11), (5.12) and (5.13) discussed in §5.2.2. We can further observe that sector 7 adopts sectors 2, 3, 5, 6 and 8. This fact exhibits that the theorems in verifying the `mark` function do rely on our mathematical and spatial graph library. For example, the theorem (5.15) applied to prove (5.13) is in sector 8.

Sector 12 (sample_mark/verif_mark_bi_dag.v) is another proof script which verifies the same `mark` function but using a different specification: the graph in memory is not merely a binary graph but a binary directed acyclic graph (DAG). At first glance it seems that this is just a special case of the binary graph so we could reuse the proof. But since we do add new requirement to the spatial predicate graph for the DAG, the corresponding ramification theorems like (5.11) and (5.12) do need special effort to prove that the new requirement is preserved. They are proved in sector 4. The other entailments are almost the same as a general binary graph. We reuse the theorem (5.13) in the final pure math entailment. Because this verification is quite similar to §5.2.2, we omit the decorated program like Figure 5.2 and further detailed explanation.



■ 1: graph/graph_model.v
■ 2: graph/path_lemmas.v
■ 3: graph/weak_mark_lemmas.v
■ 4: msl_application/DagBi_Mark.v
■ 5: msl_application/Graph.v
■ 6: msl_application/GraphBi.v
■ 7: msl_application/GraphBi_Mark.v
■ 8: msl_application/Graph_Mark.v
■ 9: sample_mark/env_mark_bi.v
■ 10: sample_mark/spatial_graph_bi_mark.v
■ 11: sample_mark/verif_mark_bi.v
■ 12: sample_mark/verif_mark_bi_dag.v

Figure 5.4: Theorems in the Verification of Marking Programs

Sector 10 (sample_mark/spatial_graph_bi_mark.v) contains implementations of interfaces to connect the spatial graph library and our verification, as discussed in Table 4.1. Instances in it are used by sectors 11 and 12. The implementation in sector 9 connect VST and our verification.

Figure 5.4 even show the connection between the mathematical graph library and the spatial graph library. There are many curves from sectors 5, 6 to sectors 1 and 2. Recall that the color of the curve is the same as the color of the caller. We read from this figure to know that the spatial library does depend on the mathematical graph library.

5.3 Verification of Spanning Tree

In Figure 5.5 we show a simplified proof sketch for a spanning tree program. Unlike graph marking, the spanning tree program changes the structure of the graph, leading to a more complicated specification, in both the pure part and the spatial part. On the other hand, the two programs have many similarities. For example, they both mark vertices through unmarked paths. They even share the same data structure `Node`, which means we can reuse the definitions of the graph γ, the spatial predicate graph, and the relation *weak_mark*.

Just like the `mark` function, the `spanning` function starts from a root vertex `x`. It assumes that `x` is valid and unmarked. After marking `x`, it inspects the left and right node pointers in turn. If the pointer (referred as i) is null, it does nothing. Otherwise, there are two situations. If the pointed node (referred as p) is not marked yet, the function calls itself to i to get the spanning tree from i. If p is marked, it means p has been visited before. In other words, there is a path from the root vertex to p other than the current one. So the function just sets i to null. The pointer i does not point to p any more. After the function call, there is one and only one way to connect every reachable node from the root, which is the definition of a spanning tree.

5.3.1 The specification of the `spanning` function

From line 4 and line 37 in Figure 5.5, we can see the specification is:

$$\begin{gathered}\{\mathsf{graph}(\mathtt{x}, \gamma) \wedge \gamma(\mathtt{x}).1 = 0\} \\ \mathtt{spanning(x)} \\ \{\exists \gamma_3 \,.\, \mathsf{vertex\_at}(\mathtt{reachable}\,\gamma\,\mathtt{x}, \gamma_3) \wedge \mathit{span}(\gamma, \mathtt{x}, \gamma_3)\}\end{gathered} \tag{5.16}$$

where $\gamma(\mathtt{x}).1 = 0$ means the first item of the triple is 0. We can see that the same specification appears in the recursive call of line 24 and 26. In the precondition, the definitions of γ and graph are the same as specified in §5.2.1. In the postcondition, according to the definition of $\mathsf{vertices\_at}$ in Figure 4.7 and the definitions of `reachable` in Figure 3.3, we have:

$$\mathsf{vertices\_at}(\mathtt{reachable}\,\gamma\,\mathtt{x}, \gamma_3) = \mathop{\Huge *}_{\mathtt{reachable}\,\gamma\,\mathtt{x}\,v} v \mapsto \gamma_3(v) \tag{5.17}$$

Superficially it is a strange choice that we do not use $\mathsf{graph}(\mathtt{x}, \gamma_3)$ but (5.17) as the spatial representation in the postcondition. They are the same if:

$$\forall v \,.\, \mathtt{reachable}\,\gamma\,\mathtt{x}\,v \leftrightarrow \mathtt{reachable}\,\gamma_3\,\mathtt{x}\,v \tag{5.18}$$

According to the behavior of the `spanning` function, for a graph γ and the resulting spanning tree γ_3, the proposition (5.18) holds only when the reachable part of γ from `x` is totally unmarked. For the graph from `x` and the subgraph from the left child of `x`, there is no problem. But after the first recursive call, when the function starts dealing with the right subgraph, the situation changes. Figure 5.6 gives a typical situation during the execution of the `spanning` function. The left graph is the state after marking the

```
struct Node { int m; struct Node * l; struct Node * r; };
// We use $R$ to represent $\mathit{reachable}(\gamma, \mathsf{x})$
void spanning(struct Node * x) {
// $\{\mathsf{graph}(\mathsf{x}, \gamma) \wedge \gamma(\mathsf{x}).1 = 0\}$
    struct Node * l, * r; int root_mark;
// $\{\mathsf{graph}(\mathsf{x}, \gamma) \wedge \exists l, r \,.\, \gamma(\mathsf{x}) = (0, l, r)\}$
// $\{\mathsf{graph}(\mathsf{x}, \gamma) \wedge \gamma(\mathsf{x}) = (0, l, r)\}$
// $\{\mathsf{vertices\_at}(\mathit{reachable}(\gamma, \mathsf{x}), \gamma) \wedge \gamma(\mathsf{x}) = (0, l, r)\}$
// $\{\mathsf{vertices\_at}(R, \gamma) \wedge \gamma(\mathsf{x}) = (0, l, r)\}$
// $\searrow \{\mathsf{x} \mapsto 0, l, r \wedge \gamma(\mathsf{x}) = (0, l, r)\}$
    l = x -> l; r = x -> r; x -> m = 1;
// $\swarrow \{\mathsf{x} \mapsto 1, \mathtt{l}, \mathtt{r} \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \exists \gamma_1 \,.\, \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1)\}$
// $\{\exists \gamma_1 \,.\, \mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1)\}$
// $\{\mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1)\}$
    if (l) {
// $\left\{\begin{array}{c} \mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \exists m_2, l_2, r_2 \,.\, \gamma_1(\mathtt{l}) = (m_2, l_2, r_2) \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \end{array}\right\}$
// $\{\mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma_1(\mathtt{l}) = (m_2, l_2, r_2) \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1)\}$
// $\searrow \{\mathtt{l} \mapsto m_2, l_2, r_2\}$
        root_mark = l -> m;
// $\swarrow \{\mathtt{l} \mapsto m_2, l_2, r_2 \wedge m_2 = \mathtt{root\_mark}\}$
// $\left\{\begin{array}{c} \mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma_1(\mathtt{l}) = (m_2, l_2, r_2) \wedge \\ m_2 = \mathtt{root\_mark} \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \end{array}\right\}$
        if (root_mark == 0) {
// $\{\mathsf{vertices\_at}(R, \gamma_1) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma_1(\mathtt{l}) = (0, l_2, r_2) \wedge \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1)\}$
// $\searrow \{\mathsf{graph}(\mathtt{l}, \gamma_1) \wedge \gamma_1(\mathtt{l}) = (0, l_2, r_2)\}$
            spanning(l);
// $\swarrow \{\exists \gamma_2 \,.\, \mathsf{vertices\_at}(\mathit{reachable}(\gamma_1, \mathtt{l}), \gamma_2) \wedge \gamma_1(\mathtt{l}) = (0, l_2, r_2) \wedge \mathit{span}(\gamma_1, \mathtt{l}, \gamma_2)\}$
// $\left\{\begin{array}{c} \exists \gamma_2 \,.\, \mathsf{vertices\_at}(R, \gamma_2) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \gamma_1(\mathtt{l}) = (0, l_2, r_2) \wedge \\ \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \wedge \mathit{span}(\gamma_1, \mathtt{l}, \gamma_2) \end{array}\right\}$
        } else { x -> l = 0; } }
// $\left\{\begin{array}{c} \exists \gamma_2 \,.\, \mathsf{vertices\_at}(R, \gamma_2) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, (\mathsf{x}, \mathsf{L}), \gamma_2) \end{array}\right\}$
// $\left\{\begin{array}{c} \mathsf{vertices\_at}(R, \gamma_2) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, (\mathsf{x}, \mathsf{L}), \gamma_2) \end{array}\right\}$
    if (r) {
        root_mark = r -> m;
        if (root_mark == 0) {
            spanning(r);
        } else { x -> r = 0; } }
// $\left\{\begin{array}{c} \exists \gamma_3 \,.\, \mathsf{vertices\_at}(R, \gamma_3) \wedge \gamma(\mathsf{x}) = (0, \mathtt{l}, \mathtt{r}) \wedge \\ \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \wedge \mathit{e\_span}(\gamma_1, (\mathsf{x}, \mathsf{L}), \gamma_2) \wedge \mathit{e\_span}(\gamma_2, (\mathsf{x}, \mathsf{R}), \gamma_3) \end{array}\right\}$
} // $\{\exists \gamma_3 \,.\, \mathsf{vertices\_at}(\mathit{reachable}(\gamma, \mathsf{x}), \gamma_3) \wedge \mathit{span}(\gamma, \mathsf{x}, \gamma_3)\}$
```

Figure 5.5: Clight Code and Proof Sketch for Spanning Tree

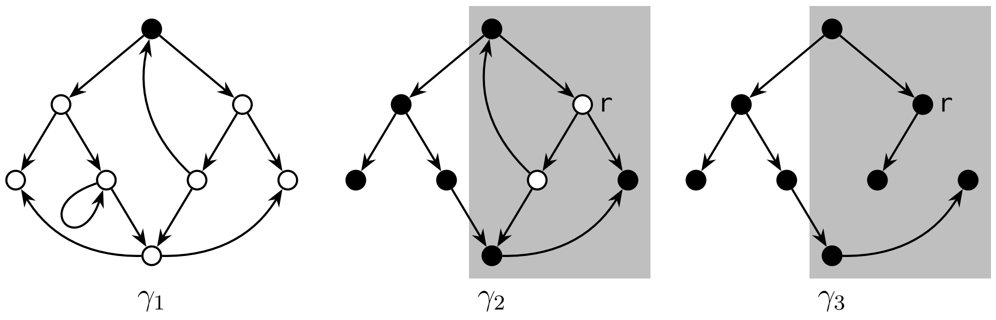


Figure 5.6: A Binary Graph During `spanning`

root vertex. The middle one is the state after the first recursive call. The right one is the final result. The middle one is the most interesting. The big gray rectangle encloses all vertices reachable from r, i.e. $\mathsf{graph}(\mathsf{r}, \gamma_2)$. We can see some of the vertices are marked. So after the second recursive call, all edges to those marked vertices are removed. Those vertices become unreachable from r. As we can see, $\mathsf{graph}(\mathsf{r}, \gamma_3)$ only contains 2 vertices. Since the specification (5.16) would be used in the second recursive call, if we adopt $\mathsf{graph}(\mathsf{r}, \gamma_3)$ in postcondition, we will lose 3 vertices. On the other hand, (5.17) is quite suitable. It means the vertices in the heap after the execution of `spanning` is the same as before. At the same time, the concrete mapping is different because it is $v \mapsto \gamma_2(v)$, not $v \mapsto \gamma_1(v)$. So (5.17) fits in all situations, regardless of whether the graph is partially marked or not.

Now we can turn to the definition of the relation $span(\gamma, \mathsf{x}, \gamma_3)$, which is more complicated than the relation *mark*. First, it should contain the relation $weak\_mark(\gamma, \mathsf{x}, \gamma_3)$ defined in (5.6): the reachable part for any unmarked path is marked and the unreachable part keeps the same. After the execution, we must get a tree, which is the purpose of the spanning tree algorithm. Thus we define a predicate *is_tree*:

$$\begin{aligned} is\_tree(\gamma, x) &\overset{\text{def}}{=} \forall y\, .\, \mathtt{reachable}\,\gamma\,x\,y \rightarrow \\ &\qquad \exists!\, p \text{ s.t. } \gamma \,\mathtt{|=}\, p \;\mathtt{is}\;\mathtt{x}\;\mathtt{\sim o\sim>}\; v\;\mathtt{satisfying}\;\top \end{aligned} \tag{5.19}$$

This says graph γ starting from x is a tree, if for any vertex y which is reachable from x in γ, there exists only one path from x to y. This is exactly the characteristic of a tree. We cannot say $\mathit{is\_tree}(\gamma_3, \mathtt{x})$ because the unreachable part of γ_3 does not change and we do not know its shape. We can only say the *reachable* part from x is a tree:

$$\mathit{is\_tree}\Big(\gamma_3 \uparrow \big(\lambda\, v \,.\, \gamma\,\mathtt{|=\ x\ \sim o\sim>}\ v\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma)\big), \mathtt{x}\Big) \tag{5.20}$$

The definition of *span* is not complete yet because *is_tree* only partially describes the feature of the shape: it is a tree but it may not be a spanning tree. At first we thought the following extra condition would be enough:

$$\forall v \,.\, \gamma\,\mathtt{|=\ x\ \sim o\sim>}\ v\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma) \to \mathtt{reachable}\,\gamma_3\,\mathtt{x}\,v \tag{5.21}$$

which says that any vertex v reachable from x along an unmarked path in γ is reachable from x in γ_3. But during the proof, we found that it is insufficient to entail line 37 from line 36. We need one extra condition:

$$\begin{aligned} &\forall a, b \,.\, \gamma\,\mathtt{|=\ x\ \sim o\sim>}\ a\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma) \to \\ &\qquad \neg\gamma\,\mathtt{|=\ x\ \sim o\sim>}\ b\ \mathtt{satisfying}\ \mathit{lunmarked}(\gamma) \to \\ &\qquad \neg\mathtt{reachable}\,\gamma_3\,a\,b \end{aligned} \tag{5.22}$$

which says if a is reachable from x along an unmarked path in γ but b is not, then in γ_3, b is unreachable from a. If (5.21) can be interpreted as "we do not remove more edges" then (5.22) can be interpreted as "we do not add extra edges".

Finally we can combine *weak_mark*, (5.20), (5.21), and (5.22) together to get the complete definition of the relation *span*, which is rather long:

$$\begin{gathered}
\mathit{span}(\gamma, \mathtt{x}, \gamma') \overset{\text{def}}{=} \mathit{weak\_mark}(\gamma, \mathtt{x}, \gamma') \land \Big(\mathit{lunmarked}(\gamma, \mathtt{x}) \rightarrow \\
\mathit{is\_tree}\big(\gamma' \uparrow \big(\lambda\, v\,.\, \gamma\, \texttt{|=}\ \mathtt{x}\ \texttt{\~{}o\~{}>}\ v\ \texttt{satisfying}\ \mathit{lunmarked}(\gamma)\big), \mathtt{x}\big) \land \\
\forall v\,.\, \gamma\, \texttt{|=}\ \mathtt{x}\ \texttt{\~{}o\~{}>}\ v\ \texttt{satisfying}\ \mathit{lunmarked}(\gamma) \rightarrow \texttt{reachable}\, \gamma'\, \mathtt{x}\, v\Big) \land \\
\Big(\forall a, b\,.\, \gamma\, \texttt{|=}\ \mathtt{x}\ \texttt{\~{}o\~{}>}\ a\ \texttt{satisfying}\ \mathit{lunmarked}(\gamma) \rightarrow \\
\neg\gamma\, \texttt{|=}\ \mathtt{x}\ \texttt{\~{}o\~{}>}\ b\ \texttt{satisfying}\ \mathit{lunmarked}(\gamma) \rightarrow \neg\texttt{reachable}\, \gamma'\, a\, b\Big)
\end{gathered} \tag{5.23}$$

Such a long definition makes our entailment about the mathematical graph extremely difficult. But it is necessary. We formally proved that if γ, γ_3 satisfy $\mathit{span}(\gamma, \mathtt{x}, \gamma')$ and γ is totally unmarked, then (5.18) holds. This is also a side proof about the soundness of the definition in (5.23).

5.3.2 The proof of the `spanning` function

Since the explanation of the proof script about the `mark` function in §5.2.2 is very comprehensive, we will omit similar details when explaining other proof scripts. From a higher perspective of view, the entailment of the state transition is just routine work. Although sometimes it is difficult to prove that the current state satisfies certain conditions required by the complicated semantics of C language, it can always be proved with patience once the right postcondition is determined. In other words, it is more important to know what to prove than simply to attack the proof blindly.

Figure 5.5 is a simplified proof sketch of the `spanning` function. We omit the description of states between line 31 and line 35 because they are similar to the specifications between line 15 and line 28. Most state transitions from line 4 to line 37 in Figure 5.5 are similar to the proof of `mark` except line 28 and line 35, which cuts edges in a graph.

We define a relation `gremove_edge` $\gamma_1\, e\, \gamma_2$ to represent the relation between the initial graph γ_1 and the resulting graph γ_2 after cutting an edge e in γ_1 as follows.

```
Definition gremove_edge (g1: PreGraph Vertex Edge) (e0: Edge)
                        (g2: PreGraph Vertex Edge) :=
  (forall v : Vertex, (vvalid g1 v <-> vvalid g2 v)) /\
  (forall e : Edge, e <> e0 -> (evalid g1 e <-> evalid g2 e)) /\
  (forall e : Edge, e <> e0 -> evalid g1 e -> evalid g2 e ->
                  src g1 e = src g2 e) /\
  (forall e : Edge, e <> e0 -> evalid g1 e -> evalid g2 e ->
                  dst g1 e = dst g2 e) /\
  ((~ evalid g2 e0) \/ (~ vvalid g2 (dst g2 e0) /\
                        src g1 e0 = src g2 e0 /\ evalid g2 e0)).
```

This definition is quite easy to understand. It is just a conjunctive proposition of 5 conjuncts. The first four say that other parts of `g2` except the edge `e0` remain the same. The last conjunct gives two possibilities: either `e0` is invalid in the resulting `g2` or the destination of `e0` is invalid in `g2`. We adopt the following notation:

$$\mathit{cut}(\gamma_1, e, \gamma_2) \overset{\text{def}}{=} \texttt{gremove\_edge}\ \gamma_1\ e\ \gamma_2 \tag{5.24}$$

So after line 28, we have two possibilities for the relation between γ_1 and γ_2: γ_2 is either the resulting graph after the execution of the `spanning` function or the result of cutting an edge. We describe such a relation as *e_span*:

$$e\_span(\gamma_1, e, \gamma_2) \overset{\text{def}}{=} \begin{cases} \mathit{cut}(\gamma_1, e, \gamma_2) & \texttt{vlabel}\ \gamma_1\ (\texttt{dst}\ \gamma_1\ e) = 1 \\ \mathit{span}(\gamma_1, \texttt{dst}\ \gamma_1\ e, \gamma_2) & \texttt{vlabel}\ \gamma_1\ (\texttt{dst}\ \gamma_1\ e) = 0 \end{cases} \tag{5.25}$$

which says if the destination of edge e is marked, then the relation is *cut*, otherwise it is *span*. With the definition of *e_span*, we unify the description of states after the conditional statement. Just like the ramification theorems (5.11) and (5.12) in the verification of `mark`, the ramification theorems of `spanning` can also be proved easily by applying theorems in the spatial library. We also omit them. When the spatial entailment is done in line 36, the entailment from line 36 to line 37 is the only crucial one left. The spatial part of line 36 to line 37 are the same. So it is a pure entailment:

$$\frac{\begin{array}{c}\gamma(\mathsf{x}) = (0, \mathsf{l}, \mathsf{r}) \\ \mathit{mark1}(\gamma, \mathsf{x}, \gamma_1) \qquad \mathit{e\_span}(\gamma_1, (\mathsf{x}, \mathsf{L}), \gamma_2) \qquad \mathit{e\_span}(\gamma_2, (\mathsf{x}, \mathsf{R}), \gamma_3)\end{array}}{\mathit{span}(\gamma, \mathsf{x}, \gamma_3)} \tag{5.26}$$

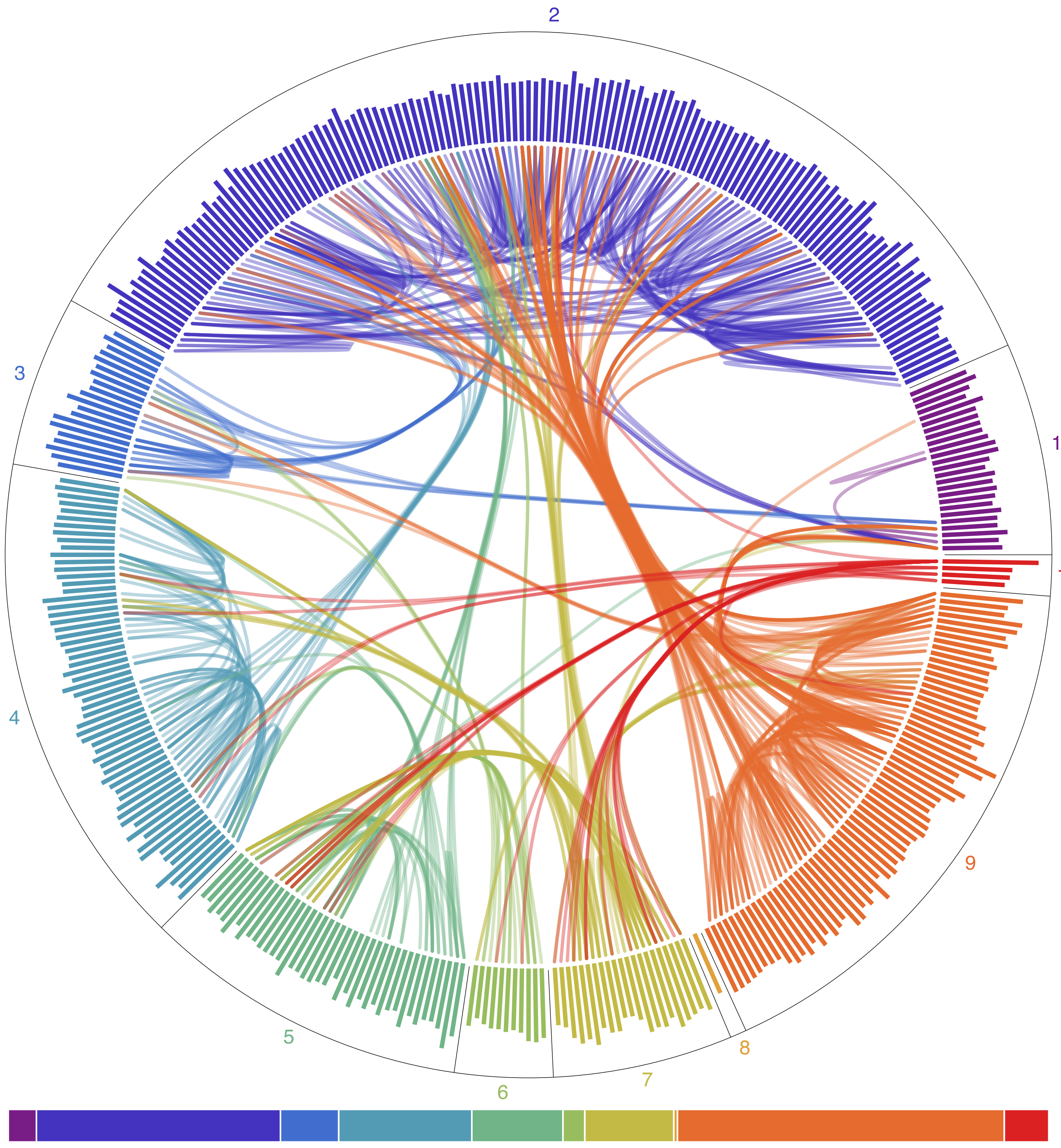
Recall the long definition of *span* in (5.23), the proof of (5.26) is extremely difficult. Since each *e_span* contains two cases, we need to discuss four cases in proving (5.26). The strength of *span* is also important because *e_span* is *span* under certain conditions. If the conclusion *span* is too strong, it is hard to prove. If the premise *span* is too weak, it is insufficient to get the conclusion. As we can see before, we modified our definition once (5.22) to get a proper definition of *span*. The whole proof script for theorem (5.26) contains more than 1700 lines of code.

5.3.3 Theorems in verifying the `spanning` function

Figure 5.7 on page 146 gives an overview of the theorems in the verification of the spanning tree program. Just like Figure 5.4, it contains various files from the mathematical and spatial graph libraries (sector 1–6) to show their support in the verification of the `spanning` function.

Sector 7 (data_structure/spatial_graph_dispose_bi.v) contains many theorems needed in verifying the `spanning` function, including the ramification theorems and the pure entailment (5.26).

Sector 8 (data_structure/spatial_graph_unaligned_bi_VST.v) implements vari-



- ■ 1: graph/graph_model.v
- ■ 2: graph/path_lemmas.v
- ■ 3: graph/weak_mark_lemmas.v
- ■ 4: msl_application/Graph.v
- ■ 5: msl_application/GraphBi.v
- ■ 6: msl_application/GraphBi_Mark.v
- ■ 7: data_structure/spatial_graph_dispose_bi.v
- ■ 8: data_structure/spatial_graph_unaligned_bi_VST.v
- ■ 9: graph/spanning_tree.v
- ■ 10: sample_mark/verif_dispose_bi.v

Figure 5.7: Theorems in the Verification of Spanning Tree Program

ous interfaces to connect VST, the spatial library and our verification.

Sector 9 (graph/spanning_tree.v) contains the definitions of *is_tree* (5.19), *span* (5.23), *e_span* (5.25), etc. and various theorems about these concepts. Although the theorem (5.26) is presented in sector 7, but its proof is very short because a series of supporting theorems in proving it are finished in sector 9. Sector 9 has more than 60 theorems. From Figure 5.7 we can see that these theorems depend extensively on sector 2 for the theorems about path and reachability. Sector 3 (graph/weak_mark_lemmas.v) which is used in the verification of the `mark` function, is also reused in the pure entailment about spanning tree.

Sector 10 (sample_mark/verif_dispose_bi.v) is the proof script which verifies the `spanning` function. From Figure 5.7 we can see that it depends on sectors 4–7. Sector 5 (msl_application/GraphBi.v) is about binary graph. Sector 6 (msl_application/GraphBi_Mark.v) is about spatial inference of graph marking. They are not as general as sector 4 (msl_application/Graph.v) but they are both used in the verification of `mark` and `spanning`.

5.4 Verification of Union-Find

The verification of the graph marking and spanning tree programs demonstrates the ability of our framework established by the mathematical and spatial graph libraries. To further show that our framework is not limited to the binary graph, we decide to verify a classical algorithm—the union-find algorithm [Cormen et al., 2009].

This is not the first formal verification of the union-find algorithm. For example, Charguéraud and Pottier [2015, 2019] verify the correctness and the worst-case amortized asymptotic complexity of an OCaml implementation of Union-Find twice via different complexity analysis techniques. They extend the tool CFML which implements separation logic for OCaml to support time credits, which is further developed and used to verify a cycle-detection algorithm [Guéneau et al., 2019].

We do not verify the time complexity of union-find. But we do verify several variants of the algorithm, including different specifications (comprehensive and slim ones), different data structures (pointer and array), and even different control flows (recursion and loop).

The union-find algorithm is proposed to perform two operations on a collection of disjoint sets efficiently. One is finding the unique set that contains a given element. The other is uniting two sets. Figure 5.8 gives

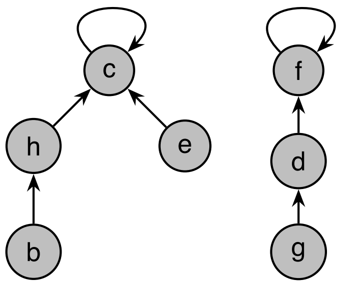


Figure 5.8: The Disjoint-Set Data Structure

an example of the data structure adopted by the algorithm. Each vertex v representing an element has one and only one edge pointing to another vertex called the *parent* of v. The vertex pointing to itself is the *representative/root* of a set. Figure 5.8 illustrates two disjoint sets. One is represented by `c` and the other is by `f`. It can be modeled as a graph with several connected components. Given an element, the `find` function traces the edges until it reaches the root. Two vertices belong to the same set if and only they have the same root. During the tracing progress, it may change the structure of the graph to make tracing easier next time. Given two elements which are not necessarily the representatives, the `unionS` function (which is named to avoid conflicts with the C keyword `union`) employs `find` to find the roots of the two elements respectively. And then it redirects the parent of one root to the other to form a bigger set. Compared to `find`, the `unionS` function is relatively straightforward. It just calls `find` twice and redirects one edge. So its verification should be easier than `find`.

5.4.1 The definition of γ and `graph`

Before turning to the concrete specifications of the two functions `find` and `unionS`, we need to specify the type of mathematical graph γ and the spatial predicate `graph` first.

Unlike `mark` and `spanning`, γ—the underlying graph model of the union-find algorithm—is no longer a binary graph. We introduce a new predicate of graph called `LstGraph` to model the unique property of the data structure used in the union-find algorithm:

```
Class LstGraph (pg: PreGraph Vertex Edge)
                    (out_edge: Vertex -> Edge): Prop := {
    only_one_edge: forall x e, vvalid pg x ->
        (src pg e = x /\ evalid pg e <-> e = out_edge x);
    no_loop_path: forall x p,
        pg |= p is x ~o~> x satisfying (fun _ => True) ->
        p = (x, nil) }.
```

The `LstGraph` takes a function `out_edge` to map a vertex to its only out edge. As its name indicates, the component `only_one_edge` says that, for any valid vertex, there is one and only one valid edge from it. The other component `no_loop_path` captures the acyclic property of the graph. But from Figure 5.8 we can see that `no_loop_path` holds for every vertex `x` except for the roots. The roots do have self-pointing loops. However, after a careful analysis, we find that the self-pointing edge of a root is only used to identify the end of a path. It is totally possible to have a graph model with null pointers as ends of paths, as long as the corresponding spatial representation has self-pointing edges for roots. The conversion from a null pointer to a self pointing vertex could be easily done in the definition of the vertex information function $\gamma(x)$.

Besides enabling `no_loop_path` established for every valid vertex, another benefit of using null pointers to identify ends of paths is that we can add `MathGraph` as part of the soundness condition of γ. Thus, a lot of theorems about `MathGraph` could be reused. In fact, we define a compositional graph predicate `LiMaFin` as the soundness condition for the graph model γ of the union-find algorithm.

```
Class LiMaFin (g: PreGraph V E) := {
    li: LstGraph g out_edge;
    ma: MathGraph g isNullDec;
    fin: FiniteGraph g }.
```

Table 5.1 gives all determined types for the γ used in the verification of union-find except the type for `Vertex`. We defer the instantiation of `Vertex`

Vertex	Edge	DV	DE	DG	Sound
V	V * unit	nat	unit	unit	LiMaFin

Table 5.1: Instantiated Types of Graph Model of Union-Find

because it does not affect the inference about the pure facts of γ and we need the theorems to be general enough to be used in the verification of the union-find algorithms with two different data structures. Since each vertex only has one out edge, we specify `Edge` type almost identical to `Vertex` so as to have a one-to-one correspondence.

The two data structures refer to the definitions of the vertex in C. One is line 1 of the proof sketch in Figure 5.9 which uses pointers to link vertices. The other is line 1 of the proof sketch in Figure 5.11 which uses integers to link vertices. The type of `Vertex`—`V`—is `pointer_val` in the former case and integer `Z` in the latter case. Both data structures contain a `rank` field which is used in the `unionS` function. So we specify the type of vertex label as natural number `nat` in Table 5.1. Then we can define the vertex information function $\gamma(v)$:

$$\gamma(v) \overset{\text{def}}{=} \left(\texttt{vlabel}\ \gamma\ v,\ \begin{matrix} \texttt{if dst}\ \gamma\ (\texttt{out\_edge}\ v) = 0\ \texttt{then}\ v \\ \texttt{else dst}\ \gamma\ (\texttt{out\_edge}\ v) \end{matrix} \right) \tag{5.27}$$

Here the `if...then...else...` statement plays the trick we mentioned before to covert a null-pointer-ending graph to a self-reference-ending graph.

```
struct Node { unsigned int rank; struct Node * parent; }
// $\{\mathsf{graph}(\gamma) \wedge \texttt{vvalid}\ \gamma\ \mathsf{x}\}$
struct Node* find(struct Node* x) {
  struct Node *p;
// $\{\mathsf{graph}(\gamma) \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \exists r, pa\,.\,\gamma(\mathsf{x}) = (r, pa) \wedge \texttt{vvalid}\ \gamma\ pa\}$
// $\searrow \{\mathsf{x} \mapsto r, pa \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \wedge \texttt{vvalid}\ \gamma\ pa\}$
  p = x -> parent;
// $\swarrow \{\mathsf{x} \mapsto r, pa \wedge \mathsf{p} = pa \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \wedge \texttt{vvalid}\ \gamma\ pa\}$
// $\{\mathsf{graph}(\gamma) \wedge \mathsf{p} = pa \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \wedge \texttt{vvalid}\ \gamma\ pa\}$
  if (p != x) {
// $\{\mathsf{graph}(\gamma) \wedge \mathsf{p} = pa \wedge pa \neq \mathsf{x} \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \wedge \texttt{vvalid}\ \gamma\ pa\}$
    p = find(p);
// $\left\{\begin{matrix} \exists \gamma', rt\,.\,\mathsf{graph}(\gamma') \wedge \mathsf{p} = rt \wedge pa \neq \mathsf{x} \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \\ \mathit{uf\_equiv}(\gamma, \gamma') \wedge \mathit{uf\_root}(\gamma', pa, rt) \wedge \gamma(\mathsf{x}) = (r, pa) \end{matrix}\right\}$
// $\searrow \left\{\begin{matrix} \mathsf{x} \mapsto r, pa \wedge \mathsf{p} = rt \wedge pa \neq \mathsf{x} \wedge \mathit{uf\_equiv}(\gamma, \gamma') \wedge \\ \mathit{uf\_root}(\gamma', pa, rt) \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \end{matrix}\right\}$
    x -> parent = p;
// $\swarrow \left\{\begin{matrix} \mathsf{x} \mapsto r, rt \wedge \mathsf{p} = rt \wedge pa \neq \mathsf{x} \wedge \mathit{uf\_equiv}(\gamma, \gamma') \wedge \\ \mathit{uf\_root}(\gamma', pa, rt) \wedge \texttt{vvalid}\ \gamma\ \mathsf{x} \wedge \gamma(\mathsf{x}) = (r, pa) \end{matrix}\right\}$
// $\{\exists \gamma''\,.\,\mathsf{graph}(\gamma'') \wedge \mathit{uf\_equiv}(\gamma, \gamma'') \wedge \mathit{uf\_root}(\gamma'', \mathsf{x}, rt) \wedge \mathsf{p} = rt\}$
  } return p;
} // $\{\exists \gamma'', rt\,.\,\mathsf{graph}(\gamma'') \wedge \mathit{uf\_equiv}(\gamma, \gamma'') \wedge \mathit{uf\_root}(\gamma'', \mathsf{x}, rt) \wedge \mathsf{ret} = rt\}$
```

Figure 5.9: Clight Code and Proof Sketch of `find`

From Figure 5.8 it is easy see that the definition (4.7) of the spatial predicate **graph** which is the reachable part from a certain vertex is not suitable anymore. We could only get a path if we use (4.7). Considering the `unionS` function which joins two sets, we need a spatial predicate to represent the whole collection of sets, i.e. the whole graph.

The data structure used in Figure 5.9 is still a collection of pointers.

We could use the `full_vertices_at` in Figure 4.7 to define graph. Its mathematical form is in (5.28).

$$\mathsf{graph}(\gamma) \overset{\text{def}}{=} \mathop{\bigast}_{\texttt{vvalid}\ \gamma\ v} v \mapsto \gamma(v) \tag{5.28}$$

The data structure used in Figure 5.10 is totally different. The parameter of the `unionS` function reveals that the spatial representation of a graph should be an array. It is defined in (5.29)

$$\begin{gathered}\mathsf{graph}(\gamma, \mathsf{x}) \overset{\text{def}}{=} \exists n\,.\,\Big(\big(\forall v\,.\,0 \leq v < n \leftrightarrow \texttt{vvalid}\ \gamma\ v\big) \wedge \\ \big(n \leq \texttt{MaxInt}/8\big) \wedge \mathsf{array\_at}\big(\mathsf{x}, [\gamma(0), \gamma(1), \gamma(2), \ldots, \gamma(n)]\big)\Big)\end{gathered} \tag{5.29}$$

where x is not the root vertex of a graph but the address of the graph array. The spatial predicate array_at is defined in VST which represents a mapping from an address to an array. The condition $n \leq \texttt{MaxInt}/8$ is required by the restriction of index arithmetic in C code.

We should note that the parameter `isNullDec` of `MathGraph` is also affected by chosen data structure. In the pointer version it is defined to compare with the null pointer. But in the array version, we define it to decide whether the parameter is less than zero.

5.4.2 The specification of `find` and `unionS`

The Figure 5.9 and Figure 5.11 give the simplified proof sketches of the `find` function. Their specifications look like the same:

$$\begin{gathered}\{\mathsf{graph}(\gamma) \wedge \texttt{vvalid}\ \gamma\ \mathsf{x}\} \\ \texttt{find(x)} \\ \{\exists \gamma'', rt\,.\,\mathsf{graph}(\gamma'') \wedge \mathit{uf\_equiv}(\gamma, \gamma'') \wedge \mathit{uf\_root}(\gamma'', \mathsf{x}, rt) \wedge \mathsf{ret} = rt\}\end{gathered} \tag{5.30}$$

```
void unionS(struct Node* x, struct Node* y) {
// $\{\mathsf{graph}(\gamma) \wedge \mathtt{vvalid}\ \gamma\ \mathsf{x} \wedge \mathtt{vvalid}\ \gamma\ \mathsf{y})\}$
    struct Node *xRt, *yRt; int xRank, yRank;
    xRt = find(x);
// $\{\exists \gamma', t.\ \mathsf{graph}(\gamma') \wedge \mathit{uf\_equiv}(\gamma, \gamma') \wedge \mathit{uf\_root}(\gamma', \mathsf{x}, t) \wedge \mathsf{xRt} = t\}$
// $\{\mathsf{graph}(\gamma_1) \wedge \mathtt{vvalid}\ \gamma_1\ \mathsf{y} \wedge \mathit{uf\_equiv}(\gamma, \gamma_1) \wedge \mathit{uf\_root}(\gamma_1, \mathsf{x}, \mathsf{xRt})\}$
    yRt = find(y);
// $\{\exists \gamma', t.\ \mathsf{graph}(\gamma') \wedge \mathit{uf\_equiv}(\gamma_1, \gamma') \wedge \mathit{uf\_root}(\gamma', \mathsf{y}, t) \wedge \mathsf{yRt} = t\}$
// $\{\mathsf{graph}(\gamma_2) \wedge \mathit{uf\_equiv}(\gamma_1, \gamma_2) \wedge \mathit{uf\_root}(\gamma_2, \mathsf{x}, \mathsf{yRt})\}$
    if (xRt == yRt) { return; }
// $\{\mathsf{graph}(\gamma_2) \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma_2)\}$
// $\{\exists \gamma'.\ \mathsf{graph}(\gamma') \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma')\}$
    xRank = xRt -> rank; yRank = yRt -> rank;
    if (xRank < yRank) { xRt -> parent = yRt;
// $\{\gamma_3 = \mathit{redir\_prnt}(\gamma_2, \mathsf{xRt}, \mathsf{yRt}) \wedge \mathsf{graph}(\gamma_3)) \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma_3)\}$
    } else if (xRank > yRank) { yRt -> parent = xRt;
// $\{\gamma_3 = \mathit{redir\_prnt}(\gamma_2, \mathsf{yRt}, \mathsf{xRt}) \wedge \mathsf{graph}(\gamma_3)) \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma_3)\}$
    } else {yRt -> parent = xRt; xRt -> rank = xRank + 1;}};
// $\{\gamma_3 = \mathit{redir\_prnt}(\gamma_2, \mathsf{yRt}, \mathsf{xRt}) \wedge \mathsf{graph}(\gamma_3)) \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma_3)\}$
// $\{\exists \gamma'.\ \mathsf{graph}(\gamma') \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma')\}$
```

Figure 5.10: Clight Code and Proof Sketch of `unionS`

They do share the same relation *uf_equiv* and *uf_root*. The only difference is the spatial predicate graph. This specification says that after the execution of `find`, the graph γ becomes γ''. The graphs γ and γ'' are sort of equivalent and the root of the given vertex x is the return value rt. Here ret means the return value. The definitions of the relation *uf_root* and *uf_equiv* are listed below:

$$\mathit{uf\_root}(\gamma, \mathsf{x}, t) \overset{\text{def}}{=} \mathtt{reachable}\ \gamma\ \mathsf{x}\ t \wedge \forall y\,.\ \mathtt{reachable}\ \gamma\ t\ y \rightarrow y = t \quad (5.31)$$

$$\begin{aligned}\mathit{uf\_equiv}(\gamma_1, \gamma_2) \overset{\text{def}}{=}\ & (\forall x\,.\ \mathtt{vvalid}\ \gamma_1\ x \leftrightarrow \mathtt{vvalid}\ \gamma_2\ x) \wedge \\ & \forall x, r_1, r_2\,.\ \mathit{uf\_root}(\gamma_1, x, r_1) \rightarrow \\ & \mathit{uf\_root}(\gamma_2, x, r_2) \rightarrow r_1 = r_2 \end{aligned} \quad (5.32)$$

The relation *uf_root* says that a vertex t is the root of x in γ if t is reachable from x in γ and, for any other vertex y which is reachable from t, y must be t. It is a rational definition because the root vertex cannot point to other vertices. The relation *uf_equiv* says that γ_1 and γ_2 may have different structures but for any vertex x, the roots of x in γ_1 and γ_2 are the same.

The specification of the function `unionS` in Figure 5.10 is listed below:

$$\begin{gathered}\{\mathsf{graph}(\gamma) \wedge \texttt{vvalid}\ \gamma\ \texttt{x} \wedge \texttt{vvalid}\ \gamma\ \texttt{y})\} \\ \texttt{unionS(x, y)} \\ \{\exists \gamma' \,.\, \mathsf{graph}(\gamma') \wedge \mathit{uf\_union}(\gamma, \mathsf{x}, \mathsf{y}, \gamma')\}\end{gathered} \tag{5.33}$$

which says after the execution of `unionS`, graph γ becomes γ'. The graphs γ and γ' satisfy a relation *uf_union*, which is rather complicated:

$$s \subset \gamma \overset{\text{def}}{=} s = \emptyset \vee \exists r \,.\, r \in s \wedge \Big(\forall v \,.\, v \in s \leftrightarrow \mathit{uf\_root}(\gamma, v, r)\Big) \tag{5.34}$$

$$\begin{gathered}\mathit{uf\_union}(\gamma_1, v_1, v_2, \gamma_2) \overset{\text{def}}{=} \Big(\forall s_1, s_2 \,.\, v_1 \in s_1 \to v_2 \in s_2 \to \\ s_1 \subset \gamma_1 \to s_2 \subset \gamma_1 \to s_1 \cup s_2 \subset \gamma_2\Big) \wedge \\ \Big(\forall s \,.\, s \neq s_1 \to s \neq s_2 \to s \subset \gamma_1 \to s \subset \gamma_2\Big) \wedge \\ \Big(\forall s \,.\, s \subset \gamma_2 \to s = (s_1 \cup s_2) \vee s \subset \gamma_1\Big)\end{gathered} \tag{5.35}$$

We first define a notation $s \subset \gamma$ to represent a statement: a set s contains exactly all those vertices whose roots are the same in the graph γ. Next we can explain the definition of the relation *uf_union*. The definition (5.35) is a long conjunction. The first conjunct says that if v_1 belongs to set s_1, v_2 belongs to set s_2, and we have both $s_1 \subset \gamma_1$ and $s_2 \subset \gamma_1$, then after the execution of `unionS`, both sets are united together, i.e. $s_1 \cup s_2 \subset \gamma_2$. The second conjunct says that other sets except s_1 and s_2 remain the same. The third conjunct says that there are no new sets in γ_2.

The definition (5.34) formalizes the concept "disjoint sets in a graph". The 3 conjuncts in the definition (5.35) employ (5.34) to formalize the concept "uniting two disjoint sets".

```
struct subset { int parent; int rank; };
int find(struct subset s[], int i) {
// $\{\mathsf{graph}(\gamma, \mathtt{s}) \land \mathtt{vvalid}\ \gamma\ \mathtt{i}\}$
    int p0 = 0;
    int p = s[i].parent;
    if (p != i) {
       p0 = find(s, p);
// $\{\exists \gamma', t\,.\,\mathsf{graph}(\gamma', \mathtt{s}) \land \mathit{uf\_equiv}(\gamma, \gamma') \land \mathit{uf\_root}(\gamma', \mathtt{p}, t) \land \mathtt{p0} = t\}$
// $\{\mathsf{graph}(\gamma_1, \mathtt{s}) \land \mathit{uf\_equiv}(\gamma, \gamma_1) \land \mathit{uf\_root}(\gamma_1, \mathtt{p}, \mathtt{p0})\}$
        p = p0;
        s[i].parent = p; }
// $\left\{\begin{array}{l} \gamma_2 = \mathit{redir\_prnt}(\gamma_1, \mathtt{i}, \mathtt{p}) \land \mathsf{graph}(\gamma_2, \mathtt{s})) \land \\ \mathit{uf\_equiv}(\gamma, \gamma_2) \land \mathit{uf\_root}(\gamma, \mathtt{i}, \mathtt{p}) \end{array}\right\}$
    return p; }
// $\{\exists \gamma', t\,.\,\mathsf{graph}(\gamma', \mathtt{s}) \land \mathit{uf\_equiv}(\gamma, \gamma') \land \mathit{uf\_root}(\gamma', \mathtt{x}, t) \land \mathsf{ret} = t\}$
```

Figure 5.11: Clight Code and Proof Sketch of `find` (array version)

5.4.3 The proof of `find` and `unionS`

The verification of the `find` function is rather straightforward once we have the right definitions of relations. In the pointer version proof sketch of `find` in Figure 5.9, we mark the localization block as before via $\searrow$ and $\swarrow$. The ramification entailments of them are just applications of theorems such as (4.20) and (4.25). The spatial entailments are done in these theorems. We just need to prove the spatial predicates in concrete verification do satisfy the premises of those theorems. In the array version proof sketch of `find` in Figure 5.11, the whole graph is stored in an array. The spatial predicate graph is defined in (5.29). We can refer to a vertex by the index of the array. Because of this definition, we do not use any ramification entailment in the spatial part of the proof. We can prove the facts about arrays directly. However, both versions of verification of the `find` function share the same pure fact entailments because they share the same pure relations.

The only thing we do not mention is the function $\mathit{redir\_prnt}(\gamma, x, y)$ which returns a new graph by changing the parent of vertex x to y in graph γ. Intuitively it is quite easy but the real definition is too complex to be listed here, because the graph we are manipulating must obey the rules defined by `LiMaFin`. So the definition of *redir_prnt* contains proofs about preservation of `LiMaFin`. We cannot find it in Figure 5.9, but it is used in the proof. The existential quantifier in line 17 hides a *redir_prnt*. It is used in all our proofs to represent the result after redirection. The function *redir_prnt* uses dependent types just because we bind property `LiMaFin` in `GeneralGraph`. If we use `LabeledGraph` instead, dependent types is not necessary, although we still need to prove `LiMaFin` at some point.

The proof of `unionS` is much easier than `find` because it is not recursive. The major difficulty is proving the following pure fact:

$$\frac{\mathit{uf\_root}(\gamma, x, r_x) \qquad \mathit{uf\_root}(\gamma, y, r_y) \qquad r_x \neq r_y}{\mathit{uf\_union}\big(\gamma, x, y, \mathit{redir\_prnt}(\gamma, r_x, r_y)\big)} \tag{5.36}$$

5.4.4 Theorems in the verification of Union-Find

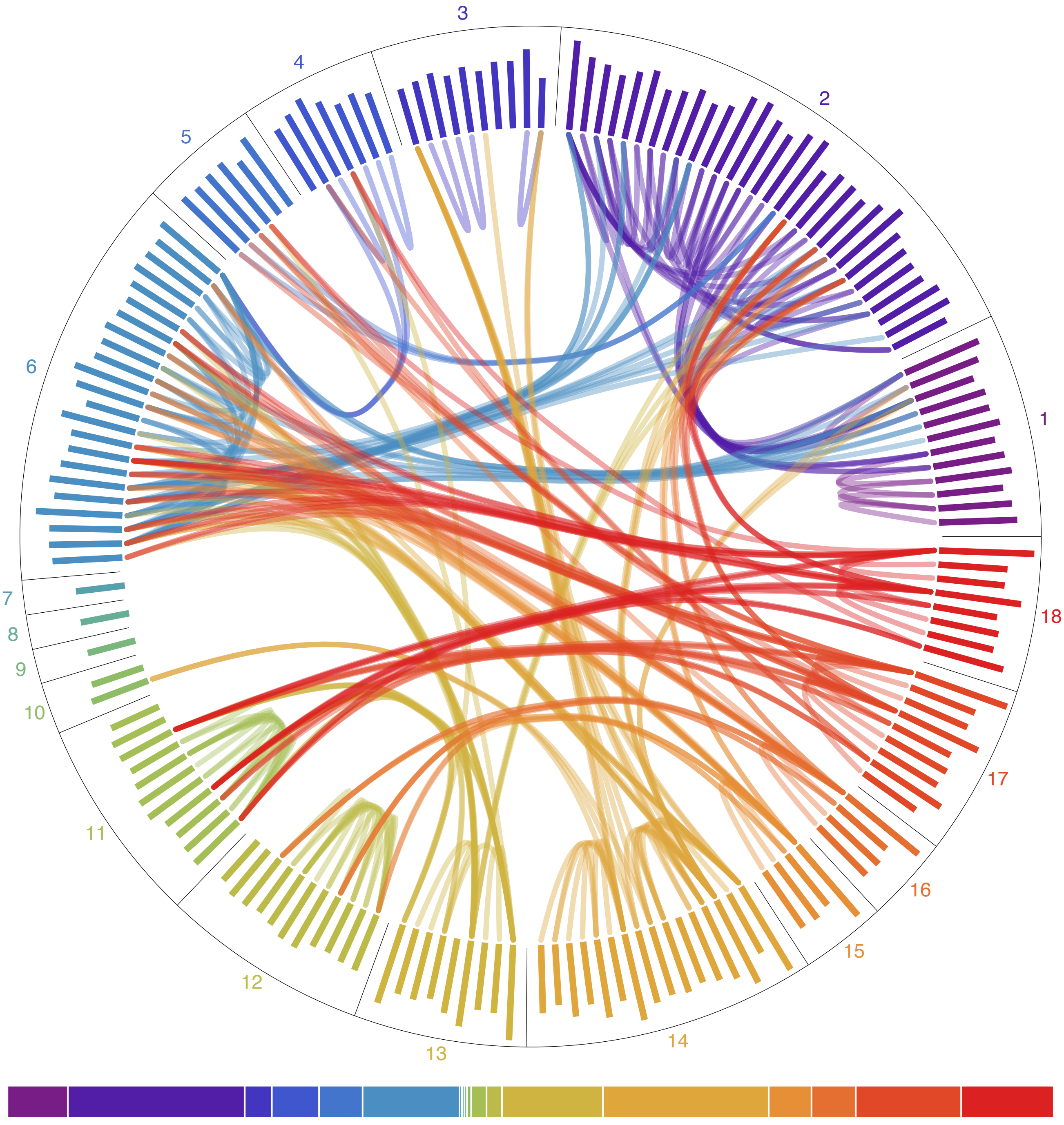
Figure 5.12 on page 157 gives an overview of the theorems in the verification of several variants of the union-find algorithm. Unlike Figure 5.4 or Figure 5.7, it does not contain files in math or spatial graph libraries.

Sector 1 (graph/LstGraph.v) contains the definition of `LstGraph` and basic facts about it.

Sector 2 (graph/UnionFind.v) contains the definitions of *uf_root*, *uf_equiv*, *uf_union* and etc. These relations are based on `PreGraph`. Sector 2 also contains theorems directly about these relations. In Figure 5.12 we can see that this file is widely cited by other proof scripts.

Sector 3 (msl_application/ArrayGraph.v) contains spatial predicates to represent graphs in arrays. The definition (5.29) is in this file. So there is no surprise to see it is used in sector 14, the proof script to verify the array version of the union-find algorithm.

Sector 4 (msl_application/GList.v) contains theorems about representing



1: graph/LstGraph.v	7: sample_mark/env_unionfind.v	13: sample_mark/verif_unionfind.v
2: graph/UnionFind.v	8: sample_mark/env_unionfind_arr.v	14: sample_mark/verif_unionfind_arr.v
3: msl_application/ArrayGraph.v	9: sample_mark/env_unionfind_iter.v	15: sample_mark/verif_unionfind_iter.v
4: msl_application/GList.v	10: sample_mark/spatial_array_graph.v	16: sample_mark/verif_unionfind_iter_rank.v
5: msl_application/GList_UnionFind.v	11: sample_mark/spatial_graph_glist.v	17: sample_mark/verif_unionfind_rank.v
6: msl_application/UnionFindGraph.v	12: sample_mark/spatial_graph_uf_iter.v	18: sample_mark/verif_unionfind_slim.v

Figure 5.12: Theorems in the Verification of Union-Find Program

graphs in a series of pointers. It is a specialization of Graph.v, just like GraphBi.v for binary graphs.

Sector 5 (msl_application/GList_UnionFind.v) contains the ramification theorems used in the verification of union-find.

Sector 6 (msl_application/UnionFindGraph.v) contains definitions and theorems independent of spatial representations. The vertex information function (5.27) is defined in this file. It also contains all pure theorems such as (5.36). This is the most widely used file in the verification of union-find. There are intensive curves to it in Figure 5.12.

Sectors 7–12 contains various implementations to interfaces which are necessary to connect VST, the spatial library and our verification.

Sectors 13-18 are proof scripts verifying 6 variants of the union-find algorithm. Figure 5.9 and Figure 5.10 are about verif_unionfind_slim.v. Figure 5.11 is about verif_unionfind_arr.v.

Proof scripts with suffix rank (sector 16, 17) add one more requirement in the post condition of the `find` function: the `rank` field of the data structure is not changed. At first we introduced `rank` into the verification because we attempted to prove the necessary amortization bounds. But soon we ran into an overflow issue: it was impossible to prove that the `rank` would not exceed max_int because the CompCert memory model does not place a bound on the total number of allocations. Informally, this overflow is impossible in practice because no computer has $2^{2^{64}}$ bytes of memory, which would be required to create enough nodes and union them together to create such an overflow, but Coq remains unconvinced.

Proof scripts whose names contain iter (sectors 15 and 16) verify the `find` function which uses loops instead of recursions. The ramification theorems are the same. By choosing the right loop invariant, only small modification of existing pure theorems is required.

Sector 13 (verif_unionfind.v) uses a stronger relation *findS* to replace the *uf_equiv* in the specification of the `find` function. This affects the pure theorems. Both the premise and conclusion are enhanced. Later we find that *uf_equiv* is strong enough and sufficient to describe the behavior.

The specification of the `unionS` functions are all the same.

5.5 Verification of a Garbage Collector

Anand et al. [2017] are developing CertiCoq, a compiler which translates programs written in Gallina—the specification language of Coq—to C code. Then CompCert will compile the C code to machine code. Gallina assumes infinite heap memory whereas C has a more realistic finite heap. CertiCoq supports Gallina's assumption via memory management at the C level. In particular, the C code generated by CertiCoq contains calls to a garbage collector (GC), also written in C. CertiCoq aims to be end-to-end certified, so the GC must also be certified.

Unlike our previous examples, the GC is a realistic program having a concrete working scenario, with sophisticated implementations. Its scale and complexity present a huge verification challenge. For example, the objects manipulated by the GC may have different number of fields and the fields may be boxed or unboxed and must be disambiguated at runtime. After a detailed and thoughtful consideration, we figured out how to model the verification via our framework. After a great engineering effort which lasted eight months, we finally finished the verification by proving the graph isomorphism between the states before and after the garbage collection.

The statistics in Table 6.1 show that the verification of GC is a huge project, which contains more than 230 definitions and more than 700 theorems. It is almost as large as the sum of the math and spatial graph libraries. Since the verification contains so many details, we cannot and do not explain every aspect of it. Instead, in the following sections, we will briefly introduce the program, explain how to abstract the data structure in GC as a graph in general, describe the principle in designing the specifications of functions, propose steps to accomplish the graph isomorphism, provide both the local and global views of the verification, and report some bugs we found in the original C code during the verification.

5.5.1 The algorithm of GC

The program we aim to verify (i.e. GC) is a generational copying garbage collector. The heap is divided into several disjoint spaces called *generations*. GC always starts from the first generation. It examines the objects in a generation to see if they are accessible from a root set. If they are, they would be copied to the next generation. When all such objects are copied, the original generation will be cleaned. The collection from one generation to another employs Cheney's algorithm [Cheney, 1970]. This collection may trigger the collection of the second generation into the third, etc.

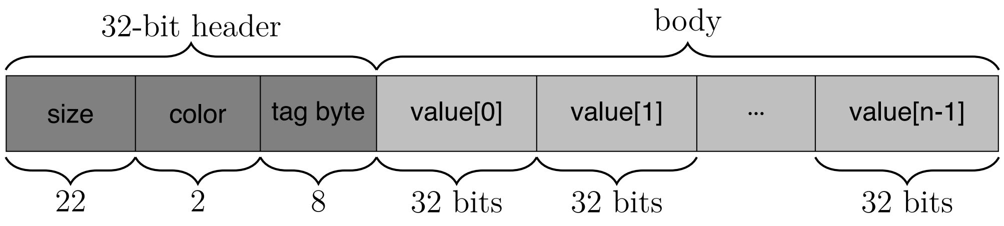


Figure 5.13: A Single Block, the Basic Unit in GC

Figure 5.13 illustrates the basic unit manipulated by GC. It is called a *block*, which has two parts. One is a 32-bit header and the other is an array with variable size. The header records the size n of the body. The data stored in each slot of the array is either an unboxed integer data value or a pointer. GC follows the convention from OCaml's garbage collector to disambiguate the two: all integers must be odd and all pointers must be even-aligned [Hickey et al., 2014]. Pointers may point to other blocks or data structures outside the GC's purview. If they point to other blocks, they always point to the `value[0]` slot of those blocks.

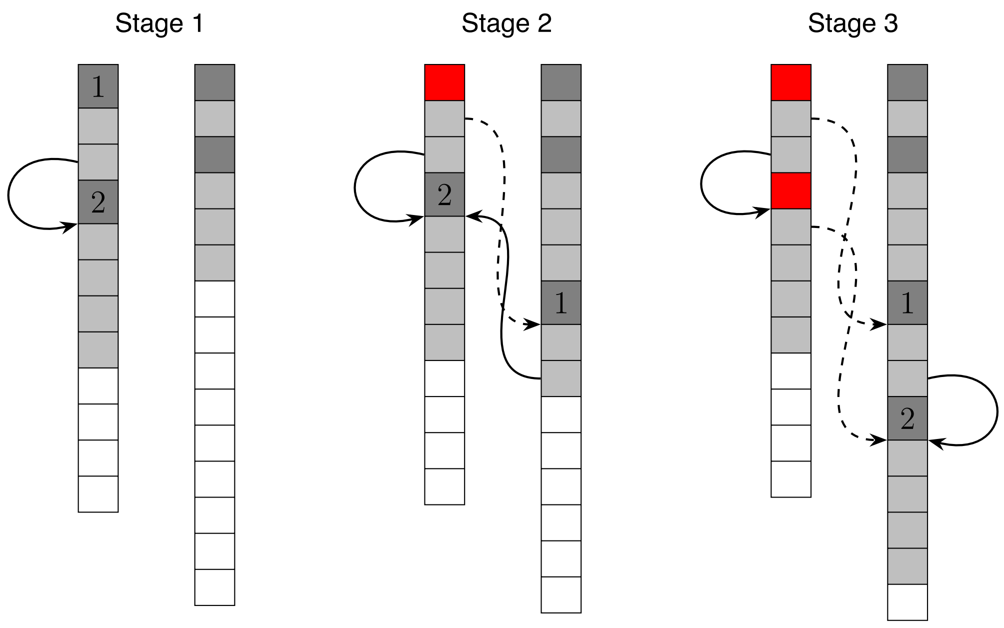


Figure 5.14: The `forward` Procedure in GC

The "examine and copy" procedure is carried by the `forward` function, which accepts a pointer p to a block, a "from" generation f, and a "to" generation t as parameters. This function plays a dual role in such a procedure. Both are illustrated in Figure 5.14. The stage 1 is the initial state which has two generations. In the left generation there are two blocks, block 1 and block 2. The dark gray boxes represent the headers of blocks and the light gray boxes represent the bodies. We can see the second field of block 1 points to block 2. Now suppose the pointer p is in the root set and it points to block 1. The `forward` functions checks that block 1 is in generation f. So block 1 is copied to the next free position in generation t, the right one. Meanwhile, `forward` will mark the original block 1 and leave the address of the copied block in the first field of the original one. The state becomes stage 2. The header of the original block 1 is red and there is a dashed arrow pointing to the copied block 1. The p in the root set will be modified. Note that modifying p in the root set **does not change the**

graph. This is one role of the `forward` function. It “forwards” the block 1.

Since the copied block 1 is exactly the same as the original, its second field also points to block 2 in the left generation. To forward the field of a block, we need to call `forward` again. This time the parameter p is the second field of the copied block 1. The `forward` function will trace p and do the same thing: copy, mark block 2 and modify its first field. After that, it will change p to point to the copied block 2, as shown in stage 3. This time modifying p **changes the graph**.

In both case, if `forward` finds that a block is marked, it will not copy it again but just change the pointer p to the new copied block. Recall that the copied block’s address has been placed in the first field of the marked block.

The `forward_roots` function accepts a root set l, a “from” generation f and a “to” generation t as parameters. For every pointer in l which points to a block in generation g, the block will be forwarded and the pointer will point to the new position.

The `do_scan` function accepts a “from” generation f, a “to” generation t, and a position s in generation t. It will scan and forward every field across blocks from s. If the field is not a pointer or does not point to generation f, `forward` would do nothing. We should note that the scan and forward operation may create new fields. So `do_scan` will loop until all fields are scanned and no new fields are created.

The `do_generation` function combines `forward_roots` and `do_scan` together. It stores the first free position s in generation t. Then it calls `forward_roots` first to forward blocks pointed by the root set. Next it calls `do_scan` from the position s. The `do_generation` function clears the generation f in the end.

Finally the `garbage_collect` puts `do_generation` in a while loop to trigger the “cascade” collection when the free space is less than half of the capacity of the "to" generation.

This is the algorithm of GC. Through a series of function calls, most tasks are carried out by the `forward` function.

5.5.2 The definitions of γ and `graph`

Figure 5.14 gives an impression about the memory layout of the blocks. They are classified according to their generations and put sequentially in each generation. If we treat each generation as a whole, then generations are distributed discontinuously. Our challenge is figuring out what the types of `LabeledGraph`'s `Vertex`, `Edge`, `DV` etc. should be, so that we can define proper spatial predicates to describe the data structure used in GC.

To abstract the data structure used in GC as a graph, blocks must be abstracted into vertices and pointers to other blocks must be abstracted into edges to other vertices. We decide to abstract the v_ith block of the v_gth generation into a vertex (v_g, v_i), i.e. a pair of natural numbers. The information stored in a block except pointers to other blocks is packed into a big record type `raw_vertex_block` as the label of a vertex. If the jth field of a vertex/block (v_g, v_i) is a pointer to other blocks, then there is an edge (v_g, v_i, j) from (v_g, v_i). The destination is stored in the `dst` function of `PreGraph`. To construct a block from an abstract vertex (v_g, v_i), the only missing information is the starting address of the v_gth generation. The information of each generation including the starting address and number of vertices is packed into another record type `graph_info` as the global label of the graph. Table 5.2 gives all determined types needed for a `LabeledGraph` γ used in GC.

Vertex	Edge	DV	DE	DG
nat ∗ nat	nat ∗ nat ∗ nat	raw_vertex_block	unit	graph_info

Table 5.2: Instantiated Types of a Graph in GC

Unlike the verification of `mark`, `spanning`, `find` and `unionS`, the γ here is not a `GeneralGraph` because we do not provide the soundness condition so far. The soundness condition is absolutely necessary in our fully functional correctness verification later. But here we decide to use `LabeledGraph` instead of `GeneralGraph` as the type of the mathematical graph γ in spatial predicates and specifications of GC.

This big decision is made because of the complexity of GC. All previous verification is about algorithms performed by one single function. It is reasonable to emphasize that the soundness is preserved after the execution of the function. However, the GC algorithm described in §5.5.1 is a composition of several functions. The preserved soundness condition for the top level function may not be preserved for low level function. For example, before and after the whole algorithm (`garbage_collect`), no vertex/block is marked "copied". But as illustrated in Figure 5.14, it is very common to have marked vertices before and after `forward`. If we put "no marked vertices" in the soundness condition, the `forward` function breaks it. If we do not put it in the soundness condition, then the verification of `garbage_collect` cannot guarantee there is no copied vertex. This dilemma forces us to remove the soundness condition. After all, the soundness condition is just a pure property. For GC, it does not affect the definition of spatial predicates. We can put "no marked vertices" to the specification of `garbage_collect` only. In other words, different functions in GC could have different requirements of γ in their specifications. There is no need to bind a non-universal soundness condition to γ in this case.

Now we can define the spatial predicate `graph` to describe the whole data structure:

$$\mathsf{graph}(\gamma) \overset{\text{def}}{=} \mathop{\Huge \ast}_{i\in[0,1,\ldots,numgen(\gamma)]} \left(\mathop{\Huge \ast}_{j\in[0,1,\ldots,numv(\gamma,i)]} \mathsf{vrep}(\gamma,(i,j)) \right) \tag{5.37}$$

$$\begin{gathered}\mathsf{vrep}(\gamma,v) \overset{\text{def}}{=} (vaddr(\gamma,v)-4) \mapsto header(\gamma,v) * \\ vaddr(\gamma,v) \mapsto fields(\gamma,v)\end{gathered} \tag{5.38}$$

The whole graph representation $\mathsf{graph}(\gamma)$ is a two-fold "big star". It is an iterated separation conjunction of generation representations. Each generation representation is an iterated separation conjunction of vertex representations. A vertex representation $\mathsf{vrep}(\gamma,v)$ is a separation conjunction of two components. One is a 4-byte header and the other is an array of vertex-dependent size. The functions $header(\gamma,v)$ and $fields(\gamma,v)$ compose the header and fields representations of v respectively. Because of the complexity we omit their concrete definitions. The function $vaddr(\gamma,v)$ calculates the address of first field of v in γ. Its definition is given in (5.39).

$$vaddr\Big(\gamma,(i,j)\Big) \overset{\text{def}}{=} start(\gamma,i) + 4 \times \sum_{k=1}^{j-1} vsize\Big(\gamma,(i,k)\Big) + 4 \tag{5.39}$$

The function $start(\gamma,i)$ retrieves the starting address of the ith generation in γ from the global label. The function $vsize(\gamma,v)$ gives the size of a vertex v, i.e. number of fields + 1. The extra 1 is for the header. So the function $vaddr$ sum up the sizes of all vertices before (i,j) in generation i. The total size times 4 is the offset from the starting address. With the offset of all vertices before and its own header (the extra 4 in (5.39)), we finally get the address of vertex (i,j) in the heap.

5.5.3 The specifications of GC

Defining specifications for those functions used in GC is another big challenge. In previous verification, we just needed to formalize our intuition about what the function does as a relation about states before and after the execution. Then we put the spatial predicates and the relation together as the specification. Sometimes this specification just works, like the graph mark program and union-find. Sometimes, we need a careful consideration (even through trial-and-error method) to modify the original specification to go through the verification, like the spanning tree program. We should note that these specifications only reflect our understanding of the programs. A successful verification means the behavior of the program fulfills the specification. It does not mean the specification is a complete description of a program's behavior.

We tried to use the same method in defining the specifications in GC. For the top level function `garbage_collect`, the intuition tell us the isomorphism between two `LabeledGraph`s is a proper relation as the specification. But for the workhorse `forward`, the intuition cannot help us too much. From its code in Figure 5.17, it can check, copy, mark, and redirect the parameter pointer. It can just check and redirect. It also can just check and do nothing at all. What makes it even worse is that its two-fold roles discussed in §5.5.1 when it does all these things. Its role depends on whether the parameter pointer is in the root set or is just a pointer in the field. Different roles correspond to different actions on the graph even for the same piece of code. Such complicated behavior cannot be abstracted into a simple, elegant relation.

Our solution is to define an inductive relation `forward_relation` which captures the exact behavior of the `forward` function.

Figure 5.15 gives a peek of the definition of `forward_relation`. Every role, every execution path of `forward` has a corresponding constructor in `forward_relation`. The type `forward_t` is an optional type. It could be an element in the root set or an edge of the graph. We use this type to disambiguate the role of `forward`. Let us inspect some of the constructors.

```
Inductive forward_relation (from to: nat):
  nat -> forward_t -> LGraph -> LGraph -> Prop :=
| fr_z: forall depth z g,
    forward_relation from to depth (inl (inl (inl z))) g g
| fr_v_not_in: forall depth v g, vgeneration v <> from ->
    forward_relation from to depth (inl (inr v)) g g
| fr_v_in_forwarded: forall depth v g,
    vgeneration v = from -> (vlabel g v).(raw_mark) = true ->
    forward_relation from to depth (inl (inr v)) g g
| fr_v_in_not_forwarded_O: forall v g,
    vgeneration v = from -> (vlabel g v).(raw_mark) = false ->
    forward_relation from to O (inl (inr v)) g
                     (lgraph_copy_v g v to)
| ...
```

Figure 5.15: Part of the Inductive Relation `forward_relation`

The constructor `fr_z` means for an integer element z in the root set, we have `forward_relation ... g g`, which means `forward` does nothing. The execution path for constructor `fr_v_in_forwarded` is when the vertex element is in the "from" generation but it is marked, the forward function will redirect element in the root set to the copied one. But this redirection does not change the graph. We again have `forward_relation ... g g`. When the vertex is not marked, it will be copied, marked and modified. We define a function `lgraph_copy_v` to represents the changes to the graph. So we have `forward_relation ... g (lgraph_copy_v g v to)` in the next constructor.

By converting every step into a premise and translating every modification about the data structure to a function which returns the modified underlying model, we get an accurate description of one execution path. By combining them together, we can define a verbose but exact relation.

More importantly, if the specification s of low level function f is defined in this way, s can be reused in the specification of a function g which calls f. For example, we can define the relation about graphs for `forward_roots` in Figure 5.16. The relation `forward_relation` is used in `fr_loop`. In the

```
Inductive fr_loop (from to: nat) (fi: fun_info):
  list nat -> roots_t -> LGraph -> roots_t -> LGraph -> Prop :=
| frl_nil: forall g roots, fr_loop from to fi nil roots g roots g
| frl_cons: forall g1 g2 g3 i il r1 r3,
    forward_relation from to O
    (root2forward (Znth (Z.of_nat i) r1)) g1 g2 ->
    fr_loop from to fi il
    (upd_roots from to (inl (Z.of_nat i)) g1 r1 fi) g2 r3 g3 ->
    fr_loop from to fi (i :: il) r1 g1 r3 g3.
Definition fr_relation from to fi r1 g1 r2 g2 :=
  fr_loop from to fi (nat_inc_list (length r1)) r1 g1 r2 g2.
```

Figure 5.16: Relation for `forward_roots`

constructor `frl_cons`, `fr_loop` itself appears in the premise. This makes `fr_loop` a real inductive definition. The relation involving recursions and loops can be easily defined using such inductive relations.

We adopt this method in defining specifications for all functions in GC, even for the highest-level `garbage_collect` function. The postconditions (including the inductive relations) in such accurate specifications can be seen as strongest postconditions. So we just need to derive the properties indicating functional correctness we want from the postconditions. Then the CONSEQUENCE rule in Figure 2.1 will establish the successful verification of the functional correctness of those programs.

One big advantage of this methodology is that we only need to perform the spatial inference once. The specifications are sort of mechanical. We do not need guess one and modify it through trial and error, as we did in verifying spanning tree. The pure functions and relations used in the specifications only reflect the spatial modification. From this perspective, the methodology separates the spatial inference and the high-level, human-understandable correctness. As far as we know, this method is quite new. The idea comes from the discussion with Qinxiang Cao. He also used this techinque in the verification of the KMP algorithm[1].

[1]It is an unpublished work.

5.5.4 The proofs of GC

As pointed out in §5.5.3, the inductive relations used in the specifications split the proofs of GC into 2 categories. One is the spatial proofs involving VST saying the C programs meet the verbose specifications. The other is the pure mathematical proof saying we can derive the functional correctness from the verbose specification.

The Figure 5.17 on page 170 gives a simplified proof sketch of the `forward` function for one of its roles, in this case the pointer `p` is the nth field of a vertex. The proof sketch is listed here to show how complicated the verification of a real program could be. The description of states is so complicated and verbose that we have to define some shortcuts such as ϕ_1, ϕ_{14} and ϕ_{26}, to make the specification shorter. Our proof script discusses the two roles of `forward` in the very beginning. So it verifies `forward` twice for its different roles. The proof script of `forward` has more than 1300 lines.

We use the relation `gc_graph_iso` defined in Figure 5.18 to represent the graph isomorphism between graphs before and after the garbage collection.

The `roots1` and `roots2` are root sets. The relation `gc_graph_iso` focuses on the reachable subgraphs from the root sets. It says the reachable subgraphs are isomorphic. Figure 3.5 on page 50 gives the definition of `label_preserving_graph_isomorphism_explicit`.

The proof saying that the graphs before and after garbage collection satisfy `gc_graph_iso` can be split into several stages. We define two more relations `gc_graph_semi_iso` and `gc_graph_quasi_iso` to help. We first prove that `forward` preserves `gc_graph_semi_iso`, so as `forward_roots` and `do_scan`. Then we prove the graphs before and after running `forward_roots` and `do_scan` continuously satisfy `gc_graph_quasi_iso`. At last we prove `do_generation`, which is a composition of `forward_roots`, `do_scan` and clear operation of generations, satisfies `gc_graph_iso`. As a while loop of the

```
// $\left\{\begin{array}{l}\forall\gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}, v, n \,.\, \mathsf{graph}(\gamma) * \mathsf{finf}(\mathit{finf}) * \mathsf{tinf}(\mathit{tinf}) \wedge \\ \mathit{compat}(\gamma, \mathit{finf}, \mathit{tinf}, \mathit{from}, \mathit{to}) \wedge \\ \mathtt{s} = \mathit{start}(\gamma, \mathit{from}) \wedge \mathtt{l} = \mathtt{s} + \mathit{gensz}(\gamma, \mathit{from}) \wedge \\ \mathtt{n} = \mathit{naddr}(\mathit{tinf}, \mathit{to}) \wedge \mathtt{p} = \mathit{vaddr}(\gamma, v) + n\end{array}\right\} \stackrel{\text{def}}{=} \phi_1$
void forward (value *s, *l, **n, *p) {
 value *v; value va = *p;
 if(Is_block(va)) {// is ptr
  v = (value *)((void *)va);
  if(Is_from(s, l, v)) {// in from
// $\{\phi_1 \wedge \exists e, v' \,.\, \mathit{lab}(\gamma, v)[n] = e \wedge \mathit{dst}(\gamma, e) = v' \wedge \mathtt{v} = \mathit{vaddr}(\gamma, v')\} \stackrel{\text{def}}{=} \phi_7$
// $\searrow \{\exists \mathit{flds}', \mathit{hdr}' \,.\, \mathit{flds}' = \mathit{lab}(\gamma, v') \wedge v' \mapsto \mathit{flds}' \wedge \mathit{hdr}' = \mathit{flds}'[-1]\} \stackrel{\text{def}}{=} \phi_8$
   header_t hd = Hd_val(v);
// $\swarrow \{\phi_8 \wedge \mathtt{hd} = \mathit{val}(\mathit{hdr}')\}$
// $\{\phi_7 \wedge \mathtt{hd} = \mathit{val}(\mathit{hdr}')\}$
   if(hd == 0) {// already forwarded
// $\{\phi_7 \wedge \mathtt{hd} = 0\}$
// $\searrow \left\{\begin{array}{l}\exists f, f' \,.\, v \mapsto f * v' \mapsto f' \wedge f = \mathit{lab}(\gamma, v) \wedge \\ f' = \mathit{lab}(\gamma, v') \wedge f'[0] = \mathit{vaddr}(\gamma, \mathit{copy}(\gamma, v')) \wedge \mathtt{p} = \&f[n]\end{array}\right\} \stackrel{\text{def}}{=} \phi_{14}$
    *p = Field(v,0);
// $\swarrow \{\phi_{14} \wedge f[n] := f'[0]\}$
// $\left\{\begin{array}{l}\phi_{30} \wedge \exists\gamma' \,.\, \mathsf{graph}(\gamma') \wedge \gamma' = \mathit{upd\_edge}(\gamma, e, \mathit{copy}(\gamma, v')) \wedge \\ \mathit{fwd\_postcondition}(\gamma, \gamma', \mathit{tinf}, \mathit{finf}, \mathit{from}, \mathit{to}, v, n)\end{array}\right\}$
   } else {// not yet forwarded
// $\{\phi_7 \wedge \mathtt{hd} \neq 0\} \stackrel{\text{def}}{=} \phi_{20}$
    int i; int sz; value *new;
    sz = size(hd); new = *n+1; *n = new+sz;
// $\left\{\begin{array}{l}\phi_{20} \wedge \mathtt{sz} = \mathit{blocksize}(\mathtt{hd}) \wedge \\ \mathtt{new} = \mathit{start}(\gamma, \mathit{to}) + \mathit{used}(\gamma, \mathit{to}) + 1 \wedge \mathtt{n} = \mathtt{new} + \mathtt{sz}\end{array}\right\} \stackrel{\text{def}}{=} \phi_{22}$
    Hd_val(new) = hd;
    for(i = 0; i < sz; i++)
     Field(new, i) = Field(v, i);
// $\left\{\begin{array}{l}\phi_{22} \wedge \exists\gamma', v'', \mathit{tinf}' \,.\, \mathsf{graph}(\gamma') * \mathsf{tinf}(\mathit{tinf}') \wedge \\ v'' = \mathit{new\_cp\_v}(\gamma, \mathit{to}) \wedge \gamma' = \mathit{copy\_vertex}(\gamma, \mathit{to}, v', v'') \wedge \\ \mathit{compat}(\gamma', \mathit{finf}, \mathit{tinf}', \mathit{from}, \mathit{to})\end{array}\right\} \stackrel{\text{def}}{=} \phi_{26}$
    Hd_val(v) = 0;
    Field(v, 0) = (value)((void *)new);
// $\{\phi_{26} \wedge \mathit{val}(\mathit{hdr}') = 0 \wedge \mathit{flds}'[0] = \mathit{copy}(\gamma, v')\} \stackrel{\text{def}}{=} \phi_{29}$
// $\searrow \{\exists \mathit{flds} \,.\, v \mapsto \mathit{flds} \wedge \mathit{flds} = \mathit{lab}(\gamma', v)\} \stackrel{\text{def}}{=} \phi_{30}$
    *p = (value)((void *)new);
// $\swarrow \{\phi_{30} \wedge \mathit{flds}[0] := \mathit{vaddr}(\gamma, v'')\}$
// $\left\{\begin{array}{l}\phi_{29} \wedge \exists\gamma'' \,.\, \mathsf{graph}(\gamma'') \wedge \gamma'' = \mathit{upd\_edge}(\gamma', e, v'') \wedge \\ \mathit{compat}(\gamma'', \mathit{finf}, \mathit{tinf}', \mathit{from}, \mathit{to})\end{array}\right\}$
}}}} // $\{\mathit{fwd\_postcondition}(\gamma', \gamma'', \mathit{tinf}', \mathit{finf}, \mathit{from}, \mathit{to}, v, n)\}$
```

Figure 5.17: Clight Code and Proof Sketch for `forward`

```
Definition gc_graph_iso (g1: LGraph) (roots1: roots_t)
           (g2: LGraph) (roots2: roots_t): Prop :=
  let vertices1 := filter_sum_right roots1 in
  let vertices2 := filter_sum_right roots2 in
  let sub_g1 := reachable_sub_labeledgraph g1 vertices1 in
  let sub_g2 := reachable_sub_labeledgraph g2 vertices2 in
  exists vmap12 vmap21 emap12 emap21,
    roots2 = map (root_map vmap12) roots1 /\
    label_preserving_graph_isomorphism_explicit
      sub_g1 sub_g2 vmap12 vmap21 emap12 emap21.
```

Figure 5.18: The Definition of Graph Isomorphism Used in GC

function `do_generation`, `garbage_collect` also satisfies `gc_graph_iso`.

5.5.5 Theorems in the verification of GC

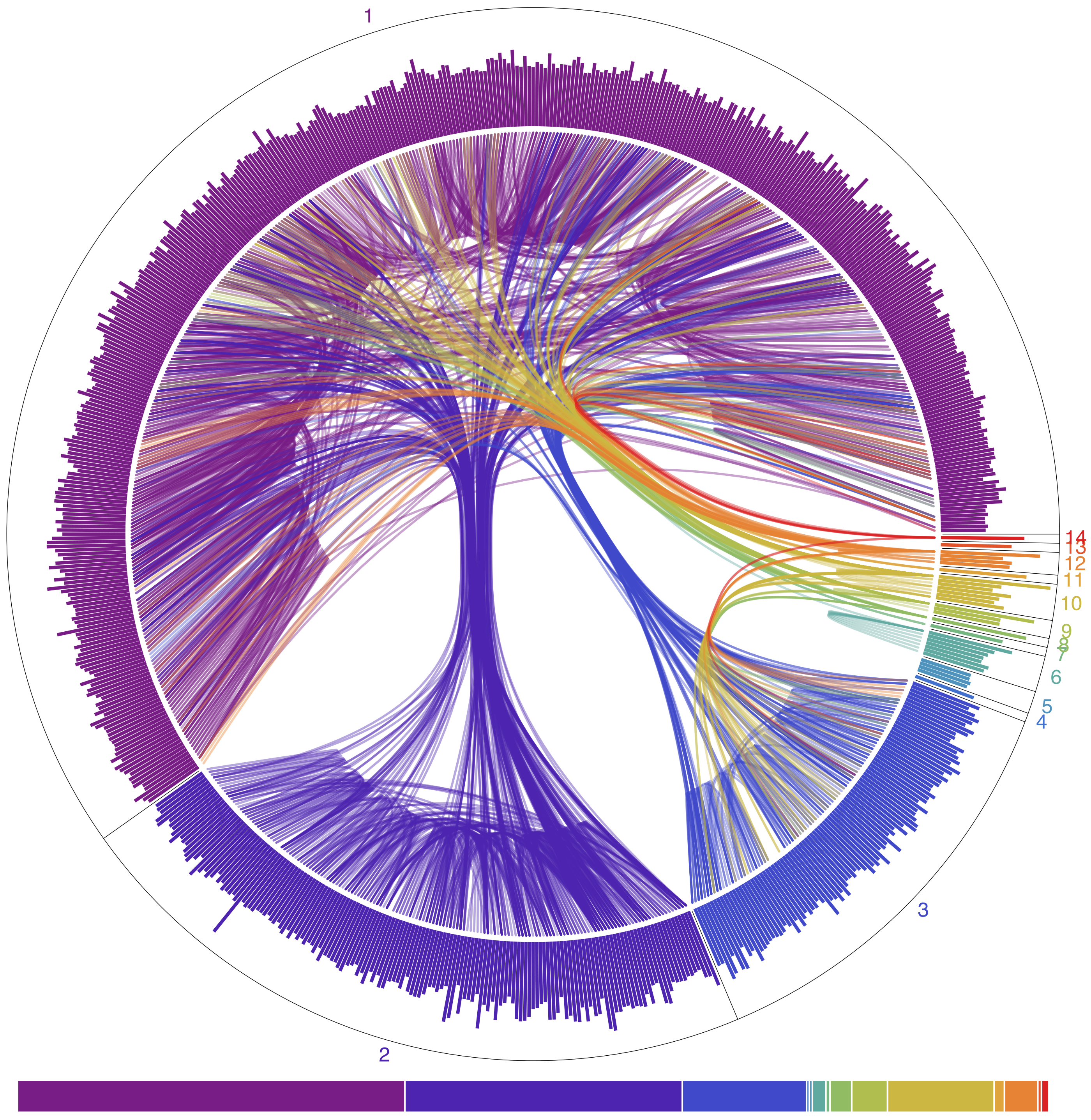
Figure 5.19 on page 172 gives an overview of the theorems in the verification of GC. Their relations are rather simple.

Sector 1 (CertiGC/GCGraph.v) contains all pure theorems used in the spatial verification of GC, i.e. verifying the functions satisfy the inductive relations. There are 432 of them.

Sector 2 (CertiGC/gc_correct.v) contains all pure theorems in proving the final graph isomorphism. There are 155 of them. We can see they only depend on GCGraph.v. It is totally expectable that none of other files depends on gc_correct.v because it just contains theorems to establish the graph isomorphism. It is our ultimate goal in the verification of GC. No other targets should depend on it.

Sector 3 (CertiGC/spatial_gcgraph.v) contains all spatial theorems in verifying GC, including ramification theorems. There are 91 in total.

Sectors 4–14 are proof scripts of all functions in GC. From Figure 5.19, it is very clear that most of them depends on sector 1 and 3, pure and spatial supporting theorems.



■ 1: CertiGC/GCGraph.v
■ 2: CertiGC/gc_correct.v
■ 3: CertiGC/spatial_gcgraph.v
■ 4: CertiGC/verif_ls_block.v
■ 5: CertiGC/verif_conversion.v
■ 6: CertiGC/verif_create_heap.v
■ 7: CertiGC/verif_create_space.v
■ 8: CertiGC/verif_do_generation.v
■ 9: CertiGC/verif_do_scan.v
■ 10: CertiGC/verif_forward.v
■ 11: CertiGC/verif_forward_roots.v
■ 12: CertiGC/verif_garbage_collect.v
■ 13: CertiGC/verif_make_tinfo.v
■ 14: CertiGC/verif_resume.v

Figure 5.19: Theorems in the Verification of the GC Program

We could note that one segment in sector 10 is extremely long. The length of a segment represents the length of the proof in log scale. This extremely long theorem is the verification of `forward`. We prove it in two rounds for its two-fold roles.

5.5.6 Bugs we found during the proof

We discovered and fixed two bugs in the source code during our verification. The first was a performance bug we discovered when developing the key invariants. The original GC code executed Cheney's algorithm too conservatively, scanning the entire `to` generation for backward pointers into `from`. We showed that scanning a subset of `to` suffices. Performance doubled.

The second bug was an overflow when subtracting two pointers to calculate the size of a space, as below. Pointers `start` and `limit` point to the beginning and end of the `i`th space of the heap `h`.

```
int w = h->spaces[i].limit - h->spaces[i].start;
```

This subtraction is defined in C and Clight, but will overflow if the difference equals or exceeds 2^{31}. We adjusted the size of the largest generation to avoid this overflow.

5.5.7 Comparison with other verification of GC

As we mentioned in §1.5, Varming and Birkedal [2008] gave a formal proof of Cheney's copying collector written in a simple, self-defined language with procedures. They defined the invariants of Cheney's algorithm in their Isabelle/HOLCF implementation of higher-order separation logic. Instead of proving graph isomorphism, they proved a similar property called heap isomorphism. Compared to our 14000 lines of verification, their verification of GC has 7500 lines. We believe that the additional complexity of our verification comes from the more complicated implementation and sematics of the C programming language.

Schism/cmr [Pizlo et al., 2010] is a certified concurrent collector built in a Java VM that services multi-core architectures with weak memory consistency. Its mark-sweep collector kernel against a relaxed memory model is formally verified by [Gammie et al., 2015] in the Isabelle/HOL proof assistant. The code they verified is specified in a simple and intuitive programming language Cimp. So it is based on a detailed abstract model.

McCreight et al. [2007, 2010] introduced GCminor, which is a certified translation step added to CompCert's compilation from Clight to assembly. GCminor makes explicit the specific invariants that the garbage collector relies upon, thus minimising errors due to the violation of invariants between the garbage collector and the mutator. Their framework eases the burden in the verification of GC written in GCminor.

Hawblitzel and Petrank [2009] annotated x86 code for two GCs for the Bartok C# compiler, and then used Boogie verification generator and the Z3 automated theorem prover to verify their correctness automatically.

The closest piece of work to our certified GC is probably the excellent certified GC for the CakeML project [Sandberg Ericsson et al., 2019], since both integrate a certified GC into a certified compiler for a functional language. Starting from an algorithm-level modelling and verification of a GC in HOL4 theorem prover, they implemented the GC in several intermediate languages which the CakeML compiler goes through. Then they proved that the GC correctly mimics the operations performed by the algorithm-level implementation. After that, the GC is treated as just another part of the program to be compiled by CakeML. The largest difference, however, is that we present an integrated graph framework suitable for reasoning about many graph algorithms, of which our GC is merely the flagship. In contrast, they focus much more narrowly on the problem of certified GCs.

Chapter 6

Conclusion and Future Work

In this thesis we explore problems during the mechanized verification of graph-manipulating programs. We develop a framework which provides various constructions to formalize graph-related concepts. The framework also contains a whole set of theorems which facilitate reasoning about graph-manipulating programs. Since we focus on the end-to-end verification of real C code, we integrate our framework into the VST toolset. Then we apply our techniques in the verification of several classical graph algorithms written in C. All these works are written and machine-checked in Coq. Table 6.1 shows some statistics about our code base.

Component	Files	Size (in lines)	Definitions	Theorems
Common Utilities	10	3,578	44	289
Math Graph Library	20	10,585	216	581
Spatial Graph Library	3	2,328	59	110
Integration into VST	11	2,783	17	172
Marking	6	775	9	20
Spanning Tree	5	2,723	17	92
Union-Find	18	3,193	107	135
Garbage Collector	16	13,858	235	712
Total Development	89	39,823	704	2,111

Table 6.1: Statistics for Our Code Base

We have the following main contributions.

- We propose the Localize rule which generalizes the Ramify rule to handle modified program variables and existential quantifiers in postconditions more smoothly.

- We develop a general and modular framework for defining and reasoning about mathematical graphs. It is expressive and powerful enough to adapt various requirements of graphs in different applications.

- We develop a general and modular spatial library for reasoning about graphs in the heap. It contains a large portion of ramification entailments which ease the burden of spatial deduction.

- By connecting mathematical graphs to spatial graphs in the heap via separation logic, we demonstrate the mechanical verification of several nontrivial graph-manipulating algorithms written in C, including graph mark, spanning tree, union-find and a garbage collector.

In the future we plan to improve the pure reasoning of graphs and similar data structures. Currently the reasoning of pure mathematical fact is rather painful. We hope that automation can improve the current situation. The graphs used in all our finished verification share the same feature: edges in those graphs carry no information other than connectivity. We plan to verify the Dijkstra's algorithm which has length information in edges, so as to make our demonstrations more complete and promising. The third potential direction is integrating our framework to some other toolkit that uses a different heap model and pursues automation, such as HIP/SLEEK [Chin et al., 2010].

Bibliography

Amal Ahmed. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 69–83, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33095-X, 978-3-540-33095-0. doi: 10.1007/11693024_6. URL http://dx.doi.org/10.1007/11693024_6.

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A Verified Compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, Apr 2015. ISSN 0164-0925. doi: 10.1145/2701415. URL http://doi.acm.org/10.1145/2701415.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014. ISBN 110704801X, 9781107048010.

Andrew W. Appel, Lennart Beringer, Qinxiang Cao, and Josiah Dodds.

Verifiable C: Applying the Verified Software Toolchain to C programs. Apr 2019. URL http://vst.cs.princeton.edu/download/VC.pdf.

Kenneth Appel, Wolfgang Haken, and John Koch. Every Planar Map is Four Colorable. *Illinois Journal of Mathematics*, 21:439–657, 1977.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. URL http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf. Snowbird, Utah.

Gertrud Bauer and Tobias Nipkow. The 5 Colour Theorem in Isabelle/Isar. In *International Conference on Theorem Proving in Higher Order Logics*, pages 67–82. Springer, 2002.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-68977-X, 978-3-540-68977-5.

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 315–331, 2012.

Wolfgang Bibel. Early History and Perspectives of Automated Deduction. In Joachim Hertzberg, Michael Beetz, and Roman Englert, editors, *KI 2007: Advances in Artificial Intelligence*, pages 2–18, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74565-5.

Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local Reasoning About a Copying Garbage Collector. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 220–231, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964020. URL `http://doi.acm.org/10.1145/964001.964020`.

J.A. Bondy and U.S.R. Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1846289696.

Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local Reasoning, Separation and Aliasing. *Space*, 4, 2004.

Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as Resource in Separation Logic. *ENTCS*, 155:247–276, 2006.

R.S. Boyer, M. Kaufmann, and J.S. Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers & Mathematics with Applications*, 29(2):27–62, 1995. ISSN 0898-1221.

Ricky W Butler and Jon A Sjogren. A PVS Graph Theory Library. Technical report, 1998.

Vinton G. Cerf. A Comprehensive Self-Driving Car Test. *Communications of the ACM*, 61(2):7–7, January 2018. ISSN 0001-0782. doi: 10.1145/3177753. URL `http://doi.acm.org/10.1145/3177753`.

Arthur Charguéraud and François Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 137–153, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22102-1.

Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, 62(3):331–365, Mar 2019. ISSN 1573-0670. doi: 10.1007/s10817-017-9431-7. URL https://doi.org/10.1007/s10817-017-9431-7.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815402. URL http://doi.acm.org/10.1145/2815400.2815402.

C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, 1970. doi: 10.1145/362790.362798.

Wei Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated Verification of Shape, Size and Bag Properties via User-Defined Predicates in Separation Logic. *Science of Computer Programming*, 77(9):1,006–1,036, 2010.

Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press New York, 2013. ISBN 978-0262026659.

Ching-Tsun Chou. A Formal Theory of Undirected Graphs in Higher-Order Logic. In *Higher Order Logic Theorem Proving and Its Applications*, pages 144–157. Springer, 1994.

Ching-Tsun Chou. Mechanical Verification of Distributed Algorithms in Higher-Order Logic. *The Computer Journal*, 38(2):152–161, 1995.

Stephen Clark. Attitude Control Failures Led to Break-up of Japanese

Astronomy Satellite. *Spaceflight Now*, Apr 2016. URL https://bit.ly/2FQWhkS.

Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. ISBN 978-3-319-10574-1. doi: 10.1007/978-3-319-10575-8. URL https://doi.org/10.1007/978-3-319-10575-8.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

The Coq Development Team. *The Coq Reference Manual, Release 8.9.0*, January 2019. Available electronically at http://coq.inria.fr/doc.

Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1988.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, chapter 21, pages 561–581. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Caroline Davies. Google Blacklists Entire Internet. *The Guardian*, 2009. URL https://bit.ly/2T99hGr.

Jared Davis and Magnus O. Myreen. The Reflective Milawa Theorem Prover is Sound (Down to the Machine Code That Runs It). *Journal of Automated Reasoning*, 55(2):117–183, Aug 2015. ISSN 0168-7433. doi: 10.1007/s10817-015-9324-6. URL http://dx.doi.org/10.1007/s10817-015-9324-6.

Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() is Broken: The Good, the Bad and the Worst Case. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 273–289, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21690-4.

Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover. In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972. ISSN 0001-0782. doi: 10.1145/355604.361591. URL `http://doi.acm.org/10.1145/355604.361591`.

Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Aug 1975. ISSN 0001-0782.

Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226, 2008. doi: 10.1145/1449764.1449782. URL `http://doi.acm.org/10.1145/1449764.1449782`.

Robert Dockins and Aquinas Hobor. Verifying time bounds for general function pointers. In *Twenty-Eigth Conference on Mathematical Founda-*

tions of Programming Semantics (MFPS 2012), pages 132–148. Springer ENTCS, June 2012.

Stephen Dolan. Fun with Semirings: A Functional Pearl on the Abuse of Linear Algebra. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 101–110, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323260. doi: 10.1145/2500365.2500613. URL `https://doi.org/10.1145/2500365.2500613`.

Catherine Dubois, Sourour Elloumi, Benoit Robillard, and Clément Vincent. Graphes et couplages en Coq. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, 2015.

Michael Dunn. Toyota's Killer Firmware: Bad Design and its Consequences. *EDN Network*, 2013. URL `https://ubm.io/2Mu3fh3`.

Jean Duprat. A Coq Toolkit for Graph Theory. *Rapport de recherche*, 15, 2001.

Steve M. Easterbrook. Climate Change: A Grand Software Challenge. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 99–104, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.1145/1882362.1882383. URL `http://doi.acm.org/10.1145/1882362.1882383`.

Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

Peter Gammie, Antony L. Hosking, and Kai Engelhardt. Relaxing safely: Verified on-the-fly garbage collection for x86-tso. In *Proceedings of the*

36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, page 99–109, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2738006. URL https://doi.org/10.1145/2737924.2738006.

Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007. doi: 10.1007/978-3-540-73368-3_52. URL https://doi.org/10.1007/978-3-540-73368-3_52.

David K Gifford and John M Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38. ACM, 1986.

Herman Heine Goldstine and John Von Neumann. *Planning and Coding of Problems for an Electronic Computing Instrument*, volume 2. Institute for Advanced Study Princeton, N. J, 1948.

Georges Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, December 2008. URL http://www.ams.org/notices/200811/tx081101382p.pdf.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39634-2.

Mike Gordon. Proof, Language, and Interaction. chapter From LCF to HOL: A Short History, pages 169–185. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-16188-5. URL http://dl.acm.org/citation.cfm?id=345868.345890.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. Submitted, Mar 2019. URL http://gallium.inria.fr/~fpottier/publis/gueneau-jourdan-chargueraud-pottier-2019.pdf.

Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A Formal Proof of the Kepler Conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017. doi: 10.1017/fmp.2017.1.

Thomas C Hales. The Jordan Curve Theorem, Formally and Informally. *The American Mathematical Monthly*, 114(10):882–894, 2007.

Chris Hawblitzel and Erez Petrank. Automated Verification of Practical Garbage Collectors. In *Proceedings of the 36th Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '09, page 441–453, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583792. doi: 10.1145/1480881.1480935. URL https://doi.org/10.1145/1480881.1480935.

Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. *Real World OCaml*. OReilly, 2014.

Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, pages 523–536, 2013.

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A Theory of Indirection via Approximation. In *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, pages 171–185, 2010. URL http://msl.cs.princeton.edu/indirection.pdf.

F. D. Kamareddine. *Thirty Five Years of Automating Mathematics*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 9048164400, 9789048164400.

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*, volume 4. Springer US, 2000a. ISBN 978-1-4757-3188-0. doi: 10.1007/978-1-4757-3188-0.

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000b. ISBN 0792377443.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL `http://doi.acm.org/10.1145/1629575.1629596`.

William Kneale and Martha Kneale. *The Development of Logic*. Oxford University Press, 1985. ISBN 0198247737.

Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 12 2015.

Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2nd edition, 2016. ISBN 9783662504963. doi: 10.1007/978-3-662-50497-0. URL `https://doi.org/10.1007/978-3-662-50497-0`.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL `http://doi.acm.org/10.1145/2535838.2535841`.

Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic: Semantics, Soundness, and a Verified Implementation. *Journal of Automated Reasoning*, 56(3):221–259, Mar 2016. ISSN 0168-7433. doi: 10.1007/s10817-015-9357-x. URL `http://dx.doi.org/10.1007/s10817-015-9357-x`.

Xavier Leroy et al. The CompCert Verified Compiler. *Documentation and User's Manual. INRIA Paris-Rocquencourt*, 2012.

N.G. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940.

Phil Ling. Cern Uses Static Analysis to Find More Than 40,000 Software Defects. *eeNews Europe*, Sep 2011. URL https://bit.ly/2S9phff.

Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 Flight 501 Failure Report by the Inquiry Board, 1996. URL http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

Lena Magnusson and Bengt Nordström. The ALF Proof Editor and Its Proof Engine. In *Proceedings of the International Workshop on Types for Proofs and Programs*, TYPES '93, pages 213–237, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. ISBN 3-540-58085-9. URL http://dl.acm.org/citation.cfm?id=189973.189982.

Per Martin-Löf. An Intuitionistic Theory of Types. *Twenty-Five Years of Constructive Type Theory*, 36:127–172, 1998. Reprinted version of an unpublished report from 1972.

John McCarthy. Towards a Mathematical Science of Computation. In *International Federation for Information Processing Congress*, pages 21–28. North-Holland, 1962.

John McCarthy. A Basis for a Mathematical Theory of Computation. *Studies in Logic and the Foundations of Mathematics*, 35:33–70, 1963.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A General Framework for Certifying Garbage Collectors and Their Mutators. *SIGPLAN Not.*, 42(6):468–479, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250788. URL https://doi.org/10.1145/1273442.1250788.

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A Certified Framework for Compiling and Executing Garbage-Collected Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 273–284, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587943. doi: 10.1145/1863543.1863584. URL https://doi.org/10.1145/1863543.1863584.

Greg Miller. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006. ISSN 0036-8075. doi: 10.1126/science.314.5807.1856. URL http://science.sciencemag.org/content/314/5807/1856.

Robin Milner. Logic for Computable Functions: Description of a Machine Implementation. Technical report, Stanford, CA, USA, 1972.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008a.

Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the Awkward Squad. ACM SIGPLAN International Conference on Functional Programming, 2008b.

Peter Naur. Proof of Algorithms by General Snapshots. *BIT Numerical Mathematics*, 6(4):310–316, Jul 1966. ISSN 1572-9125.

Tobias Nipkow, Gertrud Bauer, and Paula Schultz. Flyspeck I: Tame

Graphs. In *International Joint Conference on Automated Reasoning*, pages 21–35. Springer, 2006.

Benedikt Nordhoff and Peter Lammich. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs*, January 2012. ISSN 2150-914x.

North American Electric Reliability Council. *Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn?* Engineering Case Studies Online. North American Electric Reliability Council, 2004. URL https://books.google.com.sg/books?id=ZfJ7MwEACAAJ.

Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015a. ISSN 1661-8289. doi: 10.1007/s11786-014-0183-z.

Lars Noschinski. *Formalizing Graph Theory and Planarity Certificates*. PhD thesis, Universität München, 2015b.

Peter O'Hearn. Separation Logic. *Commun. ACM*, 62(2):86–95, January 2019. ISSN 0001-0782. doi: 10.1145/3211968. URL http://doi.acm.org/10.1145/3211968.

Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44802-0.

Peter W. O'Hearn. A Primer on Separation Logic (and Automatic Program Verification and Analysis). *Software Safety and Security*, 33:286–318, 2012.

Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, Sep 1989. ISSN 1573-0670.

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-Tolerant Real-Time Garbage Collection. *SIGPLAN Not.*, 45(6):146–159, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806615. URL https://doi.org/10.1145/1809028.1806615.

Robert Pollack. *The Theory of LEGO—A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

Nathaniel Popper. Knight Capital Says Trading Glitch Cost It $440 Million. *The New York Times*, 2012. URL https://nyti.ms/2DwkTxL.

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. doi: 10.1017/CBO9780511626364.

John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578.

John C. Reynolds. A Short Course Separation Logic Fall 2003, 2003. URL http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/wwwaac2003/aac.html.

Henry Gordon Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL http://www.jstor.org/stable/1990888.

Tom Ridge. Graphs and Trees in Isabelle/HOL. 2005.

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A Verified Generational Garbage Collector for CakeML. *Jour-*

nal of Automated Reasoning, 63(2):463–488, August 2019. ISSN 0168-7433. doi: 10.1007/s10817-018-9487-z. URL https://doi.org/10.1007/s10817-018-9487-z.

Ichiro Satoh. Software Testing for Wireless Mobile Computing. *IEEE Wireless Communications*, 11(5):58–64, 2004.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized Verification of Fine-grained Concurrent Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 77–87, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737964. URL http://doi.acm.org/10.1145/2737924.2737964.

Tetsuo Tamai. Formal Treatment of a Family of Fixed-Point Problems on Graphs by CafeOBJ. In *Formal Engineering Methods, 2000. ICFEM 2000. Third IEEE International Conference on*, pages 67–74. IEEE, 2000.

Robert Endre Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28 (3):577–593, July 1981. ISSN 0004-5411. doi: 10.1145/322261.322272. URL https://doi.org/10.1145/322261.322272.

Alfred Tarski et al. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

Max Tegmark. *Our Mathematical Universe: My Quest for the Ultimate Nature of Reality*. Alfred A. Knopf Inc, 1st edition, 2014. ISBN 0307599809.

Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Mathematical Laboratory, Cambridge, 1949.

Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical*

Society, s2-42(1):230–265, 01 1937. ISSN 0024-6115. doi: 10.1112/plms/s2-42.1.230. URL https://dx.doi.org/10.1112/plms/s2-42.1.230.

Carsten Varming and Lars Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science*, 218: 371 – 389, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2008.10.022. URL http://www.sciencedirect.com/science/article/pii/S1571066108004167. Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — A Computer-Checked Library of Univalent Mathematics. available at https://github.com/UniMath/UniMath. URL https://github.com/UniMath/UniMath.

Hao Wang. Toward Mechanical Mathematics. *IBM Journal of Research and Development*, 4(1):2–22, Jan 1960. ISSN 0018-8646. doi: 10.1147/rd.41.0002.

Ray Wang. Formal Verification: The Gap Between Perfect Code and Reality, 2017. URL https://raywang.tech/2017/12/20/Formal-Verification:-The-Gap-between-Perfect-Code-and-Reality/.

Freek Wiedijk. The de Bruijn Factor, 2000. URL http://www.cs.ru.nl/~freek/factor/.

Freek Wiedijk. The QED Manifesto Revisited. *Studies in Logic, Grammar and Rhetoric*, 10(23):121–133, 2007.

Wai Wong. A Simple Graph Theory and Its Application in Railway Signaling. In *HOL Theorem Proving System and Its Applications, 1991., International Workshop on the*, pages 395–409, Aug 1991. doi: 10.1109/HOL.1991.596304.

Mitsuharu Yamamoto, Shin-ya Nishizaki, Masami Hagiya, and Yozo Toda. Formalization of Planar Graphs. In *International Conference on Theorem Proving in Higher Order Logics*, pages 369–384. Springer, 1995.

Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. Formalization of Graph Search Algorithms and its Applications. In *International Conference on Theorem Proving in Higher Order Logics*, pages 479–496. Springer, 1998.

Hongseok Yang. *Local Reasoning for Stateful Programs.* PhD thesis, University of Illinois, 2001.

Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2007–2020, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3133974. URL http://doi.acm.org/10.1145/3133956.3133974.