Fakultät für Informatik
Technische Universität München

Verified Quantitative Analysis of Imperative Algorithms

Maximilian P. L. Haslbeck

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. Helmut Seidl

Prüfende der Dissertation:
1. Prof. Tobias Nipkow, PhD
2. Prof. Dr. Jan Hoffmann,
   Carnegie Mellon University

Die Dissertation wurde am 22.02.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.06.2021 angenommen.

Abstract

This thesis aims to extend formal verification of algorithms to not only verify functional correctness but also to analyze the quantitative properties of algorithms. Foremost, I study the running time analysis of imperative programs.

In the first part I collect, review, and formalize three Hoare logics from the literature that can be used to reason about the running time of algorithms. Chronologically, the first one is based on Nielson's Logic and Relational Hoare logic, the second one follows ideas by Carbonenaux et al. and lifts predicates to potentials for a "quantitative Hoare logic", and the third one follows ideas by Atkey to use Separation Logic with Time Credits. I formalize each of them, prove their soundness and correctness, and implement Verification Condition Generators (VCG) for each. Then, I compare those approaches and investigate their interrelations.

Stemming from ideas that arise from the first part, I extend the quantitative Hoare logic to randomized programs, investigate what other quantities besides time can be reasoned about with the same ideas, and present a formalized and abstracted version of Quantitative Separation Logic due to Batz et al..

The second part shows how to use Separation Logic with Time Credits to implement a framework for verifying the analysis of asymptotic time complexity of algorithms. I present two iterations: first, I extend Imperative-HOL with a cost semantics. I build up basic verification infrastructure and tooling that supports reasoning about complexity bounds in one and two variables. Then, I present the analysis of several nontrivial algorithms and data structures. This ranges from dynamic arrays, over red-black trees to the union-find data structure and Fibonacci heaps; and from binary search to Karatsuba's algorithm and the linear-time selection algorithm. Second, I extend an LLVM semantics in a similar manner, introducing multi-currency time credits for a fine-grained analysis that allows one to reason about the consumption of specific LLVM instructions. I also provide basic verification infrastructure and verified data structures, but the verification of algorithms is shifted to a more abstract realm.

While the verification of low-level algorithms and basic data structures may depend on the specifics of the semantics of a programming language, many algorithms' ideas can be captured on a more abstract level. Stepwise refinement allows the separation of reasoning about algorithmic ideas from their implementation details. I show that this idea also works for resource analysis and running time analysis specifically. I extend Lammich's Isabelle Refinement Framework to not only reason about functional correctness, but also the running time, or resource consumption, of programs. To this end, I present the NREST monad, which features a refinement calculus for upper bounds on resource usage. I add the concept of resource currencies and implement automation to assist verification. As a second step, I connect NREST with the concrete

semantics from the second part, allowing the synthesis of concrete implementations from abstract algorithms that preserve and refine the running time bounds from the abstract domain. Finally, I present case studies for Kruskal's minimum spanning tree algorithm and the introspection sort algorithm by Muller.

In the end, I reflect on my approach discussing whether simultaneous verification of functional correctness and quantitative properties is feasible now.

Zusammenfassung

Diese Dissertation wendet formale Verifikation auf die quantitative Analyse von Algorithmen an. Vorranging wird die Laufzeitanalyse von imperativen Programmen untersucht.

Im ersten Teil werden drei Hoare-Logiken für Laufzeitschranken aus der Literatur gesammelt, verglichen und in dem Theorembeweiser Isabelle/HOL formalisiert: die chronologisch Erste formalisiert eine Logik von Nielson, die Zweite, "quantitative Hoare-Logik", basiert auf der Potentialmethode Ideen Carbonneaux' folgend, und die dritte Logik nutzt Atkey's Separationslogik mit Time Credits. Für jede der drei Logiken werden Soundness und Completeness Theoreme, sowie ein Verification Condition Generator (VCG) verifiziert. Abschließend werden die Ansätze verglichen.

Basierend auf den gewonnen Erkenntnissen wird die "quantitative Hoare-Logik" auf probabilistische Algorithmen angewandt und eine generalisierte Version der Quantitativen Separationslogik von Batz et al. präsentiert. Außerdem werden weitere quantitative Eigenschaften neben der Laufzeit untersucht.

Im zweiten Teil wird die Technik Separationslogik mit Time Credits angewandt um praktische Werkzeuge für die Verifikation von asymptotischer Laufzeitkomplexität von Algorithmen zu realisieren. Es werden zwei Iterationen präsentiert. Die Erste erweitert Imperative-HOL um eine Kostensemantik. Zusätzlich werden grundlegende Verifikationsinfrastruktur und Werkzeuge zur Verfügung gestellt, welche die verifizierte Analyse von Komplexitätsschranken in einer und zwei Variablen unterstützt. Diese Methode wird auf einige nicht triviale Fallbeispiele angewandt. Es werden verschiedene Datenstrukturen untersucht: von dynamischen Arrays, über Rot-Schwarz-Bäume zu Union-Find und Fibonacci-Heaps. Mithilfe des Werkzeugs werden Algorithmen von kleiner bis mittlerer Größe verifiziert: von binärer Suche, über Karatsuba's Algorithmus bis zum linearzeit Median-of-Medians Algorithmus. Die zweite Iteration erweitert eine LLVM Sematik analog. Zusätzlich werden Time Credits mit Währungen eingeführt, welche eine feinere Analyse der Laufzeit ermöglicht die einzelne LLVM Instruktionen zählt. Für diese Sprache werden auch elementare Werkzeuge bereitgestellt und einfache Datenstrukturen verifiziert. Die Analyse von Algorithmen wird aber in einen abstrakteren Teil ausgelagert: das Thema des dritten Teils dieser Dissertation.

Während die Verifikation systemnaher Algorithmen und elementarer Datenstrukturen von den Eigenheiten der Semantik der Programmiersprache abhängen in der sie implementiert sind, können viele algorithmische Ideen auf einer abstrakteren Ebene erfasst werden. Schrittweise Verfeinerung erlaubt es in Beweisen die Argumentation über algorithmische Ideen von Implementierungdetails zu trennen. In dem dritten Teil dieser Dissertation wird diese Technik auf die Analyse von Ressourcenverbrauch, und im speziellen Laufzeit, ausgeweitet. Es wird das Isabelle Refinement Framework

von Lammich erweitert, um nicht nur die funktionale Korrektheit sondern auch die Laufzeitkomplexität von Programmen zu beweisen. Dazu wird die NREST-Monade präsentiert zusammen mit einem Verfeinerungskalkül für obere Schranken, das Konzept von Ressourcenwährungen eingeführt und unterstützende Automatisierung implementiert. In einem zweiten Schritt wird die NREST-Monade mit den Semantiken aus dem zweiten Teil verbunden, um die automatisierte Synthese konkreter Implementierungen von abstrakten Algorithmen zu ermöglichen. Diese Synthese erhält die funktionale Korrektheit sowie die Laufzeitschranken. Abschließend werden größere Fallstudien präsentiert: Kruskal's Algorithmus für minimale Spannbäume und Muller's Introsort-Algorithmus.

Zum Schluss, wird über den Ansatz und die Werkzeuge reflektiert und diskutiert ob damit die gleichzeitige Verifikation von funktionaler Korrektheit und quantitativen Eigenschaften in Theorembeweisern realisierbar ist.

Acknowledgments

Foremost, I want to thank Tobias Nipkow for introducing me to Isabelle and formal verification, supervising all my theses, and granting me a high degree of freedom in my research. Thank you for being such an enthusiastic teacher, a very good co-author, and an exceptional magnet for good people. Especially, it was a pleasure to discuss research ideas and debate wording with you.

I want to thank Jan Hoffmann who indirectly had a great influence on this thesis by propagating the potential method through his papers and by supervising Quentin Carbonneaux. I'm honored to have you on my thesis committee.

I was very much inspired by the work of Armaël Guéneau, Arthur Charguéraud and François Pottier. Thank you for writing your "fistful of dollars" paper and discussing its ideas with me. Furthermore, I got to know the "zen of the potential method" over Quentin Carbonneaux. Thank you, Armaël and Quentin, for visiting our group in Munich and for all the discussions in various occasions.

Thanks to Joost-Pieter Katoen and his group in Aachen, who do impressive work on probabilistic programs. Thanks to Christoph Matheja, Kevin Batz and Benjamin Kaminski for all the discussions and your invitation to Aachen.

Walter Guttmann visited us at the Chair of Logic and Verification and pointed out to me what quantales are. Thanks for that calm and insightful conversation.

Thanks to my other co-authors. First Bohua Zhan: I remember it was a lot of fun extending Imperative-HOL with a cost semantics, verifying algorithms, and writing the IJCAR'18 paper with you. In hindsight, you have been the right visitor at the right time. I'm glad we took on that project together. I'm very much indebted to Peter Lammich, who introduced me not only to the Isabelle Refinement Framework, but also to Separation Logic in general and all the obscure details of the Sepref tool. You were patient enough to discuss my ideas and bold enough to collaborate in implementing them. It was a pleasure working together with you. Unfortunately, I was not able to visit you in Manchester during the pandemic, but I will remember the fun I had in our virtual room "Munich-Manchester", pair-proving, drinking coffee, drawing horrible sketches and endlessly debating single phrases of "a few dollars more".

Thank you, Isabelle, for patiently checking all my proofs, and spending so much time with me. Really, this thanks goes out to all who have collaborated in making "the world's most complicated video game" so enjoyable, addictive and rewarding.

Max Kirchmeier, Peter Lammich, Christoph Matheja and especially Simon Wimmer read parts of my thesis and provided helpful comments, for which I thank them.

My research was financed by the Deutsche Forschungsgemeinschaft from the Koselleck grant NI 491/16-1 and I was supported by the Studienstiftung des deutschen Volkes throughout my studies.

Thanks to my colleagues at the Chair of LoVe: Johannes, Fabian, Lars, Julian, Fabian, Jonas, Kevin, Lukas, Manuel, Mohammad, and Simon. Thanks for creating an open and relaxed working environment, for enthusiastically educating the young academics and Isabelle novices, and for interesting discussions of ideas in research and beyond.

A special appreciation goes to my colleagues and friends Simon and Ondřej. Thanks for your critical thinking, for your bold moves and for pushing my focus from the immediate to the important. It's a process.

Danke an Helmut Eckl, den Lehrer der mein Interesse an Mathematik geweckt hat, und an Ludwig, den Freund der meine Begeisterung für Algorithmen und Wettbewerbsprogrammieren entfacht hat.

Ich möchte dem "Möwennest" danken, meinem Zuhause während der zweiten Welle der Corona-Pandemie und der Schreibphase dieser Dissertation. Ihr seid wunderbar.

Vielen Dank an meine Eltern und meinen Bruder: für euren Ansporn und eure Unterstützung. Danke, dass ihre meine Familie seid.

Danke Sam, dass du bei mir bist.

Contents

1 Introduction

1.1 Motivation

With the ever growing importance and size of software systems, the need for ensuring their reliability is rising. Desirable properties that have been extensively studied are termination, fail safety and functional correctness. Only gradually, guarantees of performance and resource consumption are getting into the focus: stack and heap-size bounds are important for reliability of safety-critical systems; worst-case execution time bounds are used for the analysis of real time systems and to avoid side channel attacks.

Techniques and tools [16, 17, 116, 65, 67, 119] have been developed to automatically derive worst-case resource bounds but are limited in several ways. When these automatic tools fail, human interaction is necessary and the use of proof assistants is one way of proceeding.

Proof assistants have been applied to deep theorems in mathematics [38, 43] or fundamental software components [71, 96] but hardly to quantitative algorithm analysis. Recent advances (e. g. [21]) demonstrate that non-trivial results in the area are now within reach. This dissertation aims to further follow this path of research by studying abstract logics for the quantitative analysis of imperative algorithms and developing a formal framework for carrying out this analysis within Isabelle/HOL.

In this thesis, I will tackle that challenge in three steps: first, I collect techniques from the literature that allow for formal reasoning about the running time of imperative algorithms, formalize them in a common setting, prove meta results about them and compare them. Then I implement one of the techniques in Isabelle/HOL, forming a framework for analyzing the asymptotic running time complexity and applying it to medium-sized textbook algorithms and data structures. Finally, to allow verification of larger algorithms, I present a more natural way of structuring algorithm analysis by extending a stepwise refinement approach to also reason about the running time of algorithms.

1.2 Outline

This thesis is organized in three parts: Part I is concerned with exploring abstract techniques for reasoning about the running time of programs. In Part II, I describe how to turn one of those techniques into a practical framework and how to verify medium sized algorithms and data structures with it. In Part III, I show how stepwise refinement can be extended to reasoning about resource consumption of programs and how it can be used to structure larger algorithm verifications.

In the main chapter (Chapter 2) of Part I, I first present a review of three Hoare logics for reasoning about the running time of programs. This formal treatment serves as a basis for the rest of the thesis, where I present elaborations of two of those techniques. On the one hand, the *potential method* (Section 2.3) gives rise to a compositional Hoare logic for deterministic imperative programs. I will describe how this approach can be extended to reasoning about probabilistic programs in Chapter 3 and to probabilistic heap-manipulating programs in Chapter 4. On the other hand, we implement the idea of extending Separation Logic with Time Credits (Section 2.4) in practical verification frameworks in Part II and use it as a back end for the refinement approach in Part III.

In the second part, I first describe a concrete framework for reasoning about the asymptotic complexity of imperative programs in Imperative-HOL-Time and present several case studies verified with it (Chapter 5). Then, I present a second iteration that allows fine-grained resource analysis of LLVM programs in Chapter 6 (LLVM-Time).

While in theory the techniques from Part II suffice to verify the analysis of arbitrarily complex algorithms, we need methods to make larger developments manageable. In Part III, I show how stepwise refinement can be extended to resource analysis in order to reach this goal. In Chapter 7, I prepare the ground for the rest of this part: first, by reviewing how algorithms are traditionally presented in standard textbooks in order to distill out requirements for our tool; and, second, by presenting the main components of the Isabelle Refinement Framework, which allows one to use stepwise refinement for verifying functional correctness of algorithms. I will describe the extension of that framework to reasoning about resources in the following chapters. In Chapter 8, I present the generalization of IRF's *nres* monad and gently introduce two extensions of it towards modeling resource consumption. In Chapter 9, I present how to automatically turn algorithms in the abstract monad into concrete implementations in the two languages introduced in Part II. Finally, I present case studies illustrating the applicability of the refinement approach in Chapter 10.

1.3 Publications

Some of the work presented in this thesis has been published as part of four conference papers.

Research Publications

- *Hoare Logics for Time Bounds*,
  Maximilian P. L. Haslbeck and Tobias Nipkow,
  International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2018

- *Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle*,
  Bohua Zhan and Maximilian P. L. Haslbeck,
  International Joint Conference on Automated Reasoning (IJCAR), 2018

- *Refinement with Time - Refining the Run-time of Algorithms in Isabelle/HOL*,
  Maximilian P. L. Haslbeck and Peter Lammich,
  Interactive Theorem Proving (ITP), 2019

- *For a Few Dollars More — Verified Fine-Grained Algorithm Analysis Down to LLVM*,
  Maximilian P. L. Haslbeck and Peter Lammich,
  European Symposium on Programming (ESOP), 2021

The first paper, which appeared in TACAS'18, reviews three Hoare logics from the literature for the analysis of the running time of programs and compares them. This abstract reflection lays the ground for the more applied part of this thesis. The paper forms the basis of Part I which is supplemented by two detours into the probabilistic and the Separation Logic world. The second paper, which appeared in IJCAR'18, picks up one of those Hoare logics — using Separation Logic and Time Credits — and presents a practical framework for verifying asymptotic time complexity of imperative programs in Imperative-HOL-Time. It forms the basis of Chapter 5, which is significantly expanded and illustrated with more case studies.

The paper published in the proceedings of ITP'19 presents a novel combination of refinement with time analysis. It uses Imperative-HOL-Time as a back end for a time-bound preserving synthesis approach. The last paper is yet to appear at ESOP'21 and is a second iteration of that work: it generalizes the theory, introducing the novel concept of resource currencies and establishes LLVM as a back end for the synthesis phase. As both approaches follow the "top-down" approach of stepwise refinement, this material is intertwined and the respective steps are separated into Chapters 6, 8, 9 and 10. In the present thesis the expositions in these chapters have been substantially expanded compared to the original papers.

All the theory presented in this thesis has been verified using Isabelle/HOL. The formalizations are distributed over the following developments.

Isabelle formalizations

[46] AFP entry *Hoare Logics for Time Bounds*,
`https://www.isa-afp.org/entries/Hoare_Time.html`

[49] Verification of expected running time calculus of pGCL programs (verERT),
`https://github.com/maxhaslbeck/verERT`

[48] Verification of quantitative separating connectives,
`https://github.com/maxhaslbeck/QuantSepCon`

[139] Repository *Imperative-HOL-Time*,
`https://github.com/bzhan/Imperative_HOL_Time`

[47] Repository *Nondeterministic Result Monad with Time* (NREST),
`https://github.com/maxhaslbeck/NREST`

[51] Repository Sepref-Time,
`https://www21.in.tum.de/~haslbema/Sepreftime/`

[50] LLVM-Time,
`https://www21.in.tum.de/~haslbema/llvm-time/`

The formalization *Hoare Logics for Time Bounds* [46] in the Archive of Formal Proofs (AFP) provides the formal theories described in Chapter 2.

The repository [49] contains the formalization of the *ert* calculus described in Chapter 3, and the formalization of the quantitative separating connectives and the material from Chapter 4 can be found in the repository [48].

The formalization of the framework for asymptotic analysis of imperative programs in Imperative-HOL-Time described in Chapter 5 can be found in the repository [139].

The formalization of the first iteration of combining refinement with resources using Imperative-HOL-Time as a back end is available here [51]. It depends on the formalization of Imperative-HOL-Time. The formalization of the second iteration that features fine-grained resource analysis targeting LLVM resides in [50]. I pulled out the formalization of the NREST monad in a separate repository [47].

1.4 How to read this thesis

A paragraph with a pencil indicates that the following section — or portions thereof — has appeared previously in another publication (cf. Section 1.3). Unless stated otherwise, in publications with a coauthor, I have contributed the majority of the content.

While this thesis may appear at times caught up in formal details, I would like to highlight the *nuggets of wisdom*[1], i. e. major ideas and meta observations.

Paragraphs marked with a light bulb contain what I consider the *main ideas* of this thesis. Like aphorisms, they might at first not be accessible without context, but they might comprise the essential insights of this work. Enjoy contemplating them.

Paragraphs containing rather technical side remarks that can safely be skipped without loosing the thread are marked with cogs.

In most of this thesis I am describing formalizations in Isabelle/HOL. They have been simplified and streamlined for better presentation. Their actual Isabelle representation can be found in the respective repositories (cf. Section 1.3).

Every chapter ends with a bullet-point summary capturing the main takeaway messages.

[1] Some anonymous ITP reviewer coined that term for me.

Part I

Abstract Reasoning about Running Time of Programs

2 Three Hoare Logics for Time

The content of this chapter is joint work with Tobias Nipkow. The chapter is based on the paper "Hoare Logics for Time Bounds" (Haslbeck and Nipkow [53]).

The overall goal of this thesis is to verify quantitative properties of imperative programs in a theorem prover. When only considering functional correctness of programs, a standard approach is to use Hoare logic [55], and there are many frameworks that are based on that.

It is just natural to try to extend Hoare logics that enable reasoning about running time of algorithms. We have identified three such approaches in the literature. The motivation to study those in more detail are twofold: first, we want to gain experience in how to use those approaches. I need that in order to decide which approach to take as a basis when forming a framework for the quantitative analysis of imperative algorithms. In Part II and III we reuse what we learned from this abstract study. Second, this meta study is not about the automatic analysis of running times but about fundamental questions like soundness and completeness of logics and of verification condition generators (VCGs). The need for such a study becomes apparent when browsing the related literature (e. g. [2, 18, 21]): (formalized) soundness results are of course provided, but completeness of logics and VCGs is missing.

Based on the simple imperative language IMP (Section 2.1), we formalize three logics for time bounds from the literature (Section 2.2, 2.3 and 2.4); we show their soundness and completeness w. r. t. IMP's semantics, discuss specific weaknesses and strengths and study their interrelations (Section 2.5).

We study multiple different Hoare logics because we are interested in different aspects of the logics. First, foremost we want to study soundness and completeness of the logic and also of the verification condition generators. A second aspect is modularity. We would like to combine verified results about subprograms in order to show correctness and running time for larger programs. Therefore we also study a Separation Logic for running time analysis. Third, we study the difference between precise upper bounds and order-of-magnitude upper bounds that abstract from multiplicative constants. In the latter case we speak of "big-O style" logics, following terminology by Nielson [108].

The first logic we study is a big-O style logic due to Nielson [108] (Section 2.2). We improve, formalize and verify this logic and extend it with a VCG whose soundness and completeness we also verify.

In Section 2.3 we formalize a quantitative Hoare logic following ideas by Carbonneaux et al. [16, 18] and extend their work as follows: we prove completeness of the logic and design a sound and complete VCG. Additionally we extend the logic to a big-O style logic.

Following ideas of Atkey [2] and Charguéraud and Pottier [21] we formalize a logic similar to Separation Logic (Section 2.4) for reasoning about concrete running times. We formally prove soundness and completeness.

All proofs have been formalized in Isabelle/HOL [115, 114] and are available online [46].

2.1 Basics

We consider the simple deterministic imperative language IMP. It is used in the textbook "Concrete Semantics" [114] which also studies Hoare logics on IMP and mechanizes the proof its soundness and completeness. We extend IMP with a cost semantics and build our theories upon that.

IMP's commands are built up from Skip, assignment, sequential composition, conditional and While-loop.

$$\begin{aligned} com = {} & \mathit{Skip} \mid \mathit{Assign}\ \mathit{vname}\ \mathit{aexp} \mid \mathit{Seq}\ \mathit{com}\ \mathit{com} \\ & \mid \mathit{If}\ \mathit{bexp}\ \mathit{com}\ \mathit{com} \mid \mathit{While}\ \mathit{bexp}\ \mathit{com} \end{aligned}$$

Program states are functions from variables (*vname*) to values. By default c is a command and s a state. Evaluation of a Boolean or arithmetic expression a in state s is denoted by $[\![a]\!]_s$.

We have defined a big-step semantics that counts the consumed time during execution: Skip, assignment and evaluation of Boolean expressions require one time unit. The precise definition of the semantics is routine. We write $(c,\ s) \stackrel{t}{\Rightarrow} s'$ to mean that starting command c in state s terminates after time t in state s'.

Given a pair $(c,\ s)$, $\downarrow(c,\ s)$ means that the computation of c starting from s terminates, $\downarrow_S(c,\ s)$ then denotes the final state, and $\downarrow_T(c,\ s)$ the execution time.

In the following three sections we study and extend three different Hoare logics: a classical one based on [108], one using potentials [16] and one based on Separation Logic with Time Credits [2].

2.2 Nielson

Nielson and Nielson [108] present a Hoare logic to prove the "order of magnitude of the execution time" of a program (which we call "big-O style"). They reason about triples of the form $\{P\}\ c\ \{e \Downarrow Q\}$ where P and Q are assertions and e is a time bound. The intuition is the following: if the execution of command c is started in a state satisfying P then it terminates in a state satisfying Q after $O(e)$ time units, i. e. the execution time has order of magnitude e. Note that e is evaluated in the state before executing c.

Throughout this chapter we rely on what is called a *shallow* embedding of assertions and time bounds: there is no concrete syntactic representation of assertions and time bounds but they are merely functions in HOL, our ambient logic. They map states to truth values and natural numbers.

A complication in reasoning about execution time comes from the fact that one needs to combine time bounds that refer to different points in the execution, for example when adding time bounds in a sequential composition.

$$\models_1 \{P\}\ c_1\ \{e_1 \Downarrow Q\} \wedge \models_1 \{Q\}\ c_2\ \{e_2 \Downarrow R\} \Longrightarrow \models_1 \{P\}\ c_1\ ;\ c_2\ \{e_1 + e_2 \Downarrow R\}$$

The time bounds e_1 and e_2 are evaluated in the prestates of c_1 and c_2 respectively. However, the time bound $e_1 + e_2$ in the Hoare triple of the consequence is evaluated in the prestate of c_1, but this is problematic as the application of c_1 may have changed the value of e_2.

This difficulty can be overcome with *logical variables* that enable us to transport time bounds from the prestate to the poststate of a command. We formalize logical variables by modeling assertions as functions of two states, the state of the logical variables (typically l) and the state of the program variables (typically s).

The validity of Nielson's triples is formally defined as follows:

$$\models_1 \{P\}\ c\ \{e \Downarrow Q\} = (\exists k.\ \forall l\ s.\ P\ l\ s \implies (\exists t\ s'.\ (c,\ s) \stackrel{t}{\Rightarrow} s' \wedge\ Q\ l\ s' \wedge\ t \leq k \cdot\ e\ s))$$

The "big-O-style" abstraction from multiplicative constants is baked into this definition. The time bound e is the order of magnitude that bounds the running time of the program.

The Hoare logic below needs to generate "fresh" logical variables. Thus we need to express which logical variables are already used. This is called the *support* of an assertion. Because assertions are merely functions, the support is defined semantically:

$$\mathit{support}\ Q = \{x \mid \exists l_1\ l_2\ s.\ (\forall y.\ y \neq x \implies l_1\ y = l_2\ y) \wedge\ Q\ l_1\ s \neq Q\ l_2\ s\ \}$$

Our Hoare logic is shown in Figure 2.1. It is largely a formalization of the system in [108, Table 10.4] but with two important changes: we have simplified rule *While* (details below) and we have replaced the consequence rule by conseq_K, an adaptation of Kleymann's stronger consequence rule [72]. The rules *conseq* and *const* are derived from it. Note that the latter two rules suffice for a sound and complete Hoare logic, but our proof of completeness of the VCG needs conseq_K.

Now we discuss the rules in Figure 2.1. Rules *Skip*, *Assign*, *If* and *conseq* are straightforward. Note that **1** is the time bound $\lambda s.\ 1$ and $+$ is lifted to time bounds pointwise. The notation $s[a/x]$ is short for "s with x mapped to $[\![a]\!]_s$".

Now consider rule *Seq*. Given $\{P\}\ c_1\ \{e_1 \Downarrow Q\}$ and $\{Q\}\ c_2\ \{e_2 \Downarrow R\}$ one may want to conclude $\{P\}\ c_1\ ;\ c_2\ \{e_1 + e_2 \Downarrow R\}$. As pointed out above, $e_1 + e_2$ does not lead to the correct result, as c_1 could have altered variables e_2 depends on. In order to adapt e_2 for the changes that occur in c_1, we use a shifted time bound e_2', and leave as a proof goal to show that the value of e_2' in the prestate is an upper bound on e_2 in the poststate of c_1. Rule *Seq* relates e_2' and e_2 through a fresh logical variable u that is equated with the value of e_2' in the prestate of c_1. The time bound e in the conclusion must be an upper bound of $e_1 + e_2'$.

In the *const* rule, the time bound can be reduced by a constant factor. Note that we split up Nielson's cons_e rule into *conseq* and *const*.

$$\frac{}{\vdash_1 \{P\}\ \text{Skip}\ \{\mathbf{1} \Downarrow P\}}\ \textit{Skip} \qquad \frac{}{\vdash_1 \{\lambda l\ s.\ P\ l\ (s[a/x])\}\ x := a\ \{\mathbf{1} \Downarrow P\}}\ \textit{Assign}$$

$$\frac{\vdash_1 \{\lambda l\ s.\ P\ l\ s \wedge [\![b]\!]_s\}\ c_1\ \{e \Downarrow Q\} \qquad \vdash_1 \{\lambda l\ s.\ P\ l\ s \wedge \neg[\![b]\!]_s\}\ c_2\ \{e \Downarrow Q\}}{\vdash_1 \{P\}\ \text{IF}\ b\ \text{THEN}\ c_1\ \text{ELSE}\ c_2\ \{e + \mathbf{1} \Downarrow Q\}}\ \textit{If}$$

$$\frac{\begin{array}{c}\vdash_1 \{\lambda l\ s.\ P\ l\ s \wedge e_2'\ s = l\ u\}\ c_1\ \{e_1 \Downarrow \lambda l\ s.\ Q\ l\ s \wedge e_2\ s \leq l\ u\} \\ \vdash_1 \{Q\}\ c_2\ \{e_2 \Downarrow R\} \\ (\forall l\ s.\ P\ l\ s \Longrightarrow e_1\ s + e_2'\ s \leq e\ s) \qquad u \notin \textit{support}\ P \qquad u \notin \textit{support}\ Q\end{array}}{\vdash_1 \{P\}\ c_1\ ;\ c_2\ \{e \Downarrow R\}}\ \textit{Seq}$$

$$\frac{\begin{array}{c}\vdash_1 \{\lambda l\ s.\ I\ l\ s \wedge [\![b]\!]_s \wedge e'\ s = l\ u\}\ c\ \{e'' \Downarrow \lambda l\ s.\ I\ l\ s \wedge e\ s \leq l\ u\} \\ (\forall l\ s.\ I\ l\ s \wedge [\![b]\!]_s \Longrightarrow e\ s \geq 1 + e'\ s + e''\ s) \\ (\forall l\ s.\ I\ l\ s \wedge \neg[\![b]\!]_s \Longrightarrow e\ s \geq 1) \qquad u \notin \textit{support}\ I\end{array}}{\vdash_1 \{I\}\ \text{WHILE}\ b\ \text{DO}\ c\ \{e \Downarrow \lambda l\ s.\ I\ l\ s \wedge \neg[\![b]\!]_s\}}\ \textit{While}$$

$$\frac{\begin{array}{c}\forall l\ s.\ P'\ l\ s \Longrightarrow P\ l\ s \\ \vdash_1 \{P\}\ c\ \{e \Downarrow Q\} \\ \forall l\ s'.\ Q\ l\ s' \Longrightarrow Q'\ l\ s'\end{array}}{\vdash_1 \{P'\}\ c\ \{e' \Downarrow Q'\}}\ \textit{conseq} \qquad \frac{\begin{array}{c}\exists k. \forall l\ s.\ P\ l\ s \Longrightarrow e\ s \leq k \cdot e'\ s \\ \vdash_1 \{P\}\ c\ \{e \Downarrow Q\}\end{array}}{\vdash_1 \{P\}\ c\ \{e' \Downarrow Q\}}\ \textit{const}$$

$$\frac{\begin{array}{c}\exists k. \forall l\ s.\ P'\ l\ s \Longrightarrow (e\ s \leq k \cdot e'\ s \wedge (\forall s'. \exists l'.\ P\ l's \wedge (Q\ l's' \Longrightarrow Q'\ l\ s'))) \\ \vdash_1 \{P\}\ c\ \{e \Downarrow Q\}\end{array}}{\vdash_1 \{P'\}\ c\ \{e' \Downarrow Q'\}}\ \textit{conseq}_K$$

Figure 2.1: Hoare logic for reasoning about order of magnitude of execution time

Our rule *While* is a simplification of the one in [108]. The latter is an extension with time of the "standard" While-rule for total correctness where a variable decreases with each loop iteration. However, once you have time, you no longer need that variable and we removed it. The key constraint in rule *While* is $e \geq 1 + e' + e''$. It can be explained by unfolding the loop once. The time e to execute the whole loop must be an upper bound for the time e'' to execute the loop body plus the time e' to execute the remaining loop iteration. The additional one accounts for evaluation of the loop guard b. The time e' to execute the remaining loop iterations is obtained from e by (intuitively) an application of rule *Seq*: in the first premise a fresh logical variable u is used to pull e back over c, resulting in e'. The rest of rule *While* is standard.

Soundness Soundness of the calculus can be shown by induction on the derivation of $\vdash_1 \{P\}\ c\ \{e \Downarrow Q\}$:

Theorem 1 (Soundness of $\vdash_1$)**.** $\vdash_1 \{P\}\ c\ \{e \Downarrow Q\} \implies \models_1 \{P\}\ c\ \{e \Downarrow Q\}$

Proof. By induction on the derivation of $\vdash_1 \{P\}\ c\ \{e \Downarrow Q\}$: the cases *Skip* and *Assign* are automatic. In the *Seq* and *If* case we choose the maximum of the constant factors obtained from the Hoare triples of the subprograms as constant factor of the validity of the entire command, the rest is straightforward. In the *While* case we get a constant factor k from the loop body, and choose $k + 1$ as the multiplicative constant, then we show the generalized goal

$$\begin{aligned} &l\ x = n \implies I\ l\ s \implies \exists t\ p.\ (\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s) \Rightarrow p \Downarrow t \\ &\qquad \wedge\ p \leq (k{+}1) * e\ s \wedge I\ (l(x := 0))\ t \end{aligned}$$

by induction on the natural number n. The base case is automatic, the step case involves some bookkeeping and adaptation of the logical environment, and the freshness of the counting variable x in the logical environment. The *conseq* case, obtains a k_1 from the Hoare Triple in the premise, as well as a k_2 from the side condition. Choosing the product $k_1 * k_2$, and some reasoning involving the fact that the big-step semantics is deterministic proves the goal. □

Completeness Our completeness proof follows the general pattern for Hoare logics: define a weakest precondition operator *wp* and show that the triple $\{wp\ c\ Q\}\ c\ \{Q\}$ is derivable. In our setting *wp* is defined like this:

$$wp\ c\ Q = (\lambda l\ s.\ \exists t\ s'.\ (c,\ s) \overset{t}{\Rightarrow} s' \wedge\ Q\ l\ s')$$

and we show derivability of the following triple that also takes time into account:

Lemma 2. $\mathit{finite}\ (\mathit{support}\ Q) \implies \vdash_1 \{wp\ c\ Q\}\ c\ \{\lambda s.\ \downarrow_T (c,\ s) \Downarrow Q\}$

As we need fresh logical variables for rules *Seq* and *While*, we assume that the set of logical variables Q depends on is finite.

Proof. by structural induction on c. *Skip*, *Assign*, and If are automatic or routine; for the *Seq* case we choose a fresh variable x, specialize the first induction hypothesis with $Q = (\lambda l\ s.\ wp\ c_2\ Q\ l\ s \wedge \downarrow_T (c_2, s) = l\ x)$ and together with the second induction hypothesis we can apply the *Seq* rule, side conditions are solved automatically. For the *While* case essentially we choose a fresh variable x and then use the invariant $wpw\ b\ c\ (l\ x)\ Q$, which is the weakest precondition of the loop when unfolded a fixed number of times, and the induction hypothesis for the loop body in the *While* rule, side conditions are solved routinely. □

It is instructive to observe that for this proof, only the Hoare rules *Skip* to *conseq* are needed. Neither *const* nor $conseq_K$ are used. Lemma 2 expresses the intuition that it is always possible to derive a triple with the precise execution time as a time bound. Only as a last step an abstraction of multiplicative constants and over-approximation of the time bound is necessary. This shows that for every valid triple one can first deduce a correct upper bound for the running time, only to get rid of a multiplicative constant in a final application of the *const* rule. In the end, Lemma 2 implies completeness:

Theorem 3 (Completeness of $\vdash_1$)**.**

$$[\![\ finite\ (support\ Q);\ \models_1 \{P\}\ c\ \{e \Downarrow Q\}\ ]\!] \implies \vdash_1 \{P\}\ c\ \{e \Downarrow Q\}$$

In particular, we can now apply the above observation about the shape of derivations of valid triples to provable ones, by soundness: in any derivation one can pull out all applications of *const* and combine them into a single one at the very root of the proof tree. We will observe the very same principle when studying the quantitative Hoare logic in Section 2.3.

Verification Condition Generator Showing validity of $\{P\}\ c\ \{e \Downarrow Q\}$ now boils down to applying the correctly instantiated rules of the Hoare logic and proving their side conditions. The former is a mechanical task, which is routinely automated by a verification condition generator, while the latter is left to an automatic or interactive theorem prover.

We design a VCG that collects the side conditions for an annotated program. While for classical Hoare logic it suffices to annotate a loop with an invariant I, for reasoning about execution time we introduce two more annotations for the following reason.

Now consider rule *Seq* in Figure 2.1. When applying the rule to a proof goal $\vdash_1 \{P\}\ c_1\ ;\ c_2\ \{e \Downarrow R\}$ we need to instantiate the variables P, Q, e_1, e_2, and $e_2{}'$. As for classical Hoare logic, Q is chosen to be the weakest preconditions of c_2 w. r. t. R, which can be calculated if the loops in c_2 are annotated by invariants. (Analogously for P being the weakest precondition of c_1 w. r. t. Q). Similarly, when annotating the loops in c_1 and c_2 with time bounds E, time bounds e_1 and e_2 can be constructed. Finally, $e_2{}'$ can be determined if the evolution of e_2 through c_1 is known. For straight-line programs, this can be deduced, only for loops a state transformer S has to be annotated. An annotated loop then has the form $\{I, S, E\}\ WHILE\ b\ DO\ C$ where I is the invariant and S and E are as described above.

$$
\begin{array}{ll}
\mathit{pre}\ \text{Skip}\ Q = Q & \mathit{post}\ \text{Skip}\ s = s \\
\mathit{pre}\ (x := a)\ Q = (\lambda ls.\ Q\ l\ (s[a/x])) & \mathit{post}\ (x := a)\ s = s[a/x] \\
\mathit{pre}\ (C_1; C_2)\ Q = \mathit{pre}\ C_1\ (\mathit{pre}\ C_2\ Q) & \mathit{post}\ (C_1; C_2)\ s = \mathit{post}\ C_2\ (\mathit{post}\ C_1\ s) \\
\mathit{pre}\ (\mathit{Conseq}\ \{P', \_, \_\}\ C)\ Q = P' & \mathit{post}\ (\mathit{Conseq}\ \{\_, \_, \_\}\ C) = \mathit{post}\ C \\
\mathit{pre}\ (\text{IF}\ b\ \text{THEN}\ C_1\ \text{ELSE}\ C_2)\ Q\ l\ s = & \mathit{post}\ (\text{IF}\ b\ \text{THEN}\ C_1\ \text{ELSE}\ C_2)\ s = \\
\ \text{if}\ [\![b]\!]_s\ \text{then}\ \mathit{pre}\ C_1\ Q\ l\ s\ \text{else}\ \mathit{pre}\ C_2\ Q\ l\ s & \ \text{if}\ [\![b]\!]_s\ \text{then}\ \mathit{post}\ C_1\ s\ \text{else}\ \mathit{post}\ C_2\ s \\
\mathit{pre}\ (\{I, \_, \_\}\ \text{WHILE}\ b\ \text{DO}\ C)\ Q = I & \mathit{post}\ (\{\_, S, \_\}\ \text{WHILE}\ b\ \text{DO}\ C) = S
\end{array}
$$

$$
\begin{array}{l}
\mathit{time}\ \text{Skip}\ s = 1 \\
\mathit{time}\ (x := a)\ s = 1 \\
\mathit{time}\ (C_1; C_2)\ s = \mathit{time}\ C_1\ s + \mathit{time}\ C_2\ (\mathit{post}\ C_1\ s) \\
\mathit{time}\ (\mathit{Conseq}\ \{\_, \_, \_\}\ C) = \mathit{time}\ C \\
\mathit{time}\ (\text{IF}\ b\ \text{THEN}\ C_1\ \text{ELSE}\ C_2)\ s = \\
\ \text{if}\ [\![b]\!]_s\ \text{then}\ \mathit{time}\ C_1\ s\ \text{else}\ \mathit{time}\ C_2\ s \\
\mathit{time}\ (\{\_, \_, E\}\ \text{WHILE}\ b\ \text{DO}\ C) = E
\end{array}
$$

Figure 2.2: Functions *pre*, *post* and *time*

For our completeness proof of the VCG we also need annotations that correspond to applications of rule conseq_K and record information that cannot be inferred automatically. For that purpose we introduce a new annotated command $\mathit{Conseq}\ \{P', Q, e'\}\ C$ where P', Q and e' are as in rule conseq_K.

We use capital letters, e. g. C to denote annotated commands and $\overline{C}$ is the unannotated version of C stripped of all annotations.

We use three auxiliary functions *pre*, *post* and *time*. Their definitions are shown in Figure 2.2.

The VCG reduces proving a triple $\{P\}\ \overline{C}\ \{e \Downarrow Q\}$ to checking that the annotations really are invariants, upper bounds and correct state transformers. The VCG traverses C and collects all the verification conditions for the loops into a big conjunction. The most interesting case is the loop itself:

$$
\begin{array}{l}
\mathit{vc}\ (\{I, S, E\}\ \text{WHILE}\ b\ \text{DO}\ C)\ Q = \mathit{vc}\ C\ I\ \wedge \\
\qquad (\forall l\ s.\ (I\ l\ s \wedge [\![b]\!]_s \Longrightarrow \mathit{pre}\ C\ I\ l\ s \\
\qquad\qquad \wedge E\ s \geq 1 + E(\mathit{post}\ C\ s) + \mathit{time}\ C\ s \\
\qquad\qquad \wedge S\ s = S(\mathit{post}\ C\ s)) \\
\qquad \wedge\ (I\ l\ s \wedge \neg [\![b]\!]_s \Longrightarrow Q\ l\ s \wedge E\ s \geq 1 \wedge S\ s = s))
\end{array}
$$

First, verification conditions are recursively generated from the loop body C and the invariant I as desired post condition. The invariant and the loop guard must imply preservation of the invariant, the recurrence inequation for the time bound and that the state transformer S obeys the fixpoint equation for loops. When exiting the loop, the post condition must hold, E has to pay for the last test of the loop guard, and S needs to be the identity.

The verification conditions for *Conseq* $\{P',\ Q,\ e'\}\ C$ merely check the side condition of rule $conseq_K$.

$$vc\ (Conseq\ \{P', Q, e'\}\ C)\ Q' = vc\ C\ Q\ \wedge$$
$$\exists k.\ \forall l\ s.\ P'\ l\ s \Longrightarrow time\ C\ s \leq k \cdot e'\ s$$
$$\wedge\ \forall t.\ \exists l'.\ pre\ C\ Q\ l'\ s \wedge (Q\ l'\ t \Longrightarrow Q'\ l\ t)$$

The remaining equations for vc are straightforward:

$$vc\ \text{Skip}\ Q = True$$
$$vc\ (x := a)\ Q = True$$
$$vc\ (C_1; C_2)\ Q = (vc\ C_1\ (pre\ C_2\ Q) \wedge vc\ C_2\ Q)$$
$$vc\ (\text{IF}\ b\ \text{THEN}\ C_1\ \text{ELSE}\ C_2)\ Q = (vc\ C_1\ Q \wedge vc\ C_2\ Q)$$

Theorem 4 (Soundness of vc)**.** *Let C and Q involve only finitely many logical variables. Then $vc\ C\ Q$ together with $\exists k.\ \forall l\ s.\ P\ l\ s \Longrightarrow pre\ C\ Q\ l\ s \wedge time\ C\ s \leq k \cdot e\ s$ imply $\vdash_1 \{P\}\ \overline{C}\ \{e \Downarrow Q\}$.*

That is, for proving $\vdash_1 \{P\}\ \overline{C}\ \{e \Downarrow Q\}$ one has to show the verification conditions, that P implies the weakest precondition (as computed by *pre*), and that the running time (as computed by *time*) is in the order of magnitude of e.

Now we come to the *raison d'être* of the stronger consequence rule $conseq_K$: the completeness proof of our VCG. The other proofs in this section only require the derived rules *conseq* and *const*. Our completeness proof of the VCG builds annotated programs that contain a *Conseq* construct for every *Seq* and *While* rule. The annotations of *Conseq* enable us to adapt the logical state; without this adaptation we failed to generate true verification conditions.

Theorem 5 (Completeness of vc)**.** *If $\vdash_1 \{P\}\ c\ \{e \Downarrow Q\}$ is true then there is a C such that $\overline{C} = c$, $vc\ C\ Q$ is true and $\exists k.\ \forall l\ s.\ P\ l\ s \Longrightarrow pre\ C\ Q\ l\ s \wedge time\ C\ s \leq k \cdot e\ s$.*

That is, if a triple $\vdash_1 \{P\}\ c\ \{e \Downarrow Q\}$ is provable then c can be annotated such that the verification conditions are true, P implies the weakest precondition (as computed by *pre*) and the running time (as computed by *time*) is in the order of magnitude of e.

Annotating loops with a correct S is troublesome, as it captures the semantics of the whole loop. Luckily S only needs to be correct for "interesting" variables, i.e. variables that occur in time bounds that need to be pulled backward through the loop body. Often these variables are not modified by a command. We implemented an optimized VCG that keeps track of which variables are of interest and requires S to be correct only on those. We also showed its soundness and completeness. Further details can be found in the formalization.

2.3 Carbonneaux - Quantitative Hoare Logic

Now we turn to the second Hoare logic we formalized. The main idea we use from Carbonneaux *et al.* [16] is to generalize predicates ($state \Rightarrow \mathbb{B}$) in Hoare triples to *potentials* ($state \Rightarrow \mathbb{N}_\infty$). That is, Hoare triples are now of the form $\{\Phi\}\ c\ \{\Psi\}$ where Φ

$$\frac{}{\vdash_2 \{\Phi + \mathbf{1}\}\ \text{Skip}\ \{\Phi\}}\ \textit{Skip} \qquad \frac{}{\vdash_2 \{\lambda s.\ 1 + \Phi(s[a/x])\}\ x := a\ \{\Phi\}}\ \textit{Assign}$$

$$\frac{\vdash_2 \{\Phi + \uparrow b\}\ c_1\ \{\Psi\} \qquad \vdash_2 \{\Phi + \uparrow(\neg b)\} c_2 \{\Psi\}}{\vdash_2 \{\Phi + \mathbf{1}\}\ \text{IF}\ b\ \text{THEN}\ c_1\ \text{ELSE}\ c_2\ \{\Psi\}}\ \textit{If} \qquad \frac{\vdash_2 \{\Phi\}\ c_1\ \{\Psi\} \quad \vdash_2 \{\Psi\}\ c_2\ \{\Pi\}}{\vdash_2 \{\Phi\}\ c_1\ ;\ c_2\ \{\Pi\}}\ \textit{Seq}$$

$$\frac{\vdash_2 \{I + \uparrow b\}\ c\ \{I + \mathbf{1}\}}{\vdash_2 \{I + \mathbf{1}\}\ \text{WHILE}\ b\ \text{DO}\ c\ \{I + \uparrow(\neg b)\}}\ \textit{While} \qquad \frac{\vdash_2 \{\Phi\}\ c\ \{\Psi\} \quad \Phi' \geq \Phi \qquad \Psi \geq \Psi'}{\vdash_2 \{\Phi'\}\ c\ \{\Psi'\}}\ \textit{conseq}$$

Figure 2.3: Quantitative Hoare logic

and Ψ are potentials. The resulting logic does not need logical variables. We prove soundness and completeness of that logic and design a sound and complete VCG. Then we extend the logic and VCG to big-O style reasoning.

The "zen of the potential method", i. e. the step from qualitative predicates to quantitative (potential) functions, is central to this thesis and occurs in many places.

Validity of triples involving potentials is defined as follows and is a direct generalization of validity for triples involving predicates:

$$\models_2 \{\Phi\}\ c\ \{\Psi\} = \forall s.\ \Phi\ s < \infty \implies (\exists t\ s'.\ (c,\ s) \overset{t}{\Rightarrow} s' \wedge \Psi\ s' < \infty \\ \wedge\ \Phi\ s \geq t + \Psi\ s')$$

One may interpret the refinement from $\mathbb{B}$ to $\mathbb{N}_\infty$ as follows: infinite potentials are "impossible" and thus correspond to *False*, while finite potentials correspond to *True*. In that way "$\Phi\ s < \infty$" corresponds to "Φ holds in state s". Furthermore, we interpret the difference of the *prepotential* Φ and *postpotential* Ψ as an upper bound on the actual running time. Predicates can be lifted to potentials by mapping *True* to *0* and *False* to ∞. We use the $\uparrow$ symbol for that lifting: $\uparrow P\ s = (\textit{if}\ P\ s\ \textit{then}\ 0\ \textit{else}\ \infty)$, and similarly for Boolean expressions: $\uparrow b\ s = (\textit{if}\ [\![b]\!]_s\ \textit{then}\ 0\ \textit{else}\ \infty)$.

Using this definition we obtain a compositional Sequence rule:

Lemma 6. $\models_2 \{\Phi\}\ c_1\ \{\Psi\} \wedge \models_2 \{\Psi\}\ c_2\ \{\Pi\} \implies \models_2 \{\Phi\}\ c_1\ ;\ c_2\ \{\Pi\}$

Proof. To prove this rule it suffices to observe that, from the two Hoare triples in the premises we obtain $\Phi\ s \geq t_1 + \Psi\ s'$ and $\Psi\ s' \geq t_2 + \Pi\ s''$. Then, with monotonicity and associativity of the plus operator it follows that $\Phi\ s \geq (t_1 + t_2) + \Pi\ s''$. □

The rules in Figure 2.3 define the Hoare logic $\vdash_2$ corresponding to $\models_2$. Note that $\Phi \geq \Psi$ is short for $\forall s.\ \Phi\ s \geq \Psi\ s$.

Rules *Skip*, *Assign* and *If* are straightforward. The *1* time unit added to the prepotential pays for, respectively, Skip, assignment and the evaluation of the Boolean

expression. The *conseq* rule also looks familiar, only that $\Longrightarrow$ has been replaced by its counterpart on potentials: $\geq$. You can think of a bigger potential "implying" a smaller one. Also remember that *False* corresponds to ∞.

For the *While* rule, assume one can derive that, having the potential I and a true guard b before the execution of b implies a postpotential one more than the invariant I (the plus one is needed for the upcoming evaluation of the guard, which incurs cost *1*), then one can conclude that, starting the loop with potential $I+\mathbf{1}$ (again the plus one pays for the evaluation of the guard), the loop terminates with a potential equal to I and the negation of the guard holds in the final state. Although this rule resembles the *While* rule for partial correctness, the decreasing potential actually also ensures termination.

Soundness Let us prove soundness of the set of Hoare Rules:

Theorem 7 (Soundness of $\vdash_2$)**.** $\vdash_2 \{\Phi\}\ c\ \{\Psi\} \implies \models_2 \{\Phi\}\ c\ \{\Psi\}$

Proof. Again by induction on the derivation of $\vdash_2 \{\Phi\}\ c\ \{\Psi\}$, we must show that every rule preserves validity. This is straightforward for *Skip*, *Assign*, *If* and *conseq*. The *Seq* case is essentially Lemma 6.

The most interesting case is *While*: we assume the induction hypothesis (IH_1) $\models_2 \{I + \uparrow b\}\ c\ \{I + \mathbf{1}\}$. First, we prove for arbitrary s that if $I\ s$ has a finite value $n \in \mathbb{N}$, there is some amount of time t and state s' such that $(WHILE\ b\ DO\ c, s) \overset{t}{\Rightarrow} s'$ and $t + (I\ s' + \uparrow(\neg\ b)\ s') \leq I\ s + 1$. The proof proceeds by well founded induction on the value n. There are two cases.

If the guard is *False* in s, we have $s = s'$ and $t = 1$; then we can directly use the *WhileFalse* rule from the big-step semantics and the second conjunct follows trivially.

If the guard is *True* in s, we can establish the prepotential $(I\ s + \uparrow b\ s < \infty)$ to use the outer induction hypothesis (IH_1) to obtain the state s' and the time t that passed while executing the loop body c once. In that case, we obtain the facts (a) $(c, s) \overset{t}{\Rightarrow} s'$ and (A) $t + I\ s' + 1 \leq I\ s$. It is easy to see that $I\ s' < I\ s$ follows from (A). As $Is = n$ we can now find a value $n' \in \mathbb{N}$ such that $I\ s' = n'$ and it follows $n' < n$. Thus we can use the inner induction hypothesis for the smaller value n' and obtain for the tail of the loop: execution of the tail of the loop ends in state s'' and consumes t' time, we get (b) $(WHILE\ b\ DO\ c, s') \overset{t'}{\Rightarrow} s''$ and (B) $t' + (Is'' + \uparrow(\neg b)s'') \leq Is' + 1$. It remains to verify that $(1 + t + t') + (I\ s'' + \uparrow(\neg\ b)\ s'') \leq I\ s + 1$, i. e. that we had enough potential in the beginning to pay for the initial guard evaluation (*1*), the first unrolling of the body c (t) and the rest of the loop (t'),as well as the potential after the whole execution of the loop ($I\ s''$):

$$\begin{aligned}(1 + t + t') + (I\ s'' + \uparrow(\neg b)s'') &\leq (1 + t) + I\ s' + 1 && \text{(B)}\\ &\leq I\ s + 1 && \text{(A)}\end{aligned}$$

From (a) and (b) we finally can conclude $(WHILE\ b\ DO\ c, s) \overset{t+t'}{\Longrightarrow} s''$ and finish off the inner induction. We have now established this lemma:

$$\forall s.\ I\ s = n \implies \exists t\ s'.\ (WHILE\ b\ DO\ c,\ s) \Rightarrow t \Downarrow s' \\ \wedge\ t + (I\ s' + \uparrow(\neg b)\ s) \leq I\ s + 1$$

To finish off the *While* case, we use that lemma. We can assume $I\ s + 1 < \infty$, which implies the premise and the conclusion matches the current goal. This finishes the proof. □

Completeness For proving completeness, we generalize the *weakest precondition* to the *weakest prepotential*:

$$wp\ c\ \Psi\ s = (\texttt{if} \downarrow (c,\ s)\ \texttt{then} \downarrow_T (c,\ s) + \Psi\ (\downarrow_S (c,\ s))\ \texttt{else}\ \infty)$$

The weakest prepotential maps each state s, from which c will terminate in some state s', to the time needed for this execution plus the potential $\Psi\ s'$ afterwards. A state, which results in a non terminating execution is mapped to ∞.

Verifying that $wp\ c\ Q$ indeed is the weakest prepotential involves only unfolding the definitions and some case distinctions:

Lemma 8. $\models_2 \{\Phi\}\ c\ \{\Psi\} \longleftrightarrow \Phi \geq wp\ c\ \Psi$

Analogous to the classical Hoare logic, as well as to the Nielson Hoare logic we studied in the last section, the weakest prepotential *wp* pulls a postpotential through a command to obtain the corresponding prepotential. We can now formulate nice recursion equations for the different commands:

Lemma 9.

$$wp\ Skip\ = \Psi + \mathbf{1}$$
$$wp\ (x := a)\ \Psi = \Psi[a/x] + \mathbf{1}$$
$$wp\ (c_1\ ;\ c_2)\ \Psi = wp\ c_1\ (wp\ c_2\ \Psi)$$
$$wp\ (If\ b\ c_1\ c_2)\ \Psi = (\lambda s.\ if\ [b]_s\ then\ wp\ c_1\ \Psi\ s\ else\ wp\ c_2\ \Psi\ s) + \mathbf{1}$$
$$wp\ (While\ b\ c)\ \Psi\ = (\lambda s.\ (if\ [b]_s\ then\ wp\ c\ (wp\ (While\ b\ c)\ \Psi)\ s\ else\ \Psi\ s)) + \mathbf{1}$$

Proof. All of them can be proven with the definition of *wp*, the determinism of the IMP language and some case distinctions. □

In fact, *wp* is also a (weakest) prepotential w. r. t. provability:

Lemma 10. $\vdash_2 \{wp\ c\ \Psi\}\ c\ \{\Psi\}$

Proof. We proceed by induction on the command c. Given the equations from Lemma 9, all cases but the *While* case are almost automatic. In the *While* case we choose the right hand side of the equality in the While case of *wp* as an "invariant" and specialize the induction hypothesis with $wp\ (WHILE\ b\ c)\ \Psi$, two applications of the consequence rule employing and Lemma 9 finishes the proof. □

As usual, completeness follows easily from this lemma:

Theorem 11 (Completeness of $\vdash_2$)**.** $\models_2 \{\Phi\}\ c\ \{\Psi\} \implies \vdash_2 \{\Phi\}\ c\ \{\Psi\}$

Verification Condition Generator The simpler *Seq* rule (compared to $\vdash_1$) leads to a more compact VCG. Loops are simply annotated with invariants, which now are potentials. No *Conseq* annotations are required.

Function $pre\ C\ \Psi$ determines the weakest prepotential of an annotated program C and postpotential Ψ. Its definition is by recursion on annotated commands and refines our earlier *pre* on predicates.

The VCG recursively traverses the command and collects the verification conditions at the loops (we omit the other cases of *vc*):

$$vc\ (\{I\}\ WHILE\ b\ DO\ C)\ \Psi = I + \uparrow b \geq pre\ C\ (I + \mathbf{1}) \\ \wedge\ I + \uparrow(\neg b) \geq \Psi \wedge vc\ C\ (I + \mathbf{1})$$

The two first conjuncts express invariant preservation and that the invariant "implies" the postcondition when exiting the loop. Soundness of the VCG is established by induction on the command.

Lemma 12 (Soundness of *vc*)**.** *If we can show the verification conditions* $vc\ C\ \Psi$ *and that we have at least as much potential as the needed prepotential* ($\Phi \geq pre\ C\ \Psi$) *then we can derive* $\vdash_2 \{\Phi\}\ \overline{C}\ \{\Psi\}$.

Completeness of the VCG can be paraphrased like this: if we can derive the Hoare Triple $\vdash_2 \{\Phi\}\ c\ \{\Psi\}$, we can find an annotation for c such that the verification conditions are true and Φ "implies" the prepotential.

Lemma 13 (Completeness of *vc*)**.**

$$\vdash_2 \{\Phi\}\ c\ \{\Psi\} \implies \exists C.\ \overline{C} = c \wedge vc\ C\ Q \wedge \Phi \geq pre\ C\ \Psi$$

Constant factors As for the Nielson system we can extend the quantitative Hoare logic to reason about the order of magnitude of execution time. We generalize our notion of validity from $\models_2$ to $\models_{2'}$:

$$\models_{2'} \{\Phi\}\ c\ \{\Psi\} = \exists k > 0.\ \forall s.\ \Phi\ s < \infty \Longrightarrow \exists t\ s'. \begin{cases} (c, s) \Rightarrow t \Downarrow s' \wedge \Psi\ s' < \infty \wedge \\ k \cdot \Phi\ s \geq t + k \cdot \Psi\ s' \end{cases}$$

For intuition, assume Ψ is zero: then the triple is valid if and only if the running time t is bounded by k times the prepotential Φ. This amounts to O-notation.

Correspondingly, we extend the set of Hoare rules $\vdash_2$ in Figure 2.3 to $\vdash_{2'}$ by adding the following rule:

$$\frac{\vdash_{2'} \{\lambda s.\ k \cdot \Phi\ s\}\ c\ \{\lambda s.\ k \cdot \Psi\ s\} \qquad k > 0}{\vdash_{2'} \{\Phi\}\ c\ \{\Psi\}}\ const$$

For re-establishing soundness we can adapt the proof of Theorem 7 by catering for constants and adding one more case for rule *const*.

Theorem 14 (Soundness of $\vdash_{2'}$). $\vdash_{2'} \{\Phi\}\ c\ \{\Psi\} \implies \models_{2'} \{\Phi\}\ c\ \{\Psi\}$

For the completeness proof, nothing changes. We reuse the same *wp* and the proof of $\vdash_{2'} \{wp\ c\ \Psi\}\ c\ \{\Psi\}$ is identical to that of Lemma 10 because we extended the Hoare rules, but not the command language. In particular, this means that the new *const* rule is not used in this proof. The same principle as in Section 2.2 applies: the *const* rule is only used once at the end when showing completeness from $\vdash_{2'} \{wp\ c\ \Psi\}\ c\ \{\Psi\}$:

Theorem 15 (Completeness of $\vdash_{2'}$). $\models_{2'} \{\Phi\}\ c\ \{\Psi\} \implies \vdash_{2'} \{\Phi\}\ c\ \{\Psi\}$

VCG with constants For the VCG we add one more annotated command *Const* $\{k\}$ C (where $k \in \mathbb{N}$, $k > 0$). It signals the application of a *const* rule. We reuse the old definitions of *pre* and *vc* but add new equations for *Const*:

$vc\ (Const\ \{k\}\ C)\ \Psi\ s = (vc\ C\ (\lambda s.\ k \cdot \Psi\ s) \wedge k{>}0)$
$pre\ (Const\ \{k\}\ C)\ \Psi\ s = ediv\ (pre\ C\ (\lambda s.\ k \cdot \Psi\ s)\ s)\ k$

The definition of $vc\ (Const\ \{k\}\ C)\ \Psi$ expresses that the execution of C must leave a potential of $k \cdot \Psi$ instead of just Ψ. The definition of $pre\ (Const\ \{k\}\ C)\ \Psi$ expresses that we pull back a potential of $k \cdot \Psi$ but that in the end we renormalize the prepotential by dividing (function *ediv*) by k. More precisely, *ediv* is integer division which rounds up for non integral results and is lifted to $\mathbb{N}_\infty$.

The soundness and completeness proofs must only be adapted marginally, only some algebraic lemmas about *ediv* are needed.

Wrap Up To summarize this section: we have shown how to generalize predicates to potentials, thus obtaining a compositional Hoare logic. We have extended the Hoare logic to big-O style reasoning and have adapted the calculus and proofs. We also have established sound and complete VCGs for both logics.

One drawback of the quantitative Hoare logic is that it is not modular. Imagine two independent programs c_1 and c_2 that are run one after the other. When reasoning about a subprogram c_1 we need to specify a postpotential that is then used for the following program c_2. If we change c_2, resulting in a changed time consumption, also the analysis for c_1 has do be redone. What we actually would like to do, is to reason about c_1 and c_2 locally and then combine them in a final step. Separation Logic addresses this issue.

2.4 Atkey - Separation Logic with Time Credits

Our last logic follows the idea by Atkey [2] to use Separation Logic in order to reason about the resource consumption of programs.

The principle of "local reasoning" is addressed by separation logic for disjoint heap areas; Atkey [2] uses Separation Logic with Time Credits to reason about the amortized execution time of (imperative) programs.

In this section, we follow his ideas and design a Hoare logic based on Separation Logic. As IMP does not have a heap to reason about, but we want to compare the logic to the two logics we already described, we treat the state of a program as a kind of heap: a *partial state ps* is a map from variable names to values, *dom ps* is the domain of *ps*, we call ps_1 and ps_2 *disjoint* ($ps_1 \# ps_2$) if their domains are, and we can add two partial states to form their disjoint union ($ps_1 + ps_2$).

We adapt evaluation of arithmetic and Boolean expressions, as well as the big-step semantics (now denoted by $\Rightarrow_p$) to partial states. If all necessary variables are in the domain of the partial state *ps*, these new constructs coincide with their counterparts on (full) states. The new big-step semantics rule for assignment for example has an additional premise. All other rules are similar.

$$\frac{vars\ a \cup \{x\} \subseteq dom\ ps}{(x := a, ps) \overset{1}{\Rightarrow}_p ps(x \mapsto [\![a]\!]_{ps})} Assign$$

The new semantics admit a frame rule: we can always add disjoint partial states, without affecting the computation.

Lemma 16. $$\frac{(c, ps_1) \overset{t}{\Rightarrow}_p ps_1' \qquad ps_1 \# ps_2}{(c, ps_1 + ps_2) \overset{t}{\Rightarrow}_p ps_1' + ps_2}$$

In that way we treat the set of variables as resources, on which Separation Logic can work. Additionally, as Atkey proposes, we add time credits as resources: we consider *configurations* (*ps, n*) which are pairs of partial states and natural numbers. Natural numbers, viewed as resources, are always disjoint and can be added; thus they form a separation algebra [14]. A pair of separation algebras is again a separation algebra. For predicates on configurations we thus have the separating conjunction $\star$

$$(P \star Q)(ps, n) = \exists ps_1\ n_1\ ps_2\ n_2. \begin{cases} ps = ps_1 + ps_2 \wedge n = n_1 + n_2 \wedge ps_1 \# ps_2 \wedge \\ P(ps_1, n_1) \wedge Q(ps_2, n_2) \end{cases}$$

meaning that we can split up the configuration into two disjoint configurations; one satisfying P and the other satisfying Q. Our formalization builds on an existing Isabelle/HOL theory of separation algebras by Klein et al. [69].

The validity of a Hoare triple is defined in the following way:

$$\models_3 \{P\}\ c\ \{Q\} = \forall ps\ n.\ P(ps, n) \Longrightarrow \exists ps'\ n'\ t. \begin{cases} (c, ps) \overset{t}{\Rightarrow}_p ps' \wedge \\ n = n' + t \wedge Q(ps', n') \end{cases}$$

We can now state the Hoare rules for this logic, see Figure 2.4. Note that $\$n$ denotes the configuration of an empty partial state and n time credits, $(b \hookrightarrow B)\ ps$ is true, if and only if all variables in b are in the domain of *ps* and b evaluates to B in *ps*. Updating the partial state *ps* with value v for x is denoted by $ps(x \mapsto v)$.

$$\frac{}{\vdash_3 \{\$1\}\ \text{Skip}\ \{\$0\}}\mathit{Skip}$$

$$\frac{}{\vdash_3 \{(\lambda(ps,t).\ \{x\} \cup vars\ a \subseteq dom\ ps \wedge Q\ (ps(x \mapsto [\![a]\!]_{ps}), t)) \star \$1\}\ x := a\ \{Q\}}\mathit{Assign}$$

$$\frac{\begin{array}{c}\vdash_3 \{\lambda(ps,n).\ P(ps,n) \wedge (b \hookrightarrow True)\ ps\}\ c_1\ \{Q\} \\ \vdash_3 \{\lambda(ps,n).\ P(ps,n) \wedge (b \hookrightarrow False)\ ps\}\ c_2\ \{Q\}\end{array}}{\vdash_3 \{(\lambda(ps,n).\ P(ps,n) \wedge vars\ b \subseteq dom\ ps) \star \$1\}\ \text{IF}\ b\ \text{THEN}\ c_1\ \text{ELSE}\ c_2\ \{Q\}}\mathit{If}$$

$$\frac{\vdash_3 \{P\}\ c\ \{Q\}}{\vdash_3 \{P \star F\}\ c\ \{Q \star F\}}\mathit{Frame} \qquad \frac{\vdash_3 \{P\}\ c_1\ \{Q\} \qquad \vdash_3 \{Q\}\ c_2\ \{R\}}{\vdash_3 \{P\}\ c_1\ ;\ c_2\ \{R\}}\mathit{Seq}$$

$$\frac{\vdash_3 \{\lambda(ps,n).\ I(ps,n) \wedge (b \hookrightarrow True)\ ps\}\ c\ \{I \star \$1\}}{\vdash_3 \{(\lambda(ps,n).\ I(ps,n) \wedge vars\ b \subseteq dom\ ps) \star \$1\}\ \text{WHILE}\ b\ \text{DO}\ c\ \{\lambda(ps,n).\ I(ps,n) \wedge (b \hookrightarrow False)\ ps\}}\mathit{While}$$

$$\frac{\begin{array}{c}\forall ps\ n.\ P'(ps,n) \Longrightarrow P(ps,n) \\ \vdash_3 \{P\}\ c\ \{Q\} \\ \forall ps\ n.\ Q(ps,n) \Longrightarrow Q'(ps,n)\end{array}}{\vdash_3 \{P'\}\ c\ \{Q'\}}\mathit{conseq}$$

Figure 2.4: Hoare logic with Separation Logic for reasoning about execution time

Soundness Proving soundness and completeness follows the same lines as for the quantitative Hoare logic, only complicated by the reasoning about partial states.

Theorem 17 (Soundness of $\vdash_3$)**.** $\vdash_3 \{P\}\ c\ \{Q\} \implies \models_3 \{P\}\ c\ \{Q\}$

Completeness This logic's weakest precondition is again defined as the right-hand side of the implication in the definition of validity:

$$wp\ c\ Q\ (ps,\ n) = \exists ps'\ n'\ t.\ (c,\ s) \overset{t}{\Rightarrow}_p\ ps' \wedge\ n = n' + t \wedge\ Q\ (ps',\ n')$$

For completeness we first show $\vdash_3 \{wp\ c\ Q\}\ c\ \{Q\}$ by induction on the command c, and then use the definition of validity and *wp* to finish the proof.

Theorem 18 (Completeness of $\vdash_3$)**.** $\models_3 \{P\}\ c\ \{Q\} \implies \vdash_3 \{P\}\ c\ \{Q\}$

Big-O style Similar to last subsection's system we extend the Hoare logic based on Separation Logic to big-O style reasoning. We again generalize our notion of validity (now $\models_{3'}$) and add a similar *const* rule to obtain the Hoare logic $\vdash_{3'}$. Proving soundness and completeness of this new Hoare logic follows the same lines as in the subsection before. Similarly, we come up with a simple VCG: somewhat unorthodoxly for Separation Logic, we use a backwards style, as well as we do not provide annotations for abstraction from multiplicative constants, as one final abstraction at the outer most position suffices to ensure completeness.

The approach inspired by Nielson to incorporate abstraction from multiplicative constants directly into the Hoare logic in order to reason about the order of magnitude of the running time of programs shows weaknesses and seems to complicate matters. Our theoretical results show that it is always possible to reason about the exact running time and abstract away multiplicative constants in a last step.

2.5 Interrelations

In this section, we discuss the interrelations between the Hoare logics described in the last section.

First we can compare the expressivity of the logics. Nielson logic $\models_1$ and the quantitative Hoare logic $\models_{2'}$, both big-O style logics, are equivalent in the following sense:

Lemma 19. $\models_1 \{\lfloor\Phi\rfloor_{\mathbb{B}}\}\ c\ \{\lambda s.\ \lfloor\Phi\ s - Q(\downarrow_S(c,s))\rfloor_{\mathbb{N}} \Downarrow \lfloor\Psi\rfloor_{\mathbb{B}}\} \implies \models_{2'} \{\Phi\}\ c\ \{\Psi\}$ *where* $\lfloor\Phi\rfloor_{\mathbb{B}}\ s = \Phi\ s < \infty$ *and* $\lfloor . \rfloor_{\mathbb{N}}$ *is the coercion from* $\mathbb{N}_\infty$ *to* $\mathbb{N}$*, assuming the argument is finite.*

Validity of a triple in the quantitative Hoare logic can be reduced to validity of a transformed triple in Nielson's logic. In the other direction this is only possible for assertions P and Q that do not depend on the state of their logical variables:

Lemma 20. $\models_{2'} \{\Uparrow P + e\}\ c\ \{\Uparrow Q\} \implies \models_1 \{P\}\ c\ \{e \Downarrow Q\}$ *where* $\Uparrow P\ s = (\forall l. \uparrow P\ l\ s)$

The quantitative logics support amortized resource analysis. On the face of it, Nielson's logic does not, but Lemma 19 tells us that in theory it actually does. However, automatic tools for resource analysis are mainly based on the potential method, for example [61, 17].

Furthermore, as the third system based on Separation Logic talks about partial states, in general it cannot be simulated by any of the other systems. This can only be done for assertions that act on complete states:

Lemma 21. $\models_{2'} \{\lfloor P \rfloor\}\ c\ \{\lfloor Q \rfloor\} \Longrightarrow \models_{3'} \{P\}\ c\ \{Q\}$, *when P is only true for complete partial states, with $\lfloor P \rfloor s = \inf_{n \in \mathbb{N}} \{P(\lfloor s \rfloor, n)\}$ and $\lfloor s \rfloor$ is the partial state defined everywhere and returning the same results as the total state s.*

On the other hand any triple in the quantitative Hoare logic $\models_{2'}$ can be embedded into the Separation Logic $\models_{3'}$:

Lemma 22. $\models_{3'} \{\lfloor P \rfloor\}\ c\ \{\lfloor Q \rfloor\} \Longrightarrow \models_{2'} \{P\}\ c\ \{Q\}$, *where $\lfloor P \rfloor(ps, n) = (\forall s.\ n \geq P\lfloor ps \rfloor^s)$ and $\lfloor ps \rfloor^s$ is the extension of the partial state ps by the state s to a total state.*

2.5.1 Example

Let c be the IMP program that computes the discrete square root by bisection:

```
l ::= 0 ;; r::= x + 1;;  m ::= 0 ;;
(WHILE l + 1 < r DO
   m ::= (l + r) / 2;;
   (IF m * m < x THEN l ::= m ELSE r ::= m);;
   m ::= 0)
```

With the simplification that the intervals between l and r are always powers of two, we can easily show the running time to be in the order of magnitude of $1 + \log x$. Note that we can get rid of multiplicative constants, but not additive ones!

For showing $\vdash_1 \{\lambda l\ s.\ (\exists k.1 + s\ ''x'' = 2^k)\}\ c\ \{\lambda s. \log(s\ ''x'') + 1 \Downarrow \lambda l\ s.\ True\}$ we provide the following annotations for the while loop: $I_1 = \lambda l\ s.\ s\ ''l'' \geq 0 \wedge (\exists k. s\ ''r'' - s\ ''l'' = 2^k)$, $E_1 = \lambda s.\ 1 + 5 \cdot \log(s\ ''r'' - s\ ''l'')$and $S_1 = \lambda s.\ s$; then we use our optimized VCG and prove the remaining proof obligations.

For showing $\vdash_{2'} \{(\lambda s.\ \uparrow (\exists k.1 + s\ ''x'' = 2^k) + (\log(s\ ''x'') + 1)\}\ c\ \{\lambda\_.0)$, we annotate the while loop with the potential $I_{2'} = \lambda s.\ \uparrow (s\ ''l'' \geq 0 \wedge (\exists k. s\ ''r'' - s\ ''l'' = 2^k)) + 5 \cdot \log(s\ ''r'' - s\ ''l'')$.

Let us now compare the VCGs. Our VCG for Nielson's logic requires the annotation of loops with invariants I, running time bounds E and the state transformers S. In contrast, the annotations required for the VCG for the quantitative Hoare logic are uniformly potentials. In the above example, one can see that this annotated potential $I_{2'}$ exactly contains the same information as both I_1 and E_1 in the Nielson approach. The additional 1+ in E_1 is needed, as E_1 describes the running time of the whole loop, where $I_{2'}$ describes the running time from after evaluating the loop guard. Only more practical experience can tell if it is better to work with separate I, E and S or with a combined invariant potential.

In addition our annotated commands for Nielson's system may require annotations of the form *Conseq* $\{P', Q, e'\}$, whereas for the quantitative Hoare logic we managed to reduce this to *Const* $\{k\}$ annotations. It would be desirable to reduce the *Conseq* annotations similarly.

2.6 Related Work

Hanne Riis Nielson [107, 106] was the first to study Hoare logics for running time analysis of programs. She proved soundness and completeness of her systems (on paper) which are based on a deep embedding of her assertion language. We base our formalization on the system given in [108] where assertions are just predicates, i. e. functions. However, our inference system differs from hers in several respects and our mechanized proofs in Isabelle/HOL are completely independent. Moreover we provide a VCG and prove it sound and complete.

Possibly the first example of a resource analysis logic based on potentials is due to Hofmann and Jost [60]. The idea of generalizing predicates to potentials in order to form a "quantitative Hoare logic" we borrowed from [16]: Carbonneaux *et al.* design a quantitative logic in order to reason about stack-space usage of C programs. They also formally show soundness of their logic in Coq. They employ their logic for reasoning about other resource bounds and use it as the underlying logic for an automatic tool for resource bound analysis [18, 17]. In his dissertation [15] Carbonneaux complements his tool-focused work with a theoretical treatment of an "Invariant Logic" with a completeness result w. r. t. to a small step semantics. The relation to our logics of Section 2.3 should be studied in more detail.

Atkey [2] proposed to use Separation Logic with Time Credits to reason about the amortized running time of programs; he formalized his logic and its soundness in Coq. Similar ideas were used by Hoffmann *et al.* [59] to prove lock-freedom of concurrent programs, and by Charguéraud and Pottier [21] to verify the amortized running time of the Union-Find data structure in Coq. Guéneau, Charguéraud and Pottier [41] extended their framework to also obtain O results for the running time of programs and apply it to a state-of-the-art algorithm [42, 40]. None of these works include verified VCGs. In the Part II and III of this thesis I will describe our implementations [138, 45, 44] of that technique in Isabelle/HOL.

There is also related work that extends to probabilistic programs. Kaminski *et al.* [66] reason about the expected running time of probabilistic programs and show that their approach corresponds to Nielson's logic when restricted to deterministic algorithms. Ngo *et al.* [105] extend the idea of working with potentials to reasoning about the expected running time of probabilistic programs. I will take up that work and present a formalization in Isabelle/HOL in Chapter 3. Wang et al. [129] present the first system for automated amortized resource analysis of *probabilistic functional* programs.

For formal treatment of program logics [110] is a good entry point. Basic concepts as well as formalizations of Hoare logics that lay the ground for our work can be found in [114].

2.7 Discussion

In this chapter we have studied three Hoare logics for reasoning about the running time of programs in a simple imperative language. We have formalized and verified their meta theory in Isabelle/HOL.

To focus on the time aspect we considered the simplistic imperative programming language IMP that, in particular, is deterministic and does not allow for (recursive) procedures. Extending the language for those concepts (following [110]) and studying time bounds in that context should be easy. The formalizations of Carbonneaux's Invariant Logics in Coq [15] does already cover procedures. Extending the language with probabilistic choice (following [118, 66]) is much more interesting and is covered in Chapter 3.

Studying other resources (like stack-space consumption [15]) would be interesting future work. Especially, in the light of Section 4.3 it would be interesting to revisit the work of this chapter and generalize it to arbitrary resource bounds. Initial experiments for reasoning about *lower bounds* on the running time for the quantitative Hoare logic were very promising. Studying that also for the other logics would be interesting. The concept of time receipts [103] or similarly negative integral time credits [40] could be used in the context with Separation Logic.

Baking-in the big-O style seems to complicate the theory and does not simplify the reasoning. With the structure of our completeness proofs we give a formal justification for the intuition that one can always reason with concrete constants and do a final asymptotic analysis at the end. When forming a practical framework in Chapter 5 we will follow that design principle. While Guéneau [40] develops techniques and Coq tactics to push the asymptotic reasoning into the algorithm analysis, we explore a different technique to avoid handling concrete constants and improve structuring our formalizations: stepwise refinement and *resource currencies* (Part III).

The quantitative Hoare logic seems to be a suitable foundation for automatic program analysis that is "bottom-up" in the sense that one starts with some concrete implementation, and asks for an estimate of the running time. A VCG similar to ours can be used to extract constraints on time bounds and invariants. Assuming a certain shape of those expressions, they can then be automatically solved with the help of off-the-shelve LP solvers (cf. [17]). Separation Logic with Time Credits allows for local reasoning and seems to be the more suitable logic for verifying the algorithm analysis in an interactive setting [21, 138, 40, 44]. One could integrate the automatic tools to assist the manual synthesis of time bounds, following our finding that the formalisms can be turned into one another. Those tools need not be formally verified, but — like for termination [75] — their results could be simply reconstructed and certified in the proof assistant.

2.8 Summary

- We presented three Hoare logics for reasoning about the running time of imperative programs. Unsurprisingly, they all three have basically the same expressivity.
- The "zen of the potential method" and the idea to use differences of potentials gives rise to a compositional Hoare logic that is considerably easier to use as the chronologically earlier Logic by Nielson [108]. The logic seems to be well suitable for automatic tools.
- The logic obtained by extending Separation Logic with Time Credits additionally allows for local reasoning and thus seems to be well suited for interactive theorem proving.
- Baking-in the big-O style seems to complicate the theory and does not simplify the reasoning. With the structure of our completeness proofs we give a formal justification for the intuition that one can always reason with concrete constants and do a final asymptotic analysis at the end.

3 Expected Running Time of Probabilistic Programs

In this chapter I take a detour from the main topic of this thesis and consider how we can apply the Hoare logics we reviewed in the last chapter to formally reason about the running time of probabilistic programs.

Kaminski et al. [66] present an approach to establish bounds on the expected running time of *probabilistic guarded command language* (pGCL) programs. They provide an operational semantics in terms of *Markov Decision Processes* (MDPs) and prove the equivalence with their semantics. Hölzl [62] uses the well-developed theory of MDPs in Isabelle/HOL [63] to formalize that operational semantics, the expected running time calculus and the equivalence proof. Hölzl also formalizes two well-known examples using this approach working directly on the MDP semantics: a simple random walk and the Coupon Collector. I will summarize that development in Section 3.1.

For a program c the *expected running time transformer* $ert\ c\ f\ \sigma$ gives the expected running time of c started in initial state σ under the assumption that f captures the running time of the computation following c. This has some very apparent similarity to the weakest prepotential *wp* from Section 2.3. Ngo, Carbonneaux and Hoffmann [105] first unaware of Kaminski et al.'s work [66] extend their quantitative Hoare logic to the expected running time of probabilistic programs by being so bold to simply add probabilistic choice to the calculus. It was surprisingly easy to extend our formalization of the quantitative Hoare logic to probabilistic programs. Instead of IMP I use pGCL, essentially add one more inference rule, prove soundness and completeness of the resulting calculus, and add a VCG. In contrast to Ngo et al., I do not consider procedures, nor do I try to automatically solve the verification conditions. Instead, I add a completeness result for the logic and the VCG. I describe that development in Section 3.2.

Interestingly, Kaminski et al. [66] build a connection to Nielson's Hoare logic (cf. Section 2.2) and show that their notion of *ert* is a conservative extension when restricting pGCL to deterministic programs. The fact that they also use Nielson's Logic as a reference point for Hoare logics for time bounds is reassuring that we chose the relevant logics from the literature in our meta study in Chapter 2.

While there is a well-developed theory for the *ert* calculus, reasoning about the expected running time of programs is still a laborious task. Batz et al. [7] provide a novel proof rule for a special kind of loops. It is for instance applicable for the inner loop of the Coupon Collector example. I formalize that rule, verify a more succinct proof, and a generalization of it, and apply it to the Coupon Collector problem in Section 3.3.

Because most material of this chapter is not novel but rather a combination of known results pulled together into a formalization in Isabelle/HOL, it was not published anywhere else yet. However, as there are many connections to the material of Chapter 2, I think it fits well into this thesis. Also, this summary might be useful as a basis for further verified quantitative analysis of *probabilistic* programs.

The formalization described in this chapter can be found online [49].

3.1 pGCL and the Expected Running Time Transformer

This section gives a succinct summary of the relevant parts of the expected running time calculus by Kaminski [66] and its formalization in Isabelle/HOL by Hölzl [62].

The probabilistic guarded command language (pGCL) can be seen as an extension to IMP (Section 2.1) adding two new concepts: First, assignments may involve probabilistic choice, i. e. the effect of an assignment is a probability distribution over post states. We use Isabelle's *probability mass functions* (*pmf*), which are essentially discrete probability distributions with finite support. Second, we allow nondeterministic choice between two branches, i. e. both branches are executed and then one of the results is chosen. While the theory is more general, for a streamlined presentation we fix the type of program states to be functions from variables to values (as we do in Chapter 2). Formally, we define the type *pgcl* for the deep embedding of pGCL:

$$\begin{aligned} \mathit{pgcl} = {} & \mathit{Skip} \mid \mathit{Assign}\ \mathit{vname}\ (\mathit{state} \to \mathit{val}\ \mathit{pmf}) \mid \mathit{Seq}\ \mathit{pgcl}\ \mathit{pgcl} \mid \mathit{Par}\ \mathit{pgcl}\ \mathit{pgcl} \\ & \mid \mathit{If}\ (\mathit{state} \to \mathit{bool})\ \mathit{pgcl}\ \mathit{pgcl} \mid \mathit{While}\ (\mathit{state} \to \mathit{bool})\ \mathit{pgcl} \end{aligned}$$

An assignment *Assign x u* changes the variable x to a value following the probability distribution u incurring cost *1*. If u is a Dirac distribution, i. e. assigning probability *1* to exactly one value, the assignment is *deterministic*. The command *Par* c_1 c_2 branches nondeterministically between c_1 and c_2. The command *Skip* has no effect and incurs cost *1*; command *Seq* is sequential composition. Commands *If* and *While* are straightforward and incur cost *1* for evaluating the guards.

Example 3.1.1. We formalize the simple one-dimensional random walk in pGCL: we flip a coin and depending on the result make a step to the left or right.

$$\begin{aligned} & \mathit{random\_walk} = x ::= \mathtt{return}_{pmf}\ \mathit{10};; \\ & \quad \mathit{WHILE}\ \mathit{0} < x\ \mathit{DO} \\ & \quad\quad x ::= \mathit{map}_{pmf}\ (\lambda \mathit{True} \Rightarrow x + \mathit{1} \mid \mathit{False} \Rightarrow x - \mathit{1})\ (\mathit{bernoulli}_{pmf}\ \mathit{0.5}) \end{aligned}$$

Note that $\mathtt{return}_{pmf}\ x$ is the Dirac distribution that assigns element x probability *1*, and $\mathit{map}_{pmf}\ f$ applies the function f to each value of a probability distribution. Variable x represents the current position and is initialized to *10*. While we stay in the positive half of the integers we flip a fair coin ($p = \mathit{0.5}$) and make a step to the left or to the right accordingly. If the program reaches zero it terminates. Probabilistic

branching (with probability p) can be encoded with a probabilistic assignment (coin flip with probability p) followed by a Boolean branching.[1]

Surprisingly, but well-known, the random walk reaches each point on the non-negative part with probability *1*. Also surprising, while it terminates almost surely (with probability *1*) the expected running time to get from *10* to *0* (termination) is infinite.

Example 3.1.2. As a second example we consider the Coupon Collector Problem. The idea is to compute the expected time until we collect N different coupons from a uniform, independent and infinite source of coupons.

$$
\begin{aligned}
&geom_{N,c} = WHILE\ b = 0\ DO \\
&\qquad\qquad b := map_{pmf}\ (\lambda True \Rightarrow 0 \mid False \Rightarrow 1)\ (bernouilli_{pmf}\ (c\ /\ N))
\end{aligned}
$$

$$
\begin{aligned}
&CC_N = c ::= \mathtt{return}_{pmf}\ 0;; \\
&\qquad b ::= \mathtt{return}_{pmf}\ 0;; \\
&\qquad\quad WHILE\ c < N\ DO \\
&\qquad\qquad geom_{N,c}\ ;; \\
&\qquad\qquad b ::= \mathtt{return}_{pmf}\ 0;; \\
&\qquad\qquad c ::= \mathtt{return}_{pmf}\ (c+1)
\end{aligned}
$$

While we still search for a coupon, we draw coupons until we find one we do not already have, then register the trove and restart the quest for the rest of the coupons. The expected running time of the inner loop has a geometric distribution, the outer loop will have expected running time $\Theta(N \log N)$. Note that this example is a simplification of the Coupon Collector Problem. Hölzl proves the equivalence of this version to the original version in [63].

The denotational semantics for the running time is given as an *expectation transformer* that is very similar to the weakest prepotential *wp* of the quantitative Hoare logic (cf. Lemma 9). For the probabilistic choice, we have to take the expectation over the distribution. For the demonic nondeterministic choice, we use the supremum of the two branches: here it is the bigger expected running time, as we look for the tightest *upper* bounds. The *While* rule is phrased with a least fixed point, instead of its unfolded version.

$$
\begin{aligned}
&ert :: pgcl \to (state \to ennreal) \to (state \to ennreal) \\
&ert\ Skip\ f = \mathbf{1} + f \\
&ert\ (Assign\ v\ u)\ f = \mathbf{1} + (\lambda s.\ \textstyle\int_x f\,(s(v := x))\mathsf{d}(u\,s)) \\
&ert\ (Seq\ c_1\ c_2)\ f = ert\ c_1\ (ert\ c_2\ f) \\
&ert\ (Par\ c_1\ c_2)\ f = ert\ c_1\ f \sqcup ert\_c_2\ f \\
&ert\ (If\ g\ c_1\ c_2)\ f = \mathbf{1} + (\lambda x.\ \mathtt{if}\ g\ x\ \mathtt{then}\ ert\ c_1\ f\ \mathtt{else}\ ert\_c_2\ f) \\
&ert\ (While\ g\ c)\ f = lfp\ (\lambda W\ x.\ 1 + (\mathtt{if}\ g\ x\ \mathtt{then}\ ert\ c\ W\ x\ \mathtt{else}\ f\ x))
\end{aligned}
$$

The expectation transformer $ert\ c$ maps a postexpectation to a preexpectation. An expectation of type $state \to ennreal$ maps a state to an extended non-negative real

[1] In contrast to modeling probabilistic branching directly — like Batz et al. [7] do — our encoding of probabilistic branching incurs cost *2*.

denoting the expected running time (ERT) for that starting state. Note that the ERT can also be infinite. The notion of *expecations* is due to McIver and Morgan [102]. Here, the term ***1*** is the expectation ($\lambda s.\ 1$); the operators $\sqcup$ and $+$ are lifted from *ennreal* to expectations pointwise.

In particular, the rule for sequential composition stays a simple recursive definition. By setting $f = 0$ we obtain the expected running time *ert c 0* for a program c.

We can now prove some validating theorems about the expectation transformer *ert*. For example we obtain that *ert* is monotone w. r. t. continuations:

Theorem 23. $f \leq g \implies ert\ c\ f \leq ert\ c\ g$

Following that, Hölzl [62] defines the operational small-step semantics by introducing a MDP constructed from a pGCL program. This is formalized by the MDP's transition function K and a cost function that collects the costs along a trace. The transition function $K :: (pgcl \times state) \to (pgcl \times state)\ pmf\ set$ maps a *configuration*, i. e. a pair of a program and a state, to a set of distributions of configurations, We write *Ert f c s* for the maximal expectation of the sum of *cost* over all states in all possible traces for program c started in state s with continuation $f :: state \to ennreal$. Intuitively, it is the expected cost of the MDP for program c started from state s with continuation f. This expected cost is provably equivalent to the expected running time obtained from the *ert* calculus:

Theorem 24. *(Theorem 1 in [62], Theorem 2 in [66], Theorem 6.1 in [105])*

$Ert\ f\ c\ s = ert\ c\ f\ s$

Once the *ert* calculus is backed by a formal semantics, we can turn to the case studies and determine their expected running times.

Example 3.1.3. For the simple random walk Hölzl uses the *ert* calculus to obtain an expression for the expected running time that involves a fixed point. Then he uses traditional methods to prove the result[2]: $ert\ random\_walk\ 0\ s = \infty$.

Example 3.1.4. If there are already c coupons collected, the inner loop has expected running time $3 + 2 * (N\ /\ (N - c))$. It increases with c but always stays finite. The overall running time then is:

$$2 + N * 4 + 2 * N * (\sum_{i<N-1} 1/(i+1))$$

The claims are proved by "fixed point transformations"[3]. We will later see a more automatic approach.

[2]The theory is around 300 LOC.

[3]The theory has 240 LOC, with around 40 concerning the inner loop. It also contains the refinement proof of the two equivalent formulations of the Coupon Collector (cf. Figure 4 in [62]).

$$\frac{\vdash_P \{\Phi\}\ c_1\ \{\Psi\} \qquad \vdash_P \{\Phi\}\ c_2\ \{\Psi\}}{\vdash_P \{\Phi\}\ Par\ c_1\ c_2\ \{\Psi\}}\,Par \qquad \frac{\Phi\, s \geq 1 + \int_x \Psi\,(s(v := x))\mathsf{d}(u\, s)}{\vdash_P \{\Phi\}\ v ::= u\ \{\Psi\}}\,Assign$$

Figure 3.1: New inference rules for the probabilistic quantitative Hoare logic

3.2 Probabilistic Quantitative Hoare Logic

We now extend the potential method from the quantitative Hoare logic from Section 2.3 to the *expected potential method*. Like in the deterministic case we work with triples $\{\Phi\}\ c\ \{\Psi\}$ now with potentials $\Phi, \Psi :: state \to ennreal$ to ensure compositional reasoning.

We follow the intuition that the expected running time transformer *ert* generalizes the weakest prepotential *wp* from the deterministic quantitative Hoare logic. There Lemma 8 states that a triple is valid, if the weakest prepotential of a command c of a postpotential is at most the given prepotential Φ: $wp\ c\ \Psi \leq \Phi$.

We define validity of triples in the probabilistic quantitative Hoare logic in a similar way:

$$\models_P \{\Phi\}\ c\ \{\Psi\} \longleftrightarrow \Phi \geq ert\ c\ \Psi$$

A triple is valid, if the prepotential can pay for the expected running time of c plus the continuation Ψ. The continuation is evaluated in the state at the end of the computation c and thus can be seen as the *postpotential*. Note that our logic is designed to obtain upper bounds on the expected running time of probabilistic programs.

With the *ert* rule for sequential composition in mind we immediately see that the telescoping argument works also here: if we know that $\Phi \geq ert\ c_1\ \Psi$ and $\Psi \geq ert\ c_2\ \Pi$ we can conclude with monotonicity of *ert* that $\Phi \geq ert\ c_1\ (ert\ c_2\ \Pi) = ert\ (c_1\ ;;\ c_2)\ \Pi$. Thus the sequential composition rule for triples holds (rule *Seq* in Figure 2.3).

Surprisingly, we can reuse most of the rules from the deterministic quantitative Hoare logic (cf. Figure 2.3): we only have to add a rule for nondeterministic choice (*Par*) and modify the assignment rule that now involves probabilistic choice (cf. Figure 3.1). For nondeterministic choice each branch has to adhere to the pre and postpotential. For the assignment rule, for any state s the prepotential has to pay *1* for the assignment and the expected postpotential for the updated state over the probability distribution $u\ s$. If u is a Dirac distribution, the assignment rule simplifies into the deterministic assignment rule.

When proving soundness of the inference rules, most of the proofs are straightforward. In the deterministic case, the proof for loops works by an induction on the value of the potential, which decreases with the computation. In the probabilistic case, there is nothing that decreases, only a recurrence relation that needs to be fulfilled. There, reasoning over fixed points solves the problem.[4] Then we obtain soundness of the calculus. For completeness we first prove $\vdash_P \{ert\ c\ \Psi\}\ c\ \{\Psi\}$, which is basically automatic

[4]The proof in the appendix C.2 of [105] provided the right pointers to complete the formal proof.

and only needs manual proof for the loop construct. Completeness is a corollary from that, and we finally obtain:

Theorem 25. *(Soundness and completeness of $\vdash_P$)*

$$\vdash_P \{\Phi\}\ c\ \{\Psi\} \longleftrightarrow \models_P \{\Phi\}\ c\ \{\Psi\}$$

If we restrict ourselves to pGCL programs without nondeterministic choice and probabilistic assignments, we obtain a language isomorphic to IMP. For those programs, the operational semantics on MDPs collapses into a deterministic transition system with a reward function.[5] The set of rules of the probabilistic quantitative Hoare logic then is a conservative extension of the rules of the quantitative Hoare logic. From the discussion at the end of Chapter 2 we know that the quantitative Hoare logic is as expressive as Nielson's logic. Together this implies that the Hoare logic of this chapter is a conservative extension of Nielson's Logic. Which is also a result of Kaminski et al. [66].[6]

With the setup from Chapter 2 it is straightforward to come up with a VCG. Only loops need to be annotated with an invariant potential, the definitions for *vc* and *pre* can be taken from the deterministic quantitative Hoare logic with minor modifications. Then we can also prove that the VCG is sound and complete.

Example 3.2.1. Consider a skewed random walk $rdwalk_2$, where we go away from zero with probability *0.25* and toward zero with probability *0.75*. The expected running time of that random walk is actually finite.

We can use the verification condition generator on an annotated version of $rdwalk_2$, where we leave the invariant as a free variable *I*. Then the verification conditions collect the inequalities on *I*. In our case it is only one:

$$I\ (s\ x) \geq 1 + 1 + ((I\ (s\ x + 1)) * (1 / 4) + (I\ (s\ x - 1)) * (3 / 4))$$

The inequality stems from the "invariant preservation" through the loop body. One time unit is spent for evaluating the loop guard, and the other one is spent on the coin flip in the assignment. Then follow the weighted "recursive cases".

Solving such conditions (e. g. with $I\ s = 4 * s + 1$) can be delegated to automatic tools. When we plug in a valid solution and prove the verification conditions, we obtain the desired triple $(s\ x \geq 0 \implies \{\lambda s.\ 4 * s\ x + 1\}\ rdwalk_2\ \{0\})$ which entails a bound on the expected running time: $s\ x \geq 0 \implies ert\ rd\_walk\ 0\ s \leq 4 * s\ x + 1$.

Outsourcing the solving of those constraints is actually what Ngo et al. [105] do: similar to their earlier work on deterministic programs [18, 17] they collect the conditions on the potentials used in the program and then fix the shape of the potential functions. Their potential functions are linear combinations of elementary base potential functions. Finding the coefficients in the potential functions can be reduced to off-the-shelf LP solving. The LP solver need not be verified, as plugging its solution

[5] It would be interesting to obtain an operational small step semantics of IMP by establishing this connection formally.

[6] We leave the formalization of that idea as future work (`https://github.com/maxhaslbeck/verERT/issues/2`).

into the program and then proving the verification conditions inside the proof assistant verifies the result.

Example 3.2.2. Another example is a Geometric series:

$$\begin{aligned} &geo = WHILE\ x = 0\ DO \\ &\qquad x ::= map_{pmf}\ (\lambda b.\ \texttt{if}\ b\ \texttt{then}\ 1\ \texttt{else}\ 0)\ (bernoulli_{pmf}\ 0.5) \end{aligned}$$

From this we can extract the following recurrence relation:

$$P\ 0 \geq 2 + (P\ 1)\ /\ 2 + (P\ 0)\ /\ 2$$

Which can be solved with $I\ s = (\texttt{if}\ s\ x = 0\ \texttt{then}\ 5\ \texttt{else}\ 1)$. Note that *geo* is a special case of the inner loop of Example 3.1.2 with $c/N = 0.5$. We will see in the next section that we also can automatically find its expected running time without solving for the invariant by hand.

3.3 A Proof Rule for f-i.i.d. Loops

We have seen that we can reduce finding the ERT of a probabilistic program to fining solutions of systems of inequalities on potentials. While the extraction of those inequalities can be conveniently automated, solving those constraints is still hard. The problem is obviously the looping constructs, which result in recurrence relations. For certain loops that rule out undesired parts of the data flow across loop iterations Batz et al. [7] provide a proof rule that does not involve finding an invariant. They call this class of probabilistic loops *f-i.i.d. loops*.

Batz et al. identify a syntactic fragment of pGCL that only allows such loops and provide a formal translation from Bayesian networks (BNs) with observations into that fragment. The ERT of a program obtained from a BN by that translation corresponds to the expected sampling time of that BN. Using the new proof rule allows automatic determination of that expected sampling time.

In the following we will introduce the concept of *f*-i.i.d. loops, the new proof rule with a generalization and apply it to the inner loop of the Coupon Collector example.

Weakest preexpectation Before we define the class of *f*-i.i.d. loops, we need to introduce the *weakest preexpectation transformer wp* for reasoning about expected outcomes of executing probabilistic programs. That notion is originally due to Kozen [73]. McIver and Morgan [102] added nondeterministic choice and coined the terminology in use. We have already seen expectations, they are essentially functions that map states to extended non-negative reals *ennreals*, or $\mathbb{R}^{\infty}_{\geq 0}$.

The weakest preexpectation transformer *wp* of a program c maps a postexpectation f to a preexpectation $wp\ c\ f$, such that $wp\ c\ f\ \sigma$ is the expected value of f after executing c on initial state σ. It is defined recursively on all pGCL programs:

$$\begin{aligned} &wp :: pgcl \to (state \to ennreal) \to (state \to ennreal) \\ &wp\ Skip\ f = f \\ &wp\ (Assign\ v\ u)\ f = (\lambda s.\ \textstyle\int_x f\ (s(v := x))\mathsf{d}(u\ s)) \end{aligned}$$

$$wp\ (Seq\ c_1\ c_2)\ f = wp\ c_1\ (wp\ c_2\ f)$$
$$wp\ (Par\ c_1\ c_2)\ f = wp\ c_1\ f \sqcap wp\_c_2\ f$$
$$wp\ (If\ g\ c_1\ c_2)\ f = (\lambda x.\ \texttt{if}\ g\ x\ \texttt{then}\ ert\ c_1\ f\ \texttt{else}\ ert\_c_2\ f)$$
$$wp\ (While\ g\ c)\ f = lfp\ (\lambda W\ x.\ (\texttt{if}\ g\ x\ \texttt{then}\ ert\ c\ W\ x\ \texttt{else}\ f\ x))$$

The infimum operator $\sqcap$ is lifted from ennreal to expectations pointwise. It is used in the nondeterministic choice to model demonic nondeterminism. This results in lower bounds on the expectation: in case of nondeterministic choice we take the smaller expectation of the two branches. Choosing the supremum operator instead leads to upper bounds on the expectation, analogously we define the *angelic weakest preexpectation transformer* *awp* with only the rule for *Par* modified to:

$$awp\ (Par\ c_1\ c_2)\ f = awp\ c_1\ f \sqcup awp\_c_2\ f$$

The other rules are straightforward, e. g. *Skip* has no effect on the state and thus no effect on the postexpectation. The only difference to *ert* is that we do not add the cost *1*. Similarly, in contrast to *ert* we add no costs in the assignment, branching and looping construct.

We can also define *aert* using the infimum instead of the supremum for the *Par* construct and obtain lower bounds on the expected running time in case of nondeterministic choice. We call programs that do not involve nondeterministic choice, i. e. do not contain the construct *Par*, *fully probabilistic*. For fully probabilistic pGCL programs *wp* and *awp* are equal, for general programs we have $wp\ c\ f \leq awp\ c\ f$.

Intuitively, both *wp* and *ert* pull an postexpectation through a program, while *ert* adds costs along the way. For *wp*, it does not matter whether one pulls through two postexpectations and adds them up afterwards, or one pulls through the sum in one go. Furthermore, from the zero postexpectation — mapping each state to *0* — one obtains the prepotential zero. The same is not true for *ert*: *ert c 0* is not necessarily *0*, rather it is the expected running time of *c*. Nor can we split up *ert c* $(f + g)$ into a sum of two *ert*s because the costs would be added up twice. But we can decompose a general *ert c f* with continuation *f* into the part that calculates the running time *ert c 0* and the part that pulls through the continuation *wp c f*. The former two properties are called *healtiness conditions* [102], the latter is called the *decomposition of ert*.

Lemma 26 (Healthiness condition of *wp* [102])**.** *For all pGCL programs c, expectations* f_1, f_2 *and* $a \in \mathbb{R}^{\infty}_{\geq 0}$ *the following holds:*

$$wp\ c\ (a * f_1 + f_2) = a * wp\ c\ f_1 + wp\ c\ f_2$$
$$wp\ c\ 0 = 0$$

The same rules also hold for *awp*. Also, note that *wp c 1 = 1* does not always hold and pGCL programs cannot crash. Consequently, the fact *wp c 1 = 1* expresses that *c* terminates with probability *1*.

Lemma 27 (Decomposition of *ert* (Lemma A.15 in [118]))**.** *For all fully probabilistic pGCL programs c, expectations f,*

$$ert\ c\ f = ert\ c\ 0 + wp\ c\ f$$

For notation, we introduce the *Iverson brackets*, which embed predicates into expectations: The expectation $[\varphi]$ is *1* if φ is True and *0* otherwise.[7] With that we can rephrase the *wp* rules for *If* and *While*:

$$wp\ (\mathit{If}\ g\ c_1\ c_2)\ f = [g] * wp\ c_1\ f + [\neg g] * wp\ c_2\ f$$
$$wp\ (\mathit{While}\ g\ c)\ f = \mathit{lfp}\ (\lambda W.\ [g] * \mathit{ert}\ c\ W + [\neg g] * f)$$

f-i.i.d. Loops I will now introduce a proof rule that allows to determine the ERTs of *independent and identically distributed loops* (*i.i.d. loops* for short). Such loops have a loop body that computes the same distribution of states for each loop iteration. The loop in Example 3.1.1 is not of that sort because the distribution of values of x changes over several loop iterations. In contrast, the inner loop $geom_{N,c}$ of the Coupon Collector Problem (Example 3.1.2) is i.i.d.: In every iteration the loop leaves all variables but b unchanged, and b is set to *0* or *1* with probability $p = c \,/\, N$ and $1 - p$ respectively. I will use that program as a running example in this section. The new proof rule will allow us to automatically determine the ERT of this loop, i. e. the average amount of time required until a new coupon is discovered.

Let us establish a syntactical notion of the i.i.d. property. It relies on the property of expectations to not be *affected* by a pGCL program. The term *Mod c* is defined to be the set of variables that occur on the left-hand side of an assignment in c. It is a simple over-approximation of the variables that are modified by c. Furthermore, the term *Vars f* denotes the variables an expectation f depends on. As expectations f are not deeply embedded but mere functions, we define *Vars* semantically. Now, f is unaffected by c if $\mathit{Vars}\ f \cap \mathit{Mod}\ c = \emptyset$, denoted by $f \not\Mid c$.

If $g \not\Mid c$, the expectation g can be pulled through the program c unaffected. That is, g can be *treated like a constant* w. r. t. the transformer *wp* and be pulled through it like the factor a in Lemma 26. For our running example, let us denote the loop body of $geom_{N,c}$ by *loopbody*. Then, the expectation $f = wp\ loopbody\ ([b = 0])$ is unaffected by the loop body of $geom_{N,c}$. Note that $[b = 0]$ is affected by *loopbody*, but f is not. Consequently, we have $wp\ loopbody\ f = f * wp\ loopbody\ 1 = f$. The last equality holds, as the loopbody always terminates. In general, we obtain the following property:

Lemma 28 (Scaling by unaffected expectations (Lemma 1 in [7]))**.** *For a fully probabilistic pGCL program c and expectations f, g:*

$$g \not\Mid c \implies wp\ c\ (g * f) = g * wp\ c\ f.$$

The intuitive notion of i.i.d. loops can be loosened to some extent. The loop body does not need to compute the exact same distribution of states. It only needs to leave the probability that the guard is true after one iteration of the loop ($wp\ c\ [\varphi]$) as well as the expected value of the postpotential f after one iteration in case we leave the loop ($wp\ c\ ([\neg\varphi] * f)$) are unaffected by the loop body. This is collected in the following definition.

[7] It is similar to the embedding from predicates to potentials, and we will see other instances of such embeddings in Chapter 4.

Definition 1 (*f*-independent and identically distributed loops (Definition 5 in [7]))**.** *For a pGCL program c, a predicate φ and an expectation f, we call the loop While φ c f-independent and identically distributed (f-i.i.d. for short), if:*

$$wp\ c\ [\varphi] \not\circledast c \wedge \quad wp\ c\ ([\neg\varphi]{*}f) \not\circledast c$$

Example 3.3.1. Our running example program $geom_{N,c}$ is *f*-i.i.d. for all expectations *f*. We have can easily prove,

$$wp\ loopbody\ [b = 0] = 0.5 \not\circledast loopbody$$
$$g := wp\ loopbody\ ([b \neq 0] * f) = 0.5 * f\ (s(b{:=}1)) \not\circledast loopbody$$

The first expectation is constant and thus unaffected. For the second expectation, the loop body only modifies variable *b* ($Mod\ loopbody = \{b\}$) but in the preexpectation *b* is overwritten with *1*, thus it is not changing *g* (i. e. $b \notin Vars\ g$).

Now we can state and prove the proof rule for the ERT of an *f*-i.i.d. loop *While φ c*.

Theorem 29 (Proof Rule for ERTs of f-i.i.d. Loops (Theorem 4 in [7]))**.** *Let c be a fully probabilistic pGCL program, φ a predicate and f an expectation such that the following conditions hold:*

1. *While φ c is f-i.i.d.*
2. *the loop body terminates almost-surely (wp c 1 = 1)*
3. *every iteration runs in the same expected time (ert c 0 $\not\circledast$ c)*

Then for the ERT of the loop While φ c w. r. t. continuation f it holds that

$$ert\ (While\ \varphi\ c)\ f = 1 + \frac{[\varphi]*(1+ert\ c\ ([\neg\varphi]*f))}{1-wp\ c\ [\varphi]} + [\neg\varphi] * f$$

Proof. Batz et al. give a quite technical proof involving orbits of the loops. Borrowing ideas from Hölzl's semantic proof of the inner loop of the Coupon Collector problem, I have found a more abstract proof.

The heart of the argument is that for the least fixed point of a linear expression on extended non-negative reals we can give a closed form using a geometric series:

$$b < 1 \implies lfp\ (\lambda r.\ a + b * r\ ) = a\ /\ (1 - b)$$

In the final expression we get the middle term with that technique. The first and last summand stem from the cost from the first evaluation of the guard and the continuation in case when the loop guard is false.

Massaging the least fixed point from the *ert* form into the form needed to apply the above method involves a parallel induction on the least fixed points and the decomposition of *ert* (Lemma 27). □

Example 3.3.2. We can now apply this proof rule to our running example. Let program *coco* initialize the state with $b = 0$ and $c = C$ and then execute $geom_{N,c}$. Then, we get the final result:

$$N > 1 \wedge C < N \implies \mathit{ert}\ \mathit{coco}\ 0\ s = 3 + (2 * N \,/\, (N - C))$$

Which is the same result as we found earlier (with $C \,/\, N = 0.5$, in Example 3.2.2), only with the cost *2* more for the two additional assignments in the beginning.

Wrap-up To conclude this chapter we want to comment on extensions we did and future work we envision. For the first iteration of the proof of Theorem 29 we followed the proofs by Batz et al. [7] quite closely, but could simplify the proof considerably by taking ideas of Hölzl's verification of the Coupon Collector.

Once the abstracted proof was established, we thought about extending it to also allow nondeterministic choice. We could prove a rules similar to Lemma 27 and Lemma 28 for *awp* with inequalities instead of the equalities. Finally we could give the same proof rule for *awp* as Theorem 29 only with an inequality instead of the equality and an extra added assumption.[8]

I think that proof assistants and specifically the probability theory in Isabelle/HOL are mature enough to verify state-of-the-art results in the area of analysis of probabilistic programs without too much pain. I want to point out that the proofs in the appendices of the papers on the *ert* calculus [66, 7] — and also on Quantitative Separation Logic [8] which we will see in the next chapter — are very detailed and I am impressed by the work getting the details right without the help of a proof assistant. Turning those informal proofs into formal Isabelle proofs was still laborious and can be considered boring *frog*[9] *work* distracting from more interesting *bird work*. But I argue that in this case the advantages of easing the review process and the proof assistant helping when abstracting and generalizing established proofs justify the extra work of verifying it.

We mentioned, that Batz et al. [7] use pGCL and the derived proof rule to automatically determine the sample rate for Bayesian networks. It would be interesting future work to formalize that formalism and the translation to pGCL.

3.4 Summary

- The *ert* calculus can be used to reason about the expected running time of pGCL programs. The probability theory in Isabelle/HOL is mature enough to verify that calculus and provide a proof for the equivalence w. r. t. an operational semantics in terms of Markov Decision Processes.

- The quantitative Hoare logic can be extended to probabilistic programs. I prove soundness and completeness of the logic as well as of its VCG.

- A proof rule for a special kind of loops can be used to automatically determining the expected running time of a fragment of pGCL programs. Reasoning with

[8] The extra assumption asserts that if a postexpectation is unaffected by the loop body, also its preexpectation for one loop unrolling is unaffected by the loop body.

[9] Term coined by Dyson [30].

fixed points simplifies the correctness proof of that rule and it can be extended to pGCL programs with nondeterministic choice.

4 Quantitative Separation Logic & Quantales

In Chapter 2 we have seen the quantitative Hoare logic and its drawback of not being modular. Separation Logic enables modular reasoning providing means to reason about disjoint parts of the heap and thus enabling the frame rule. If we set the concept of time credits aside for now, we can ask whether we can extend the quantitative Hoare logic by the concept of partial heaps and separating conjunction.

Batz et al. [8] answer that question positively. They present a Quantitative Separation Logic to reason about the expected running time of programs of the *heap-manipulating probabilistic guarded command language* (denoted hpGCL). This extends the probabilistic quantitative Hoare logic I have covered in Section 3.3.

To me, their most interesting contribution is the idea how to lift the separating conjunction and the magic wand from Boolean assertions ($\Sigma \to$ *bool*) to quantitative potentials ($\Sigma \to$ *ennreal*). After having formalized their Quantitative Separation Logic in Isabelle/HOL — following their paper and joint discussions — for expectations of type $\Sigma \to$ *ennreal*, I observed that the measuring type *ennreal* can be generalized. With the help of the proof assistant Isabelle/HOL it was straightforward to collect the properties such a type γ needs to fulfill. But it needed an expert in algebra to point out to me the structure in question being *quantales*. This discovery opened up the connection to a broad field of new related work to me. Note that the measuring types *bool*, *[0, 1]*, *ennreal* and *enat* with a suitable orderings and compose operations are instances of that structure and give rise to the standard Separation Logic, Separation Logic over probabilities, Separation Logic over expectations and Separation Logic over potentials.

In this chapter I will first introduce the quantitative separating connectives and the formalization thereof (Section 4.1). Then, I quickly summarize its application to the Quantitative Separation Logic (QSL) and its formalization (Section 4.2). Note that while Batz et al. [8] extend the weakest preexpectation calculus in their paper, I use a similar but different instance here that allows to extend the *ert* calculus from last section.[1] Finally, I collect literature and observations we found about the role of quantales in quantitative separating connectives, and conjecture some connections to other fields of program language semantics (Section 4.3).

[1] It is the instance Matheja describes as future work in his PhD thesis [100, Chapter 9].

4.1 Quantitative Separating Connectives

In this section I describe my formalization of material from Section 3.2 and 3.3 of "Quantitative Separation Logic" by Batz et al. [8]. While the ideas are the same, I apply it to a different domain.

We have already seen some introduction to Separation Logic in Section 2.4. In this Section I will apply the "zen of the potential method" to Separation Logic, by lifting predicates to potentials. As the running example let us consider potentials that map partial states (now denoted by M) to extended non-negative real (*ennreal*). We will later see, that this can be relaxed to functions that map elements from a separation algebra to elements of a quantale.

We have already seen how we can embed predicates into potentials. In this chapter, we will denote that embedding as $[P] = (\lambda h.\ \texttt{if}\ P\ h\ \texttt{then}\ 0\ \texttt{else}\ \infty)$. As ∞ corresponds to *False* when evaluating a predicate, we interpret *satisfying a potential* φ as measuring some finite quantity, i. e. $\varphi\ h < \infty$. *Standard conjunction* ($\wedge$) is modeled by pointwise addition. This is a conservative extension, as for any two predicates P and Q we have $[P \wedge Q] = [P] + [Q] = \lambda h.\ [P]\ h + [Q]\ h$. If for a partial state h one of the predicates is false the expression has value ∞; only if both are true the expression has value 0.

What is the intended meaning of the potential? In this section it should be the running time, or the expected running time potential of some portion of the state. If we join or "add up" two portions, the potential should be added.

The ordering on predicates is routinely defined as $P \leq Q \longleftrightarrow (P \implies Q)$, which stems from a pointwise lifting and the fact that *False* is the bottom element of the Boolean lattice. For potentials, we also lift the natural ordering on *ennreal* with ∞ as the top element. So, for showing backwards compatibility, we have to flip the ordering. Then we can prove: $[P] \geq [Q] \longleftrightarrow P \leq Q$. Only if P is true and Q is false, the right-hand side of the equivalence is false, but then also the left-hand-side is false (as $[P] = 0$ and $[Q] = \infty$). In the rest of this section, I will talk a lot about suprema and infima. When abstracting the theory to arbitrary types with an ordering, we have to keep in mind that we actually use *ennreal* with the flipped ordering, and need to flip the ordering, as well as suprema and infima.

Quantitative Separating Conjunction Let us recall the definition of *separating conjunction* in standard Separation Logic:

$$(P \star Q)\ h \longleftrightarrow \exists h_1\ h_2.\ h = h_1 + h_2 \wedge P\ h_1 \wedge Q\ h_2$$

In words, partial state h satisfies $P \star Q$ iff there exists a partition of h into two partial states h_1 and h_2 such that h_1 satisfies P and h_2 satisfies Q.

The key insight is how to generalize the existential quantifier. Intuitively, the quantified predicate $\exists x.\ P\ x$ "minimizes the falsehood of $P\ x$" with the best possible choice of x: if there is at least one instance that makes P true, the quantified predicate is true. The best possible potential, is the one furthest away from "False". In our case it is *ennreal*'s bottom element 0. So, the existential quantifier is generalized by an infimum

operator. In the Quantitative Separation Logic (QSL), instead of the falsehood, we minimize a quantity, i. e. the distance from the infinite potential: Out of all partitions $h = h_1 + h_2$ we choose the one that minimizes the sum $P\ h_1 + Q\ h_2$. Thus, we define the *quantitative separating conjunction* as follows:

Definition 2 (Quantitative Separating Conjunction)**.**

$$(\varphi \star \psi)\ h = \mathit{Inf}_{h_1,h_2}\ \{\ \varphi\ h_1 + \psi\ h_2 \mid h = h_1 + h_2\ \}$$

To check whether this definition makes sense, we observe that if we restrict potentials to embedded predicates we obtain the classical operator: For predicates P and Q, we have $([P] \star [Q])\ h \in \{0, \infty\}$ and moreover $([P] \star [Q])\ h = 1$ holds in QSL if and only if $(P \star Q)\ h = \mathit{True}$ in SL.

Quantitative Separating Implication For classic Separation Logic, the *separating implication* for predicates P and Q is defined as

$$(P \mathrel{-\!\!\star} Q)\ h \longleftrightarrow \forall h'.\ (h' \# h \wedge P\ h') \implies Q\ (h + h')$$

So h satisfies $P \mathrel{-\!\!\star} Q$ iff the following holds: Whenever we can find heap h' disjoint from h such that h' satisfies P, then $h + h'$ must satisfy Q. Put differently: we measure the truth of Q in *extended* heaps $h + h'$, where all admissible extensions h' must satisfy P.

To connect potentials φ and ψ in a similar fashion, intuitively, $\varphi \mathrel{-\!\!\star} \psi$ intends to measure ψ in extended heaps, subject to the fact that extensions "satisfy" φ (i. e. $\varphi\ h < \infty$).

As for the universal quantifier, the key insight is now to generalize it with a supremum: The quantified predicate $\forall x.\ P\ x$ "maximizes the falsehood of $P\ x$" by requiring only one false instance to make the whole quantified predicate false. In QSL we require the result nearest to the infinite potential, so this time we maximize a quantity: out of all heap extensions h' that "satisfy" a given expectation φ, we choose an extension that maximizes the quantity $\psi\ (h + h')$. We define the quantitative separating implication by a supremum:

Definition 3 (Quantitative Separating Implication)**.**

$$(\varphi \mathrel{-\!\!\star} \psi)\ h = \mathit{Sup}_{h'}\ \{\ \psi\ (h + h') - \varphi\ h' \mid h' \# h \wedge (\varphi\ h' < \infty \vee \psi\ (h + h') < \infty) \wedge (\varphi\ h' > 0 \vee \psi\ (h + h') > 0)\ \}$$

Here, the operator $-$ is just subtraction on extended non-negative real numbers. The constraints on φ and ψ are needed to avoid the corner case $\infty - \infty$, and the corner case $0 - 0$ can also be excluded as it matches the supremum of the empty set. If no suitable extension h' can be found, we take the supremum over the empty set, which results in the bottom element of *ennreal*: 0.

On the type *ennreal* we have the corner cases defined as expected: if $r \neq \infty$ we have $\infty - r = \infty$ and $r - \infty = 0$.

As a sanity check, we study whether the definition allows for backwards compatibility. First, we have $[P] \mathrel{-\!\!\star} [Q] \in \{0, \infty\}$. If we take the supremum of the empty set and

the result is *0*. Otherwise we remember that any embedded predicate can only have potential *0* or ∞. Then we find a h' with either $[P]\ h' = 0$ and $[Q]\ (h + h') = \infty$ (then its value is $\infty - 0 = \infty$) or $[P]\ h' = \infty$ and $[Q]\ (h + h') = 0$ (then its value is $0 - \infty = 0$).

Second, we also have $([P] \mathrel{-\!\!\star} [Q])\ h = 0 \longleftrightarrow (P \mathrel{-\!\!\star} Q)\ h = \mathit{True}$.

I omit the definition of quantitative versions of *septraction* and *separating coimplication* (for classical Separation Logic cf. Figure 1 in [6]), but believe that they can be defined similarly.

Properties of quantitative separating connectives Before we review the main properties of the quantitative separating connectives, let us define the neutral assertion, i. e. the empty partial state: $[emp]\ h = (\mathtt{if}\ dom\ h = \emptyset\ \mathtt{then}\ 0\ \mathtt{else}\ \infty)$. It will serve as the neutral assertion and together with the quantitative separating conjunction is a *commutative monoid*.

Theorem 30. *($partialheap \to ennreal$, $\star$, $[emp]$) is a* commutative monoid*, i. e. for all potentials φ, ψ, π the following holds:*

1. *Associativity:* $\varphi \star (\psi \star \pi) = (\varphi \star \psi) \star \pi$
2. *Neutrality of* $[emp]$*:* $\varphi \star [emp] = [emp] \star \varphi = \varphi$
3. *Commutativity:* $\varphi \star \psi = \psi \star \varphi$

Another important property that is preserved from the classic Separation Logic, is that $\star$ is monotonic, and $\mathrel{-\!\!\star}$ is anti-monotonic in the first component and monotonic in the second component:

Theorem 31 (Monotonicity of $\star$ and $\mathrel{-\!\!\star}$)**.**

1. $\varphi \leq \varphi'$ *and* $\psi \leq \psi'$ *implies* $\varphi \star \psi \leq \varphi' \star \psi'$
2. $\varphi' \leq \varphi$ *and* $\psi \leq \psi'$ *implies* $\varphi \mathrel{-\!\!\star} \psi \leq \varphi' \mathrel{-\!\!\star} \psi'$

The most striking property of this section is that quantitative $\star$ and $\mathrel{-\!\!\star}$ are adjoint operators:

Theorem 32 (Adjointness of $\star$ and $\mathrel{-\!\!\star}$)**.** $\pi \leq \varphi \star \psi \longleftrightarrow \psi \mathrel{-\!\!\star} \pi \leq \varphi$

A direct corollary is the following:

Corollary 33 (Quantitative Modus Ponens)**.** $\psi \leq \varphi \star (\varphi \mathrel{-\!\!\star} \psi)$

As already mentioned earlier, the key insight of this section is how to generalize the universal and existential quantification from predicates to potentials. It seems trivial after remarking it. I consider it another incarnation of the potential method.

Notes on the Formalization The verification of this theory was surprisingly smooth in Isabelle/HOL. I think it is because not much background theory is needed and the underlying algebraic structure allows abstract reasoning.

In the first iteration I directly verified the material from [8] with the help of the authors. There, I used the domain (*ennreal*, $\leq$, $*$, *1*). Conveniently, the definition of multiplication and division for *ennreal* for the corner cases *0* and ∞ matched the ones needed. Using a type γ which forms a separation algebra [69] gives rise to the commutative monoid ($\gamma \to$ *ennreal*, $\star$, [*emp*]).

After completing the first iteration, I wondered which properties are needed for the underlying type and operations. I generalized the theory and collected the necessary properties of the domain in order to allow the definition of quantitative separating connectives that form a separation algebra and fulfill the adjointness property (Theorem 32). I parameterized the theory with a type and operations $+$ and $-$.

With the established general theory it is easy to study other domains: e. g. the Boolean case which results in the standard Separation Logic connectives, and the domain for potentials (*ennreal*, $\geq$, $+$, *0*) which I presented in this section. For the latter, again the corner cases of $+$ and $-$ for *ennreal* are as needed.

I will comment on the generalization to quantales in Section 4.3. Before that, I will describe what we can use this assertion language for.

4.2 Quantitative Separation Logic (QSL)

In this section I describe my formalization of material from Section 4 of "Quantitative Separation Logic" [8] by Batz et al.. Again I apply it to *ert* instead of *wp*, but follow their ideas and structuring of the presentation.

With the quantitative separating connectives set up, we now want to modularly reason about the expected running time of probabilistic programs with pointer structures. For that we will extend pGCL with commands that manipulate the heap, adjust the *ert* transformer, the quantitative Hoare logic and prove a frame rule for it.

Note that, the resulting language (short hpGCL) is both an extension of pGCL and also morally of IMP. Thus by excluding the probabilistic and nondeterministic choice, we obtain a calculus that extends the quantitative Hoare logic from Section 2.3 but comes with the possibility for modular reasoning using the frame rule.

The heap-manipulating probabilistic guarded command language Figure 4.1 shows the definition of hpGCL as a deeply embedded type *hpgcl*. Additionally to the constructs in pGCL, we add four commands that are concerned with the heap: the command *New v e* reserves a field on the heap, stores the value of the expression *e* in the current state to that field, and sets the variable *v* to its address. Note that for simplicity values and addresses have the same type (here we choose integers). With the command *Free e* one can calculate an address from an expression and deallocate the heap portion at that address. The command *Lookup v e* calculates the expression *e*, looks up the

$$
\begin{aligned}
\mathit{hpgcl} = {} & \mathit{Skip} \\
& \mid \mathit{Assign}\ \mathit{vname}\ (\mathit{stack} \to \mathit{val}\ \mathit{pmf}) \\
& \mid \mathit{Seq}\ \mathit{hpgcl}\ \mathit{hpgcl} \\
& \mid \mathit{Par}\ \mathit{hpgcl}\ \mathit{hpgcl} \\
& \mid \mathit{If}\ (\mathit{stack} \to \mathit{bool})\ \mathit{hpgcl}\ \mathit{hpgcl} \\
& \mid \mathit{While}\ (\mathit{stack} \to \mathit{bool})\ \mathit{hpgcl} \\
& \mid \mathit{New}\ \mathit{vname}\ (\mathit{state} \to \mathit{val}) \\
& \mid \mathit{Free}\ (\mathit{stack} \to \mathit{addrs}) \\
& \mid \mathit{Lookup}\ \mathit{vname}\ (\mathit{state} \to \mathit{val}) \\
& \mid \mathit{Update}\ (\mathit{stack} \to \mathit{addrs})\ (\mathit{stack} \to \mathit{val})
\end{aligned}
$$

Figure 4.1: The deeply-embedded hpGCL language.

value at that address and writes it to the variable v. Finally, the command $\mathit{Update}\ e\ e'$, writes the value of the expression e' to the address e.

The *program state* $(s,\ h)$ of a hpGCL program consists of a *stack* and a *heap*. The former is just the state we know from pGCL and IMP. It is a function from variable names to values. The heap is a partial function from heap addresses to values. Thus heaps form a separation algebra. Note that all expressions in hpGCL are evaluated only with the stack. To calculate with values from the heap, they first need to be fetched via a *Lookup* into a local variable on the stack and then can be processed.

We lift assertions to program states adding the stack: they are functions from a pair of stack and heap to extended non-negative reals $((\mathit{stack} \times \mathit{heap}) \to \mathit{ennreal})$. We also lift the quantitative connectives and use the same notation, as in the rest of this section we always will talk about assertions on program states.

$$
\begin{aligned}
(A \star B)\ (s,h) &= (\lambda h_1.\ A\ (s,h_1)) \star (\lambda h_2.\ B\ (s,h_2)) \\
(A \mathrel{-\!\!\star} B)\ (s,h) &= (\lambda h_1.\ A\ (s,h_1)) \mathrel{-\!\!\star} (\lambda h_2.\ B\ (s,h_2))
\end{aligned}
$$

Also we introduce basic assertions: the empty heap $[\mathit{emp}]$, the lifted predicate heap assertion $[\varphi]$, and the points-to assertion $[e \mapsto v]$ describing a portion of the heap with the value v at the address e evaluates to in the current stack. The assertion $[e \mapsto \_]$ only restricts that there is exactly one field on the heap at the specified address, but the value is unknown.

Recall the *ert* rule for *Skip* for pGCL (Section 3.1): $\mathit{ert}\ \mathit{Skip}\ f = \mathbf{1} + f$. This would be a valid rule for hpGCL also: we add one to the expected running time of the continuation f. But we actually can express the addition of the potential *1* also differently. We define the potential assertion $\$\,c\ (s,\ h) = (\mathtt{if}\ h = 0\ \mathtt{then}\ c\ \mathtt{else}\ \infty)$. Then, as the empty heap is always orthogonal to any other heap, and the quantitative separating conjunction minimizes the value we have: $\mathbf{1} + f = \$1 \star f$. Note that we still work on heaps only. We did not add time credits to the separation algebra. Rather, the lifting from predicates to potentials allows to "give value" to some empty portion of the heap.

We now briefly go over the rules for *ert* in Figure 4.2. The first six rules are the same as for pGCL, only we can use the potential assertion $\$1$ to write them more succinctly.

$$
\begin{aligned}
&\mathit{ert} :: \mathit{hpgcl} \to ((\mathit{stack} \times \mathit{heap}) \to \mathit{ennreal}) \to ((\mathit{stack} \times \mathit{heap}) \to \mathit{ennreal})\\
&\mathit{ert}\ \mathit{Skip}\ f = \$1 \star f\\
&\mathit{ert}\ (\mathit{Assign}\ v\ u)\ f = \$1 \star (\lambda(s,\ h).\ \textstyle\int_x f\,(s(v := x), h)\mathsf{d}(u\,s))\\
&\mathit{ert}\ (\mathit{Seq}\ c_1\ c_2)\ f = \mathit{ert}\ c_1\ (\mathit{ert}\ c_2\ f)\\
&\mathit{ert}\ (\mathit{Par}\ c_1\ c_2)\ f = \mathit{ert}\ c_1\ f \sqcup \mathit{ert}\ c_2\ f\\
&\mathit{ert}\ (\mathit{If}\ g\ c_1\ c_2)\ f = \$1 \star (\lambda(s,\ h).\ \texttt{if}\ g\ s\ \texttt{then}\ \mathit{ert}\ c_1\ f\,(s,\ h)\ \texttt{else}\ \mathit{ert}\ c_2\ f\,(s,\ h))\\
&\mathit{ert}\ (\mathit{While}\ g\ c)\ f = \mathit{lfp}\ (\lambda W\ (s,\ h).\ 1 + (\texttt{if}\ g\ s\ \texttt{then}\ \mathit{ert}\ c\ W\ (s,\ h)\ \texttt{else}\ f\,(s,\ h)))\\
&\\
&\mathit{ert}\ (\mathit{New}\ v\ e)\ f = \$1 \star \mathit{Sup}_{a\in\mathbb{Z}}\ ([a \mapsto e] \mathrel{-\!\!\star} (\lambda(s,\ h).\ f\,(s(v{:=}a),h)))\\
&\mathit{ert}\ (\mathit{Free}\ e)\ f = \$1 \star [e \mapsto \_] \star f\\
&\mathit{ert}\ (\mathit{Lookup}\ v\ e)\ f = \$1\star\ (\lambda(s,\ h).\ \mathit{Inf}_{x\in\mathbb{Z}}(([e \mapsto x] \mathrel{-\!\!\star} (\lambda(s,\ h).\ f(s(v{:=}x),\ h))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \star\ [e \mapsto x])\ (s,\ h)))\\
&\mathit{ert}\ (\mathit{Update}\ e\ e')\ f = \$1 \star [e \mapsto \_] \star ([e \mapsto e'] \mathrel{-\!\!\star} f)
\end{aligned}
$$

Figure 4.2: The rules of the *ert* calculus for hpGCL.

The latter four need more explanation. Those rules basically follow the same ideas as the *wp* rules in Section 4.2 of Batz et al. [8]. We will only explain the rule for memory allocation, more details on the intuition of the other rules can be found in the mentioned resource.

Memory allocation The memory allocation statement *New v e* exhibits unbounded nondeterministic behavior. Operationally, the statement starts from some initial state $(s,\ h)$. Then the memory manager allocates a single field at a *fresh* address a and writes the evaluated expression $e\ s$ to that memory cell. After allocating the cell, its address is stored in variable v, and we obtain a final state $(s',\ h')$. Since a is chosen *nondeterministically*, we cannot give any a-priori guarantees on a except for $a \notin \mathit{dom}\ h$. Notice that we assume in our memory model there are at any point infinitely many free addresses available for allocation — this command will never cause a memory fault.

We now search for the expected running time of *New v e* with respect to a continuation f. We need to find an expectation $\mathit{ert}\ (\mathit{New}\ v\ e)\ f$ that evaluated in the initial state coincides with the quantity f evaluated in the final state. To construct that expectation, we evaluate f in the initial state and rectify the differences to measuring f in the post state. The first difference is, that in the initial state the memory cell at address a is not yet allocated. We can rectify this by demanding an extension to the heap that fulfills the postassertion f. To express that, we can use the quantitative separating implication: $[a \mapsto e] \mathrel{-\!\!\star} f$. Notice that the heap h' with a single memory cell holding $e\ s$ at address a is the only valid extension that satisfies $[a \mapsto e]$, in this case the supremum in the quantitative separating implication only is taken over that h'. The continuation will actually be evaluated with the variable v set to the address a, we can rectify this change of the stack by a syntactic replacement: $[a \mapsto e] \mathrel{-\!\!\star} (\lambda(s,h).\ f\,(s(v{:=}a),\ h))$. Now, the newly allocated address was chosen nondeterministically. As we chose *ert* to express

upper bounds on the expected running time w. r. t. demonic nondeterminism, we have to take the "worst-case" which is here a supremum over the results for all the possible addresses $Sup_{a\in\mathbb{Z}}\ ([a \mapsto e] \mathrel{-\!\!\star} (\lambda(s,h).\ f\ (s(v{:=}a),\ h)))$. Finally, we have to add the costs for the command and we obtain

$$ert\ (New\ v\ e)\ f = \$1 \star Sup_{a\in\mathbb{Z}}\ ([a \mapsto e] \mathrel{-\!\!\star} (\lambda(s,\ h).\ f\ (s(v{:=}a),\ h)))$$

.

Properties of *ert* As for the *ert* function for pGCL we already saw earlier, we have similar properties for the *ert* on hpGCL. Because of the unbounded nondeterminism in the memory allocation command, *ert* is not continuous anymore. But it is still monotone. See Section 4.3 in [8] for a discussion.

Quantitative Frame Rule The main motivation for Separation Logic is to allow modular development, in particular the *frame rule* allows for local reasoning. We have already seen it in Section 2.4. Essentially, it states that a part of the heap that is not explicitly modified by a program is unaffected by that program. It suffices to reason locally about the parts of the heap the program actually touches, and it is possible to add disjoint untouched portions of the heap later. The frame rule reads as follows:

$$\frac{\{\varphi\}\ c\ \{\psi\}}{\{\varphi \star \eta\}\ c\ \{\psi \star \eta\}}\ \text{if } Mod\ c \cap Vars\ \eta = \emptyset.$$

Here, *Mod c* is the set of variables that are updated by the program c, i. e. all variables appearing on a left-hand side of an assignment. Moreover, *Vars* η is the set of variables that occur in η (cf. Section 3.3). The rule intuitively reads: if a program c started on a portion of the heap satisfying φ terminates with out an error in a partial heap satisfying ψ we can safely add a disjoint portion of the heap to the pre and postcondition (referred to as *the frame* η). The side condition rules out that c changes variables on the stack that η depends on.

Note that we have two kinds of separation here. First, the separating conjunction makes sure that the heap portions the assertions talk about are disjoint, otherwise the precondition would be false and the Hoare triple trivially true. And, in order to thread through the frame η we have to make sure that it does not change its truth value. We already made sure that the heap does not change (with the separating conjunction guaranteeing disjointness of the heap portions), but the program might alter variables on the stack that would then alter the truth value of the frame. Thus we need to add the side condition.[2]

As for the probabilistic quantitative Hoare logic, we can use the *ert* calculus to define Hoare Triples in hpGCL:

[2]Note that in Section 2.4 we did not have that additional side condition, as we only had a stack, and no heap, and the separation algebra acted on the stack, ensuring disjoint sets of variables, and additionally that a Hoare triple that accesses or writes a variable without the ownership of that variables can not be proven correct.

$$\models_H \{\varphi\}\ c\ \{\psi\} \longleftrightarrow \varphi \geq ert\ c\ \psi$$

The triple is valid if the prepotential is an upper bound on the expected running time of c with continuation ψ. As hpGCL is a conservative extension of pGCL all the Hoare inference rules mentioned in Section 3.2 are still valid. Adding the appropriate rules for the four new commands is straightforward, and we now turn to proving the frame rule.

After unfolding the definitions the goal boils down to the following theorem.[3]

Theorem 34 (Quantitative Frame Rule)**.** *For every hpGCL-program c and expectations φ, ε with $Mod\ c \cap Vars\ \varepsilon = \emptyset$, we have $ert\ c\ (\varphi \star \varepsilon) \leq ert\ c\ \varphi \star \varepsilon$*

Proof. We can prove this theorem by structural induction on the program c. What we have to do for all the cases is essentially to pull the infimum operator from the separating conjunction that is buried below the *ert* on the left-hand side towards the outside. Then we obtain the right-hand side of the inequality. This works, as all the operations in *ert* (integral, addition) are continuous, only for the four new commands we need to pull the infimum through some supremum or reorder two infima, which is both admissible as we only require an inequality. For loops, we can not rely on standard fixpoint induction over countable iterations as *ert* is not continuous. But as it still is monotone, we can employ transfinite induction.[4] □

Note that the converse direction of the inequality does not hold. That is also the case in the *wp* instance studied by Batz et al. [8].

Soundness of the *ert* calculus In this thesis I do not provide a soundness proof of that calculus w. r. t. an operational semantics, but I comment on a first attempt and the pen-and-paper proof by Batz et al..

A first approach would be to extend the existing proof for pGCL we have seen in Section 3.1. But it is not clear to me how to extend the proof to unbounded nondeterminism. For a first approach I tried to extend the normal pGCL semantics with a nondeterministic choice of some infinite set. Fixing that soundness proof boils down to weakening a finiteness assumption (cf. Lemma *E_inf_lfp* in Theory "Markov_Decision_Process" in *Markov_Models−AFP*). Hölzl states that "it is not clear how to remove it or even weaken it" when talking about that Lemma 27 in [63].

On the other hand, Batz et al. provide a soundness proof for their *wp* calculus w. r. t. to an operational semantics on MDPs with a reward function and a *scheduler* to model nondeterminism. Formalizing that operational semantics and the equivalence proof is interesting future work.

[3] Batz et al. [8] sketch a similar derivation for *wp* in Section 4.7

[4] I use the induction rule *lfp_lockstep_induct* from theory *Lib* in Platzer's [120]. Thanks to the Isabelle search tool for helping me find that very theorem.

Conservative Extension The *ert* calculus of this chapter is a conservative extension to the *ert* calculus without the heap (cf. Section 3.1). This can be seen straightforwardly: if we discard the last four commands we obtain the same *ert* rules.

If we further drop the nondeterministic and probabilistic choice we obtain the quantitative Hoare logic from Section 2.3.

Notes on the formalization Modeling hpGCL in Isabelle/HOL is straightforward, given the work on pGCL from earlier verifications. The primitive pure assertions on the heap model are lifted routinely, and the quantitative separating connectives result from the formalization described in Section 4.1.

I consider the formal proof of the quantitative frame rule (Theorem 34) as the main contribution of this section. For the *While* case a specific induction rule was needed that does not require *ert* to be continuous, but works with *ert* only being monotone. The proof should be easily portable to other domains (Boolean, and $([0; 1], \leq, *, 1)$).

Before applying the *ert* calculus for hpGCL to applications, soundness must be established. The unbounded nondeterminism seems to prevent an extension of the existing soundness proof for pGCL. Instead, a different model of the operational semantics needs to be formalized and soundness of *ert* can be established relative to it.

4.3 Quantales

In the last sections we have seen how Separation Logic can be lifted following the "zen of the potential method" and gives rise to a program logic for expected running time of heap-manipulating probabilistic programs. While the main ideas stem from the paper "Quantitative Separation Logic" by Batz et al. [8], I presented a different version of QSL that fits better into the scope of this thesis. However, both versions are instances of the same general theory. I discovered that general theory after formalizing the material of Batz et al. and then using the proof assistant to abstract from the concrete instance. Walter Guttmann pointed out to me the structure of the underlying algebraic object as *quantales*.

In my opinion, a big strength of proof assistants is that after formalizing some theory one can easily experiment with generalizing it. Collecting necessary properties of structures or preconditions of proofs is assisted by the tool.

In the first part of the remainder of this section I will sketch the general theory of quantitative separating connectives parameterized by a quantale. Then I discuss possible instances and their use. The discovery of the concept of quantales, opened up a whole new world of related work. I collect and summarize relevant literature and conjecture some connections to other fields. This review is kept short, as it mostly consists of conjectures and ideas for future work, but no proper contributions or formalized results.

4.3.1 The Generalized Quantitative Separating Connectives

Now that we established the quantitative separating connectives in Section 4.1 we can ask ourselves what kind of structures are needed for that kind of theory. Intuitively, extended non-negative real numbers (*ennreal*) are too specific and we need to identify which kind of structure we need in order to obtain QSLs. One could now go through Section 4.1 and abstract all types and operations and collect what properties are needed for them in order to get the proofs through. That is exactly what I did after verifying the QSL version featured in [8].

Let me reconstruct that process, and collect the necessary properties needed. We started to model assertions ($M \to Q$) as functions from a separation algebra (M) to a measuring type Q. First we come up with an embedding ($[\cdot]$) of Boolean assertions ($M \to \mathbb{B}$) into quantitative assertions. For that we demand Q to have a *complete lattice* structure with an ordering $\leq$ and top ($\top$) and bottom ($\bot$) elements.[5] In the embedding the bottom element corresponds to False and the top element to True, giving rise to $(P \leq Q) \longleftrightarrow [P] \leq [Q]$.

The measuring type (Q) additionally needs to provide a composition operation $\oplus :: Q \times Q \to Q$ and a neutral element (0). The first property about that operation is to show that its lifting to assertions is a conservative extension of the standard conjunction ($\wedge$), i. e. $[P \wedge Q] = [P] \oplus [Q]$. That already suggests that (Q, $\oplus$, 0) needs to form a commutative monoid.

As a next step we define the general quantitative separating conjunction ($\star$). We need an supremum operator defined on Q,[6] the composition operation $\oplus$ on Q and the disjointness and sum operations on M. With Q being a complete lattice and M being a separation algebra we have all the necessary operations. For the quantitative separating implication ($\mathrel{-\!\!\star}$) we additionally need an adjoint for the operation $\oplus$. We denote that operation with $-$.

We can now prove that the generalized connectives are conservative extensions to the standard operators of Separation Logic. Monotonicity of $\oplus$ w. r. t. the ordering on Q can be lifted to monotonicity of $\star$ on assertions $M \to Q$. In order to prove associativity of $\star$ we need the fact, that we can distribute the operation $\oplus$ over the supremum. Essentially that is the distributive property required for (Q, $\leq$, $\oplus$) to be a *quantale*:

$$c \oplus Sup\ A = Sup_{x \in A}\ (c \oplus x)$$

As we already required $\oplus$ to be commutative, we also obtain the commuted distributive property. Furthermore, we required (Q, $\oplus$, 0) to be a commutative monoid and we can lift the neutral element to functions to obtain $0 :: M \to Q$. With that we can now also show that ($M \to Q$, $\star$, 0) is a commutative monoid. Here, I also need to note that if the neutral element is the top element — which is the case for the instance in Section 4.2, the Boolean instance, and the instance for probabilities — we have some more convenient properties.

[5] In the instance in Section 4.2 the ordering was flipped, so $\top = 0$ and $\bot = \infty$.

[6] In general we use the supremum. Note that because we had to flip the ordering in our running example instance, we use the infimum operator for the definition of $\star$.

To sum it up, the structure we need for $(Q, \leq, \oplus)$ is a commutative quantale. The main property we want to prove between $\star$ and $\mathrel{-\!\!\star}$ is adjointness:

$$\pi \leq \varphi \star \psi \longleftrightarrow \psi \mathrel{-\!\!\star} \pi \leq \varphi$$

Many other important properties are direct corollaries of that theorem. Proving it boils down to requiring the (guarded) adjointness property of $\oplus$ and $\ominus$:

$$\bot < C \vee \bot < B \wedge C < \top \vee B < \top \implies (A \leq B \ominus C) \longleftrightarrow (A \oplus C \leq B)$$

Actually, once we have that $(Q, \leq, \oplus)$ is a commutative quantale, the existence of an operation that fulfills the adjointness property is guaranteed. Then the $-$ operation has to coincide with that operation on all the cases except the ones ruled out by the guard in the above property. In order to allow more flexibility, I came up with locales that require the adjoint operation $-$ and the guarded adjointness property. When provided with a commutative quantale $(Q, \leq, \oplus)$, one can directly obtain the adjoint operator and instantiate the locale.

In the following I go through all the instances that are known to me and which I have actually implemented and formalized in Isabelle/HOL.

Boolean Quantale We have already seen, that the quantitative separating connectives generalize the standard Separation Logic operators. When setting $Q := \mathbb{B}$, $\leq := \implies$ and $\oplus := \wedge$ we obtain the standard Separation Logic over the separation algebra M: the embedding $[\cdot]$ is the identity function, Supremum is the existential quantifier, Infimum the all quantifier, the adjoint operation is $a - b \longleftrightarrow (b \implies a)$, and consequently the instantiated quantitative separating conjunction and implication are equivalent to their standard counterparts. For example, the adjointness property reads:

$$(a \implies (c \implies b)) \longleftrightarrow ((a \wedge c) \implies b)$$

We can use that instance, together with a program logic with resources (e. g. variable names are resources as in Section 2.4, or the heap is a resource, or we can even add time credits, or other resources) we can come up with a weakest precondition calculus *wp* and use the established assertion language. The total correctness triple $\{\varphi\}\ c\ \{\psi\}$ then expresses that program c started from a state that satisfies φ terminates in a final state that fulfills ψ.

Quantale for Numbers The second instance for the general theory is the one chosen for our running example in this chapter: we set $Q := \mathit{ennreal}$, $\leq := \geq$, $\oplus := +$ and the rest is already known. This structure is called *Lawvere's quantale* [95].

Extending that with an expected running time calculus for some (probabilistic) programming language with resources then yields a quantitative Hoare logic over assertions from a Quantitative Separation Logic. The judgment $\{\varphi\}\ c\ \{0\}$ then expresses that program c has expected running time no more than φ.

Note that, if we restrict ourselves to $Q = \mathit{enat}$ we can develop the same theory, as $(+)$ and its adjunct $(-)$ are closed for *enat*, and apply it to IMP.

Quantale for Probabilities and Expectations The instance originally used in Batz et al. [8] is tailored to expectations. We obtain their instances when using assertions into extended non-negative real numbers (their $\mathbb{E}$) or numbers in the unit interval (their $\mathbb{E}_{\leq 1}$), together with multiplication and the standard ordering on reals. While the latter instance fits into the general framework of quantales, the former instance feels more ad hoc and some alterations are needed in order to fit it into that structure. In particular, for proving conservativity of QSL as an assertion language some non-canonical changes are necessary.[7]

They go on and define a weakest preexpectation calculus (cf. *wp* from Section 3.3) on hpGCL and use that to reason about programs. What kind of questions can they answer with that approach?

First, by determining $wp\ c\ [\Phi]$ for a program c and an embedded Boolean assertion Φ one can evaluate for a starting state s a lower bound on the probability that Φ holds after the execution of c from state s. For example, they calculate the probability that a certain array configuration is reached after calling a procedure that is supposed to randomly shuffle an array. Or they calculate a lower bound for the probability that a faulty garbage collector — that iterates over a tree of elements, and "forgets" to free an element with some probability $p > 0$ — leaves an empty heap after the operation $(wp\ c\ [emp])$. It turns out that the probability only makes sense if one started with a proper tree and the probability that the program did never "forget": $[tree(x)] \star (1-p)^{size}$. Where $tree(x)$ is a Boolean assertion describing a *binary tree* at address x and *size* is an assertion mapping a part of the heap to its size.

Second, not only preprobabilities for postpredicates (they live in $\mathbb{E}_{\leq 1}$) can be determined. Also proper preexpectations for a given postexpectation (expectations in $\mathbb{E}$) are possible. Here is an example, what can be done with it: Consider the assertion $len(x,\ 0)$ for an address x that maps a partial heap containing a linked list to the length of the list, and to ∞ otherwise. The program c repeatedly flips a coin (that gives Heads with probability p) and prepends an element to a list until it gets Heads as a result. Asking for $wp\ c\ (len(x,\ 0))$ gives the expected length of the list evaluated in the prestate. For $p = 0.5$ on average the list length will increase by one: we have $wp\ c\ (len(x,\ 0)) = len(x,\ 0) + 1$.

Similar to the quantitative frame rule we proved for the expected running time calculus (Theorem 34) they prove a rule for their *wp* calculus. In the faulty garbage collector example, they apply the rule when using the "Hoare triple" for the recursive delete operation, which only works on a part of the heap, in the body of the delete operation.

In Chapter 2 we have seen that we can either use the lifting from predicates to potentials, or the usage of time credits to enable reasoning about the running time of programs. We have now seen that the first idea can also be used to lift Separation Logic from Boolean assertions to assertions that essentially are probabilities. It is particularly interesting what would happen if one adds time credits to the *wp* calculus that uses QSL for probabilities. Maybe then $\{[\$m] \star \varphi\}\ c\ \{[\psi] \star \mathbb{T}\}$ expresses that φ is a lower

[7] For more details see the end of Section 3.2 in [8].

bound on the probability that c terminates in a state satisfying ψ consuming at most m time steps.

As I only discovered quite late that the theory can be phrased generally for quantales, I only then discovered that there is already a formalization of *quantales*: Struth [125] formalizes quantales as a type class in Isabelle/HOL. It is future work to connect the formalization of quantitative separating connectives to it. Furthermore, this discovery opened up a whole new area of research up for me. I will comment on that in the following section.

4.3.2 Quantales Everywhere

As already said, the discovery of quantales being the underlying structure that enables quantitative separating connectives to make sense, lead me to a new area of research. In this section I describe only the tiny fraction of that work, that is immediately related and still within scope of this thesis. Exploring more of that material and spelling out the "zen of the potential method" in that realm may involve some very interesting research. But that road was not taken within this thesis project. Now let me explain the connection and give a quick exploration into the field.

In 2011, Dang et al. [25] present an algebraic approach to Separation Logic. They show that the structure of Separation Logic is a commutative quantale. To phrase that in our terms, they show that the structure $(heap \to \mathbb{B}, \leq, \star, emp)$ is a commutative quantale. Then they observe that the separating implication ($-\!\!\star$) corresponds to the residual of the quantale (Lemma 5.4 in [25]) and many laws about $-\!\!\star$ can be proved from the standard theory of residuals. Also many other identities and rules of classical Separation Logic can be automatically proved in the realm of quantales, as these propositions can be reformulated algebraically in first-order form. They carry on abstracting many related properties and phrasing them in terms of quantales (e. g. classes of assertions, frame rule) but the main idea of that paper for me is to show that standard Separation Logic assertions ($heap \to \mathbb{B}$) form a commutative quantale. In order to distinguish between different "levels" of quantales I call this an *assertion quantale*.

In 2015, Dongol, Gomes and Struth [28] go some steps further. They use functions $f :: M \to Q$ from a partial monoid M into a quantale Q, and call them *power series*. To avoid confusion I will call that quantale Q at the base of the power series the *measurement quantale* and its operation *addition* denoted by $+$.[8] The addition operation from the Quantale can be lifted pointwise to power series and multiplication is *convolution*:

$$(f \otimes g)\ x = Sup_{x=y+z}\ f\ y \oplus g\ z$$

Here $+$ acts on M, $\oplus$ acts on Q and $\otimes$ acts on $Q^M = (M \to Q)$. This very much looks exactly like the definition of separating conjunction from quantitative Separation Logic (cf. Definition 2). They carry on showing that a quantale and a partial monoid (which is essentially a separation algebra if it also is commutative) give rise to

[8] This matches the intuition build up with our running example, where $Q = ennreal$ and the operation is addition of numbers.

an assertion quantale $(Q^M, \leq, \otimes, 1)$. But then they only instantiate the measurement quantale as the Boolean semiring and interpret power series as characteristic function (i. e. predicates). Pairing the resource monoid with a stack, they use this mechanism to define Separation Logic. They refer to Dang et al. [25] for the definition of magic wand wand as (upper) adjoint and note that the quantale setting gives lots of theorems for free. Essentially, they already came up with the abstraction to use general measurement quantales and deduce many useful theorems for the assertion quantale and its connectives.

But in the rest of their paper they only use $Q = \mathbb{B}$. Also later work by Dongol, Hayes and Struth [29] that explores the unifying concept of convolution only considers generalizations and many examples for M.[9]

Dongol, Gomes and Struth [28] further consider the *predicate transformer* quantale that does not only work on states but also incorporates program behavior. Gomes [37] further develops that algebraic approach and presents infrastructure for program verification formalized in Isabelle/HOL.

What at least is remarkable, is that to the best of my knowledge for the quantale Q only the Boolean semiring was used. I think that the focus on that case was due to the applications that arise from them in program verification. With the potential method, and also the reasoning about probabilistic programs, I might have identified two applications with other instances of quantales Q. I suspect that there are connections to category theory and there exist further domains that might fit into this framework, e. g. fuzzy logic, quantum states, and quantum Hoare logic [97].

It would certainly be interesting future work to explore those applications and what properties one would get if they are phrased in the general algebraic framework that has already been identified.

Because this is not the main topic of this thesis, I do not follow that road any further. Instead, in the rest of this thesis I will use the approach to use Separation Logic with Time Credits — as described in Section 2.4 — and I will present a practical verification framework to reason about the running time of imperative programs in Isabelle/HOL in the next chapter.

4.4 Limitations and Future Work

Before turning to a practical verification framework to reason about the running time of deterministic imperative programs in Isabelle/HOL, let me present some limitations and ideas for future work.

While QSL is expressible enough to reason about the expected running time of probabilistic programs, there is not yet enough reasoning infrastructure to tackle serious case studies. I did not provide any verified example for QSL. The examples I considered either are trivial and do not convey any new knowledge, or they are very hard and need considerable amount of work. In particular, a good test piece would be the

[9] From private communication with one of the authors I know that they do consider other domains for Q in a forthcoming article, and are considering applying their work to quantitative examples.

verification of randomized meldable heaps in hpGCL. That data structure was first verified in QSL by Hannah Arndt [1]. An improved version of her proof can be found in Matheja's PhD thesis [100, §8.5].

Lifting common techniques in Separation Logic and implementing them for QSL would be one way to go. Another idea is to use existing infrastructure for deterministic heap-manipulating programs to prove Hoare triples correct for the fragment of hpGCL without probabilistic choice. Many randomized algorithms might take a random choice in the beginning and execute a deterministic algorithm following that choice, or at least uses some subprocedure that does not involve random choice. The *average case time complexity* of deterministic algorithms can be modeled by adding a random choice of the input following a given distribution in the beginning of the algorithm. Coming up with tools for this integration might be straightforward, finding and conducting meaningful case studies is certainly within reach.

Studying what parts of the algebraic approach for program correctness tools [37] can be lifted to the quantitative analysis of programs is certainly exciting future work. It might be fruitful to study other parts of this thesis in the light of quantales: for example the generic *wp* framework (Section 6.1.3) and the NREST monad (Chapter 8).

4.5 Summary

- We have seen that Separation Logic can be lifted from Boolean assertions to quantitative assertions and sensible quantitative separating connectives can be defined.
- The resulting Quantitative Separation Logic can be used as an assertion language for heap-manipulating probabilistic programs (hpGCL) and we can reason about the expected running time of such programs.
- Finally we have seen, that the Quantitative Separation Logic has the general concept of quantales underlying. We explore related work on quantales and program verification and conjecture some ideas for future work.

Part II

Verifying Asymptotic Time Complexity of Imperative Programs

In Chapter 2 I identified Atkey's approach to use Separation Logic with Time Credits to reason about the running time of imperative programs to be the most promising approach for interactive theorem proving. In this middle or second part of this thesis I present two incarnations of that idea and add verification tools in order to make verification of the functional correctness and running time analysis of algorithms practical.

First, I present the shallowly-embedded programming language Imperative-HOL-Time with a cost model, develop a compositional methodology to verify the running time of algorithms in that language and provide automatic tools to support the individual verification tasks (Chapter 5). From a monadic program in Imperative-HOL-Time we can extract code for hybrid programming languages such as OCaml or SML, which allow imperative features such as references and mutable arrays. To illustrate the applicability I show several case studies of small to medium size algorithms and data structures.

Then, I present a second verification framework by Lammich that is built up quite modularly and can be instantiated by any program logics that provides a Separation Logic and a weakest precondition predicate. We instantiate it with a shallowly embedded LLVM semantics [81] that comes with a fine-grained cost model (Chapter 6): it counts how many operations of each LLVM instruction are used in a computation. In order to support this in Separation Logic I introduce time credits with *currencies*. From the verified LLVM programs I can extract LLVM text that can be compiled by the LLVM compiler to efficient executable code. I provide verified implementations of basic data structures and operations.

5 Imperative-HOL-Time

The content of this chapter is joint work with Bohua Zhan. This chapter is based on the paper "Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle" (Zhan and Haslbeck [138]).

I identified that Separation Logic with Time Credits seems to be the method to use when it comes to verifying the running time analysis of imperative programs in an interactive theorem prover. In this chapter I take that idea and implement a verification framework that allows the simultaneous verification of functional correctness and running time of Imperative-HOL-Time programs.

To achieve that goal we have collected ideas and verification projects from several sources and compose them into a usable verification environment.

First of all, we have used the idea of Atkey to use Separation Logic and Time Credits, which was used by Charguéraud and Pottier [21] to verify an involved data structure in Coq. To realize that in Isabelle/HOL, we have extended Imperative-HOL [13] with a cost semantics (Section 5.1), and its Separation Logic [90, 137] with time credits.

We follow the insight, that the program logic should work on concrete costs, whereas abstracting to the asymptotic complexity should happen at the end of an analysis. Guéneau et al. [41] propose a way how to integrate asymptotic complexity bounds into specifications and present a methodology that supports a natural reasoning style. In Section 5.2 I spell out that approach by showing how we compose modular specifications and presenting a methodology on how to tackle the analysis of imperative programs. We divide the work into three clearly-separated parts: the proof of functional correctness, reasoning about Separation Logic and the analysis of asymptotic behavior of the running time functions. While the first part is very algorithm specific, the latter two can be supported by automatic tools.

Our automation setup for reasoning about Separation Logic with Time Credits is described in Section 5.3. We adapt and extend automation by Zhan [137] and Lammich [84, §2.5] for Imperative-HOL with support for time credits.

For the complexity analysis, we build upon the formalization of Landau symbols by Eberl [31]. To automatically analyze divide-and-conquer algorithms we can use a stronger form of the Master theorem verified by Eberl [32]. Building upon those two developments we provide a proof tactic to automatically determine the asymptotic behavior of running time functions in one or two variables (Section 5.4).

Finally, I demonstrate the broad applicability of our framework with several case studies (Section 5.5). For some of the algorithms we reuse existing work on functional algorithms [33, 109, 134, 137] and integrate it into our framework, for others we develop the verifications completely on our own. This includes case studies involving advanced

techniques for running time analysis such as the use of the Akra–Bazzi theorem (for mergesort, median of medians selection, and Karatsuba's algorithm) and amortized analysis (for dynamic arrays, skew heaps, and splay trees). I also provide an example (Knapsack problem) illustrating asymptotic complexity on two variables.

5.1 A Cost Model for Imperative-HOL

This section is based on Section 3.1 of the paper "Refinement with Time — Refining the Run-Time of Algorithms in Isabelle/HOL" (Haslbeck and Lammich [45]).

Imperative-HOL with time [138] incorporates Atkey's [2] idea to include *time credits* in Separation Logic into the Imperative-HOL [13] framework. In essence, it enables reasoning about imperative programs and their running time in Isabelle/HOL. While all the details can be found in Section 2.1 of [138], I will give an abstract explanation here that suffices for our purposes.

A procedure in the monad takes a heap as input and can either fail or return a tuple consisting of a return value, a new heap and a natural number, specifying the number of computation steps used. The type of a procedure with result type α is given by:

$$\alpha\ Heap = Heap\ (heap \rightarrow (\alpha \times heap \times nat)\ option)$$

The bind operator as well as fixpoint iteration, `while` and other combinators are defined in a straightforward manner. The term $(h,\ c) \Rightarrow (r,\ h',\ t)$ expresses that procedure c started on heap h does not fail and takes time t to produce result r and heap h'.

While heaps themselves do not form a separation algebra, there is an abstraction function *abs* that maps a pair of heap and time credits to an abstract heap.[1] Abstract heaps together with suitable definitions of disjointness and heap addition form a separation algebra. An assertion P, i. e. a mapping from an abstract heap to bool, being true for a heap h and time credits n is denoted by $abs(h, n) \models P$. There are basic assertions for an abstract heap containing an array without time credits $(a \mapsto_a xs)$, references without time credits $(r \mapsto_r v)$ and time credits ($\$n$).

Recall that, the *separating conjunction* $P \star Q$ expresses that the heap and time credits can be partitioned into two disjoint parts satisfying assertions P and Q respectively. As noted before, the strength of Separation Logic is that this disjointness enables modular reasoning, which also carries over to reasoning about time credits.

Hoare triples are defined in the following way:

```
1   <P> c <λr. Q r> =
2     (∀h n. abs (h,n) ⊨ P ⟶ (∃h' t r. (c,h) ⇒ (r, h', t)
3                                ∧ abs (h', n − t) ⊨ Q r ⋆ true ∧  t ≤ n) )
```

where the assertion *true* is true for any heap, thus enabling garbage collection of heap elements and time credits.[2] The Hoare triple $<P>\ c\ <\lambda r.\ Q\ r>$ denotes that procedure

[1] I will comment on that simplification in Section 5.7.

[2] In general, we call this "garbage collection assertion" the *affine top* $\mathbb{T}$ [20]. It determines what portions of the heap can be safely discarded. In the case of Imperative-HOL-Time we have $\mathbb{T} = true$.

c started from a heap satisfying P terminates with a return value r in a resulting heap that satisfies $Q\ r \star true$. In particular it states that the starting heap holds enough time credits n in order to pay for the cost t of executing the procedure c (see line 3).

The cost model assigns most basic commands (e. g. accessing or updating a reference, getting the length of an array) to consume one unit of computation time. Commands that operate on an entire array take $n + 1$ units of computation, where n is the length of the array. Examples for basic commands are:

$<a \mapsto_a xs \star \$1 \star \uparrow(i < |xs|)>\ Array.upd\ i\ x\ a\ <\lambda r.\ a \mapsto_a xs[i:=x] \star \uparrow(r = a)>$
$<\$(n + 1)>\ Array.new\ n\ x\ <\lambda r.\ r \mapsto_a replicate\ n\ x>$

where $\uparrow P$ is a pure assertion, which is valid for an empty heap if P holds globally, $xs[i:=x]$ denotes a list xs updated at position i with value x, and $replicate\ n\ x$ denotes a list of n elements x.

Observe, that a Hoare triple of the form $<P \star \$\, n>\ c\ <Q>$ implies that the procedure c costs *at most* n time credits. We very often state Hoare triples in this form, and so only prove upper bounds on the computation time of the program.

5.2 Methodology

In this section, I describe our strategy for organizing the verification of an imperative program together with its time complexity analysis. The strategy is designed to achieve the following goals:

- The proof of functional correctness of the algorithm should be separate from the analysis of memory layout and time credits using Separation Logic.
- The analysis of time complexity should be separate from proof of correctness.
- The time complexity analysis should work with asymptotic bounds Θ most of the time, rather than with explicit constants.
- Compositionality: verification of an algorithm should result in a small number of theorems, which can be used in the verification of a larger algorithm. The statement of these theorems should not depend on implementation details.

We first explain how we specify Imperative-HOL-Time programs to allow for a modular development process. Then we explain how to structure proving specifications: We first consider the general case and then describe the additional layer of organization for proofs involving amortized analysis.

5.2.1 Specifications

A specification of a program that is exhibited to a user consists of two facts: First, a Hoare triple stating the functional correctness and validity of a concrete running time bound; and second, a fact expressing that the running time bound has a certain complexity class.

Example 5.2.1. Consider the program $atake_{impl}\ n\ p$. We specify it to take an array at address p and initialize an additional array that holds the first n elements of a, while only using linear time in the length of the array. Let us denote its running time bound by $atake_{time}$, then we have the following lemmas in the specification:

$<p \mapsto_a as \star \$(atake_{time}\ |as|)>\ atake\ n\ p\ <\lambda r.\ r \mapsto_a take\ n\ as \star p \mapsto_a as>$
$atake_{time} \in \Theta(n)$

Here *take n as* for a functional list *as* is the list containing the first n elements of *as*.

Note that we use a Θ-bound here. This is because we need the stricter information for the application of Akra-Bazzi (cf. Section 5.4). But this is not a restriction, as we are only interested in upper bounds on the running time, and any O-bound can easily be turned into a Θ-bound by over-approximating it (for $f \in O(g)$ and $f' := max\ f\ g$ it holds $f' \in \Theta(g)$ and $f \leq f'$).

Observe that the proof engineer has to specify how the input is measured, i. e. what quantity is applied to the running time function. Here the length of the abstract list $|as|$ gives the size of the input. An alternative is to make the running time function depend on the abstract parameters (e. g. here $atake_{time} :: \alpha\ list \to nat$). Then, in the second theorem, the Θ needs to specify the filter, i. e. how the input list should be measured. While this might give rise to a cleaner theory for composing running time functions, we did not follow that road for simplicity.

Guéneau et al. [41] chose to package up functional correctness together with the complexity claims into one predicate *specO* hiding the running time bound behind an existential quantifier. We separate the theorems and give the bound a name. However, the users of that specification should — as they never have to look into the definition of the program — never have to look into the definition of the running time bound but only work with the fact about its asymptotic behavior.

5.2.2 General Case

Let us consider the general case first. For a procedure with name *f*, we define three Isabelle functions:

f_{fun}: The functional version of the procedure.

f_{impl}: The imperative version of the procedure.

f_{time}: The running time function of the procedure.

The definition of f_{time} should be stated in terms of running time bounds of procedures called by f_{impl}, in a way parallel to the definition of f_{impl}. If f_{impl} is defined by recursion, f_{time} should also be defined by recursion in the corresponding manner.

The theorems to be proved are:

1. The functional program f_{fun} satisfies the desired correctness property.
2. A Hoare triple stating that f_{impl} implements f_{fun} and runs within f_{time}.

```
merge_sort_impl X = do {
  n → Array.len X;
  if n ≤ 1 then return ()
  else do {
    A → atake (n div 2) X;
    B → adrop (n div 2) X;
    merge_sort_impl A;
    merge_sort_impl B;
    merge_list_impl (n div 2)
      (n − n div 2) A B X
  }
}
```

```
merge_sort_fun xs =
 (let n = length xs in
  (if n ≤ 1 then xs
   else
    let as = take (n div 2) xs;
        bs = drop (n div 2) xs;
        as' = merge_sort_fun as;
        bs' = merge_sort_fun bs;
        r = merge_list_fun as' bs'
    in r
  )
 )
```

Figure 5.1: An Imperative-HOL-Time implementation of mergesort and its corresponding functional version of the algorithm.

3. The running time f_{time} satisfies the desired asymptotic behavior.
4. Combining 1 and 2, a Hoare triple stating that f_{impl} satisfies the desired correctness property, and runs within f_{time}.

Here, the proof of Theorem 2 is expected to be routine, since the three definitions follow the same structure. Theorem 3 should involve only analysis of asymptotic behavior of functions, while Theorem 1 should involve only reasoning with functional data structures. In the end, Theorems 3 and 4 present an interface for external use, whose statements do not depend on details of the implementation or of the proofs.

We illustrate this strategy on the final step of the verification of merge sort. The definitions of the imperative and functional programs are shown side by side in Figure 5.1. Note that the former is working with a functional list, while the latter is working with an imperative array on the heap.

The running time function of the procedure is defined as follows:

$$
\begin{aligned}
&n \leq 1 \implies \mathit{merge\_sort}_{time}\ n = 2\\
&n > 1 \implies \mathit{merge\_sort}_{time}\ n = 2 + \mathit{atake}_{time}\ n\\
&\qquad\qquad + \mathit{adrop}_{time}\ n + \mathit{merge\_sort}_{time}\ (n\ \mathit{div}\ 2)\\
&\qquad\qquad + \mathit{merge\_sort}_{time}\ (n - n\ \mathit{div}\ 2) + \mathit{merge\_list}_{time}\ n
\end{aligned}
$$

The theorems to be proved are as follows. First, correctness of the functional algorithm $\mathit{merge\_sort}_{fun}$:

$$\mathit{merge\_sort}_{fun}\ xs = \mathit{sort}\ xs$$

Second, a Hoare triple asserting the agreement of the three definitions:

$$
\begin{aligned}
&<p \mapsto_a xs \star \$\,(\mathit{merge\_sort}_{time}\ |xs|)>\\
&\ \mathit{merge\_sort}_{impl}\ p
\end{aligned}
$$

$<\lambda\_.\ p \mapsto_a\ merge\_sort_{fun}\ xs>$

Third, the asymptotic time complexity of $merge\_sort_{time}$:

$merge\_sort_{time} \in \Theta(\lambda n.\ n \log n)$

Finally, Theorems 1 and 2 are combined to prove the final Hoare triple for external use, with $merge\_sort_{fun}\ xs$ replaced by $sort\ xs$.

5.2.3 Amortized analysis

In an amortized analysis, we fix some type of data structure and consider a set of primitive operations on it. For simplicity, we assume each operation has exactly one input and output data structure (extension to the general case is straightforward). A potential function Φ is defined on instances of the data structure and represents time credits that can be used for future operations. Each procedure f is associated its running time f_{t} and an advertised running time f_{at}. They are required to satisfy the following inequality: let a be the input data structure of f and let b be its output data structure, then[3]

$$f_{\mathrm{at}} + \Phi(a) \geq f_{\mathrm{t}} + \Phi(b). \tag{5.1}$$

The proof of inequality 5.1 usually involves arithmetic, and sometimes the correctness of the functional algorithm. For skew heaps and splay trees, the analogous results are already proved in [109]. Only slight modifications are necessary to bring them into the right form for our use.

The organization of an amortized analysis in our framework is as follows. We define two assertions: the *raw* assertion $raw_{assn}\ t\ a$ stating that the address a points to an imperative data structure refining t, and the *amortized* assertion, defined as

$$amor_{assn}\ t\ a = raw_{assn}\ t\ a \star \$\,(\Phi(t)),$$

where P is the potential function.

For each primitive operation implemented by f, we define f_{fun}, f_{impl}, and f_{time} as before, where f_{time} is the *actual* running time. We further define a function f_{atime} to be the proposed advertised running time. The theorems to be proved are as follows (compare to the list in Section 5.2.2):

1. The functional program f_{fun} satisfies the desired correctness property.
2. A Hoare triple using the amortized assertion stating that f_{impl} implements f_{fun} and runs within f_{atime}, which is a consequence of the following:
   - 2a. A Hoare triple using the raw assertion stating that f_{impl} implements f_{fun} and runs within f_{time}.
   - 2b. The inequality between amortized and actual running time.

[3] In many presentations, the amortized running time f_{at} is simply *defined* to be $f_{\mathrm{t}} + \Phi(b) - \Phi(a)$. Our approach is more flexible in allowing f_{at} to be defined by a simple formula and isolating the complexity to the proof of (5.1).

3. The *amortized* running time f_{atime} satisfies the desired asymptotic behavior.

4. Combining 1 and 2, a Hoare triple stating that f_{impl} satisfies the desired correctness property and runs within f_{atime}.

In the case of data structures (and unlike merge sort), it is useful to state Theorem 4 in terms of yet another, abstract assertion which hides the concrete reference to the data structure. This follows the technique described in [137, Section 5.3]. Theorems 3 and 4 are the final results for external use.

We now illustrate this strategy using splay trees as an example. The raw assertion is called *btree*. The basic operation in a splay tree is the "splay" operation, from which insertion and lookup can be easily defined. For this operation, the functions *splay*, $splay_{impl}$, and $splay_{time}$ are defined by recursion in a structurally similar manner. Theorem 2a takes the form:

$$<btree_{assn}\ t\ a \star \$\,(splay_{time}\ x\ t)>\ splay_{impl}\ x\ a\ <btree_{assn}\ (splay\ x\ t)>$$

Let Φ_{splay_tree} be the potential function on splay trees. Then the amortized assertion is defined as:

$$splay\_tree_{assn}\ t\ a = btree_{assn}\ t\ a \star \$\,(\Phi_{splay\_tree}\ t)$$

The advertised running time for splay has a relatively simple expression:

$$splay_{atime}\ n = 15 * (\lceil 3 * log\ 2\ n \rceil + 2)$$

The difficult part is showing the inequality relating actual and advertised running time (Theorem 2b):

$$\begin{aligned} &bst\ t \implies splay_{atime}\ (size1\ t) + \Phi_{splay\_tree}\ t \\ &\qquad \geq splay_{time}\ x\ t + \Phi_{splay\_tree}\ (splay\ x\ t), \end{aligned}$$

here *size1 t* is the of the tree t plus one. This claim follows from the corresponding lemma in [109]. Note the requirement that t is a binary search tree. Combining 2a and 2b, we get Theorem 2:

$$\begin{aligned} &bst\ t \implies \\ &\ <splay\_tree_{assn}\ t\ a \star \$\,(splay_{atime}\ (size1\ t))> \\ &\ \ splay_{impl}\ x\ a \\ &\ <splay\_tree_{assn}\ (splay\ x\ t)> \end{aligned}$$

The asymptotic bound on the advertised running time (Theorem 3) is:

$$splay_{atime} \in \Theta(\lambda x.\ \log x)$$

The functional correctness of *splay* (Theorem 1) states that it maintains sorting of the binary search tree and its set of elements:

$$\begin{aligned} &bst\ t \implies bst\ (splay\ a\ t), \\ &set\_tree\ (splay\ a\ t) = set\_tree\ t \end{aligned}$$

Here *set_tree t* maps tree t to the set of values stored in it. The following abstract assertion hides the concrete tree behind an existential quantifier:

$$splay\_tree\_set_{assn}\ S\ a = (\exists_A t.\ splay\_tree_{assn}\ t\ a \star \uparrow(bst\ t) \star \uparrow(set\_tree\ t = S))$$

Here, the operator $\exists_A$ is the existential quantifier lifted to assertions. The final Hoare triple takes the form ($|S|$ denotes the cardinality of S):

$<splay\_tree\_set_{assn}\ S\ a \star \$(splay_{atime}\ (|S|\ +\ 1))>$
$splay_{impl}\ x\ a$
$<\lambda r.\ splay\_tree\_set_{assn}\ S\ r>$

Now that we have seen how to organize the verification of an imperative program, we turn to how we can mechanize the proofs.

5.3 Reasoning Framework

In this section, I describe automation to handle proofs of the kind of Theorem 2. I will discuss automation for reasoning about Separation Logic with Time Credits. Our extension of Lammich's [84, §2.5] automatic method "sep_auto" for Separation Logic will be covered at the end of this section. First, I present an extension of the setup discussed in [137] for reasoning about ordinary Separation Logic. Here, I focus on the additional setup concerning time credits.

The proof of a Hoare triple for program proceeds by symbolically executing that program while maintaining a symbolic heap. In the beginning that heap is the precondition of the Hoare triple to be proven. Any basic step in the proof is as follows: suppose the current heap satisfies the assertion $P \star \$T$ and the next command has the Hoare triple

$<P' \star \$T' \star \uparrow b>\ c\ <Q>$

where b is the pure part of the precondition, apply the Hoare triple to derive the successful execution of c, and some assertion on the next heap. In ordinary Separation Logic (without $\$T$ and $\$T'$), this involves matching P' with parts of P, proving the pure assertions b, and then applying the frame rule. In the current case, we additionally need to show that $T' \leq T$, so $\$T$ can be rewritten as $\$T = \$(T' + T'') = \$T' \star \$T''$.

In general, proving this inequality can involve arbitrarily complex arguments. However, due to the close correspondence in the definitions of f_{time} and f_{impl}, the actual tasks usually lie in a simple case, and we tailor the automation to focus on this case. First, we normalize both T and T' into polynomial form:

$$T = c_1 p_1 + \cdots + c_m p_m, \quad T' = d_1 q_1 + \cdots + d_n q_n, \tag{5.2}$$

where each c_i and d_j are constants, and each p_i and q_j are non-constant terms or 1. Next, for each term $d_j q_j$ in T', we try to find some term $c_i p_i$ in T such that p_i equals q_j according to the known equalities, and $d_j \leq c_i$. If such a term is found, we subtract $d_j p_i$ from T. This procedure is performed on T in sequence (so $d_2 q_2$ is searched on the remainder of T after subtracting $d_1 q_1$, etc.). If the procedure succeeds with T'' remaining, then we have $T = T' + T''$.

The above procedure suffices in most cases. For example, given the parallel definitions of $merge\_sort_{impl}$ and $merge\_sort_{time}$ in Section 5.2.2, it is able to show that $merge\_sort_{impl}$ runs in time $merge\_sort_{time}$. However, in some special cases, more is needed. The extra reasoning often takes the following form: if s is a term in the normalized form of T, and $s \geq t$ holds for some t (an inequality that must be derived during the proof), then the term s can be replaced by t in T.

In general, we permit the user to provide hints of the form

@have $s \geq_t t$,

where the operator $\cdot \geq_t \cdot$ is equivalent to $\cdot \geq \cdot$, used only to remind *auto2* [137] that the fact is for modification of time credit only. Given this instruction, *auto2* attempts to prove $s \geq t$, and when it succeeds, it replaces the assertion $h_i \models P \star \$T$ on the current heap with $h_i \models P \star \$T' \star true$, where the new time credit T' is the normalized form of $T - s + t$. This technique is needed in case studies such as binary search and median of medians selection (see the explanation for the latter in Section 5.5).

Besides *auto2* there exists another tactic for handling Separation Logic and proving Hoare triples in Imperative-HOL: Lammich [84, §2.5] provides *sep_auto* — a strong automation for vanilla Imperative-HOL — which we extend by the above mentioned time frame inference routine to also handle programs in the time-aware case. Essentially it first collects all time credits in P and P'. Then it applies the normal frame inference algorithm on the parts that do not involve time credits. This matching typically instantiates all free variables in P'. The matching of time credits then is implemented by the component of *auto2* described above.

By fixing the form of specifications (as described in Section 5.2.1) they can be used as an interface. It does not matter how we established the Hoare triples of some program c — be it *auto2*, *sep_auto* or any other method — we can always register those facts with the respective automation in order to make it available for future calls of the tactics for programs that use c as a subroutine. This is made possible by strictly restricting the form of those rules. The same applies to claims of asymptotic complexity, for which we will describe automation in the next section.

5.4 Reasoning About Asymptotic Time Complexity

Working with asymptotic complexity informally can be particularly error-prone, especially when several variables are involved. Some examples of fallacious reasoning are given in [41, Section 2]. In an interactive theorem proving environment, such problems can be avoided, since all notions are defined precisely, and all steps of reasoning must be formally justified. In the following we will first present how asymptotic analysis is formalized in Isabelle. Then quickly sketch Eberl's formalization of a generalization of the Master Theorem which we use to handle recurrences that stem from divide-and-conquer algorithms. Finally, we present our automatic setup to solve proofs of the kind of Theorem 3 of our methodology.

5.4.1 Landau Symbols

For the definition of the big-O notation, or more generally Landau symbols, we use the formalization by Eberl [31], where they are defined in a general form in terms of filters, and therefore work also in the case of multiple variables.

In our work, we are primarily interested in functions of type $nat \to real$ (for the single variable case) and $nat \times nat \to real$ (for the two-variable case). Given a function g of one of these types, the Landau symbols $O(g), \Omega(g)$ and $\Theta(g)$ are sets of functions of the same type. In the single variable case, using the standard filter (*at_top* for *limit at positive infinity*), the definitions are as follows:

$$\begin{aligned}
f \in O(g) &\longleftrightarrow \exists c > 0.\ \exists N.\ \forall n \geq N.\ |f(n)| \leq c \cdot |g(n)| \\
f \in \Omega(g) &\longleftrightarrow \exists c > 0.\ \exists N.\ \forall n \geq N.\ |f(n)| \geq c \cdot |g(n)| \\
f \in \Theta(g) &\longleftrightarrow f \in O(g) \wedge f \in \Omega(g)
\end{aligned}$$

In the two-variable case, we will use the product filter $at\_top \times_F at\_top$ throughout. Expanding the definitions, the meaning of the Landau symbols are as expected:

$$\begin{aligned}
f \in O_2(g) &\longleftrightarrow \exists c > 0.\ \exists N.\ \forall n, m \geq N.\ |f(n,m)| \leq c \cdot |g(n,m)| \\
f \in \Omega_2(g) &\longleftrightarrow \exists c > 0.\ \exists N.\ \forall n, m \geq N.\ |f(n,m)| \geq c \cdot |g(n,m)| \\
f \in \Theta_2(g) &\longleftrightarrow f \in O_2(g) \wedge f \in \Omega_2(g)
\end{aligned}$$

5.4.2 Akra–Bazzi Theorem

A well-known technique for analyzing the asymptotic time complexity of divide-and-conquer algorithms is the Master Theorem (see for example [24, Chapter 4]). The Akra–Bazzi theorem is a generalization of the Master Theorem to a wider range of recurrences. Eberl [32] formalized the Akra–Bazzi theorem in Isabelle/HOL, and also wrote tactics for applying this theorem in a semi-automatic manner. Notably, the automation is able to deal with taking ceiling and floor in recursive calls, an essential ingredient for actual applications but often ignored in informal presentations of the Master theorem.

In this section, we state a slightly simpler version of the result that is sufficient for our applications. Let $f : \mathbb{N} \to \mathbb{R}$ be a non-negative function defined recursively as follows:

$$f(x) = g(x) + \sum_{i=1}^{k} a_i \cdot f(h_i(x)) \qquad \text{for all } x \geq x_0 \tag{5.3}$$

where $x_0 \in \mathbb{N}$, $g(x) \geq 0$ for all $x \geq x_0$, $a_i \geq 0$ and each $h_i(x) \in \mathbb{N}$ is either $\lceil b_i \cdot x \rceil$ or $\lfloor b_i \cdot x \rfloor$ with $0 < b_i < 1$, and x_0 is large enough that $h_i(x) < x$ for all $x \geq x_0$.

The parameters a_i and b_i determine a single characteristic value p, defined as the solution to the equation

$$\sum_{i=1}^{k} a_i \cdot b_i^p = 1 \tag{5.4}$$

Depending on the relation between the asymptotic behavior of g and $\Theta(x^p)$, there are three main cases of the Akra–Bazzi theorem:

Bottom-heavy: if $g \in O(x^q)$ for $q < p$ and $f(x) > 0$ for sufficiently large x, then $f \in \Theta(x^p)$.

Balanced: if $g \in \Theta(x^p \ln^a x)$ with $a \geq 0$, then $f \in \Theta(x^p \ln^{a+1} x)$.

Top-heavy: if $g \in \Theta(x^q)$ for $q > p$, then $f \in \Theta(x^q)$.

All three cases are demonstrated in our examples (in Karatsuba's algorithm, merge sort, and median of medians selection, respectively).

5.4.3 Automating Complexity Analysis

We now present our automation setup for the analysis of asymptotic behavior of running time functions. Eberl [31] already provides automation for Landau symbols in the single variable case. In addition to incorporating it into our framework, we add facilities for dealing with function composition and the two-variable case.

Because side conditions for the Akra–Bazzi theorem are in the Θ form, we mainly deal with Θ and Θ_2, stating the exact asymptotic behaviors of running time functions. However, since running time functions themselves are very often only upper bounds of the actual running times, we are essentially still proving big-O bounds on running times of programs.

In our case, the general problem is as follows: given the definition of $f_{time}(n)$ in terms of some $g_{time}(s(n))$ (running time of procedures called by f_{impl}), simple terms like $4n$ or 1, or recursive calls to f_{time}, determine the asymptotic behavior of f_{time}.

To begin with, we maintain a table of the asymptotic behavior of previously defined running time functions. The attribute *asym_bound* adds a new theorem to this table. This table can be looked-up by the name of the procedure.

We restrict ourselves to asymptotic bounds of the form

$$\text{polylog}(a, b) = (\lambda n.\ n^a (\log n)^b),$$

where a and b are natural numbers. In the two-variable case, we work with asymptotic bounds of the form

$$\text{polylog}_2(a, b, c, d) = (\lambda(m, n).\ \text{polylog}(a, b)(m) \cdot \text{polylog}(c, d)(n)).$$

This suffices for our present purposes and can be extended in the future. Note that this restriction does not mean our framework cannot handle other complexity classes, only that they will require more manual proofs (or further setup of automation).

Non-recursive case When the running time function is non-recursive, the analysis proceeds by determining the asymptotic behavior in a bottom-up manner.

To handle terms of the form $g_{time}(s(n))$ where s is linear, we use the following composition rule: if $u \in \Theta(\text{polylog}(a, b))$, and $v \in \Theta(\lambda n.\ n)$, then $u \circ v \in \Theta(\text{polylog}(a, b))$.

Composition in general is quite subtle: the analogous rule does not hold if u is the exponential function.[4]

The asymptotic behavior of a sum is determined by the absorption rule: if $g_1 \in O(g_2)$, then $\Theta(g_1 + g_2) = \Theta(g_2)$. Here, we make use of existing automation in [31] for deciding inclusion of big-O classes of polylog functions. The rule for products is straightforward.

The combination of these three rules can solve many examples automatically. E.g. this (artificial) example: if $f_1 \in \Theta(\lambda n.\ n)$ and $f_2 \in \Theta(\lambda n.\ \log n)$, then

$$(\lambda n.\ f_1(n+1) + n \cdot f_2(2n) + 3n \cdot f_2(n\ \mathit{div}\ 3)) \in \Theta(\lambda n.\ n \log n).$$

Analogous results are proved in the two-variable case (note that unlike in the single variable case, not all pairs of polylog$_2$ functions are comparable. e.g. $O(m^2 n + mn^2)$). For example, the following can be automatically solved: if additionally $f_3 \in \Theta(\lambda(m,n).\ mn)$ and $f_4 \in \Theta(\lambda(m,n).\ m+n)$, then

$$(\lambda(m,n).\ f_1(n) + f_2(m) + mn + f_3(m \text{ div } 3, n+1)) \in \Theta(\lambda(m,n).\ mn).$$
$$(\lambda(m,n).\ 1 + f_1(n) + f_2(m) + f_4(m+1, n+1)) \in \Theta(\lambda(m,n).\ m+n).$$

Recursive case There are two main classes of results for analysis of recursively-defined running time functions: the Akra–Bazzi theorem and results about linear recurrences. For both classes of results, applying the theorem reduces the analysis of a recursive running time function to the analysis of a non-recursive function, which can be solved using automation described in the previous part.

The Akra–Bazzi theorem is discussed in Section 5.4.2. Theorems about linear recurrences allow us to reason about for-loops written as recursions. They include the following: in the single variable case, if f is defined by recursion as

$$f(0) = c, \quad f(n+1) = f(n) + g(n),$$

where $g \in \Theta(\lambda n.\ n)$, then $f \in \Theta(\lambda n.\ n^2)$.

In the two-variable case, if f satisfies

$$f(0, m) \le C, \quad f(n+1, m) = f(n, m) + g(m)$$

where $g \in \Theta(\lambda n.\ n)$, then $f \in \Theta_2(\lambda(n,m).\ nm)$.

Example 5.4.1. As an example, consider the problem of showing $\Theta(\lambda n.\ n \log n)$ complexity of $\mathit{merge\_sort}_{\mathit{time}}$, defined in Section 5.2.2. This applies the balanced case of the Akra–Bazzi theorem. Using this theorem, the goal is reduced to:

$$(\lambda n.\ 2 + \mathit{atake}_{\mathit{time}}\ n + \mathit{adrop}_{\mathit{time}}\ n + \mathit{merge\_list}_{\mathit{time}}\ n) \in \Theta(\lambda n.\ n)$$

(the non-recursive calls run in linear time). This can be shown automatically using the method described in the previous section, given that $\mathit{atake}_{\mathit{time}}$, $\mathit{adrop}_{\mathit{time}}$, and $\mathit{merge\_list}_{\mathit{time}}$ have already been shown to be linear.

[4] https://math.stackexchange.com/questions/761006/big-o-and-function-composition

5.5 Case Studies

In this section, I first present the main case studies verified using our framework in the original paper [138]. They focus on showing the applicability of our approach and the coverage of our automation. After that, I briefly sketch two larger developments conducted by students under my supervision using the framework: Löwenberg [98] ported the running time analysis proof of the union-find data structure by Charguéraud and Pottier [21] from Coq to Isabelle/HOL. Furthermore, there are two verifications projects [126, 39] that work towards the verification of Fibonacci heaps [24].

5.5.1 Gallery of Use Cases

Our first set of examples can be divided into three classes: divide-and-conquer algorithms (using the Akra–Bazzi theorem), algorithms that are essentially for-loops (using linear recurrences), and amortized analysis.

For all the case studies in this section we mainly used *auto2* for proving Hoare Triples involving Separation Logic with Time Credits. We measure the complexity of a proof by counting the number of steps in the proof: each lemma statement counts as one step and each hint provided by the user as an additional step. In the table below, #Hoare counts the number of steps for proving the Hoare triples (Theorems 2 and 4). #Time counts the number of steps for reasoning about running time functions (Theorem 3). We also list the ratio (Ratio) between the sum of #Hoare and #Time to the number of lines of the imperative program (#Imp). This ratio measures the overhead for verifying the imperative program with running time analysis. In particular this does *not* include verifying the correctness of the functional program (Theorem 1). In addition we list the total lines of code for each case study.

	#Imp	#Time	#Hoare	Ratio	LOC
Binary search	11	10	14	2.18	82
Merge sort	38	11	12	0.61	121
Karatsuba	58	18	28	0.79	250
Select	51	41	31	1.41	447
Insertion sort	15	3	4	0.47	42
Knapsack	27	9	8	0.63	113
Dynamic array	55	19	37	1.02	424
Skew heap	25	38	21	2.36	257
Splay tree	120	51	37	0.73	447
Red-black tree	270	51	44	0.35	891

Using our automation the average overhead ratio is slightly over 1. On a dual-core laptop with 2GHz each, processing all the examples takes around ten minutes. The development of the case studies, together with the framework itself, took about 4 person months.

Next we give details for some of the case studies.

Karatsuba's algorithm The functional version of Karatsuba's algorithm for multiplying two polynomials is verified in [27]. To simplify matters, we further restrict us to the case where the two polynomials are of the same degree.

The recursive equation is given by:

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + g(n). \tag{5.5}$$

Here $g(n)$ is the sum of the running times corresponding to non-recursive calls, which can be automatically shown to be linear in n. Then the Akra–Bazzi method gives the solution $T(n) \in \Theta(n^{log_2 3})$ (bottom-heavy case).

Median of medians selection Median of medians for quickselect is a worst-case linear-time algorithm for selecting the i-th largest element of an unsorted array [24, Section 9.3]. In the first step of the algorithm, it chooses an approximate median p by dividing the array into groups of 5 elements, finding the median of each group, and finding the median of the medians by a recursive call. In the second step, p is used as a pivot to partition the array, and depending on i and the size of the partitions, a recursive call may be made to either the section $x < p$ or the section $x > p$. This algorithm is particularly interesting because its running time satisfies a special recursive formula:

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + g(n), \tag{5.6}$$

where $g(n)$ is linear in n. The Akra–Bazzi theorem shows that T is linear (top-heavy case).

Eberl verified the correctness of the functional algorithm [33]. There is one special difficulty in verifying the imperative algorithm: the length of the array in the second recursive call is not known in advance, only that it is bounded by $\lceil 7n/10 \rceil$. Hence, we need to prove monotonicity of T, as well as provide the hint $T(\lceil 7n/10 \rceil) \geq_t T(l)$ (where l is the length of the array in the recursive call) during the proof.

Knapsack The dynamic programming algorithm solving the Knapsack problem is used to test our ability to handle asymptotic complexity with two variables. The time complexity of the algorithm is $\Theta_2(nW)$, where n is the number of items, and W is the capacity of the sack. Correctness of the functional algorithm was proved by Simon Wimmer.

Skew heap and splay tree For these two examples, the bulk of the analysis (functional correctness and justification of amortized runnig time) is done by Nipkow [109]. Our work is primarily to define the imperative version of the algorithm and verifying its agreement with the functional version. Some work is also needed to transform the results in [109] into the appropriate form, in particular rounding the real-valued potentials and running time functions into natural numbers required in our framework.

Red-black tree Zhan formalized red-black trees in Imperative-HOL and proved their functional correctness [137, §6.2]. We can plainly reuse the imperative code as well as the verification of the functional abstraction. In contrast to counting the number of "steps" for the whole verification (as in [137, §6]) we count the steps for reasoning about time (#Time) and Hoare triples (#Hoare) but not the functional correctness. The overhead of adding the reasoning about running time was minimal. Most of the proofs were automatic and we essentially only had to massage an upper bound on the maximum depth of the red-black tree into the right form. This data structure was used as a set implementation in the breadth-first search component of the case study verifying the Edmonds–Karp algorithm for network flow [45].

Dynamic array Dynamic Arrays [24, Section 17.4] are one of the simpler amortized data structures. We have verified the version that doubles the size of the array whenever it is full (without automatically shrinking the array). In the following I will describe the amortized analysis of dynamic arrays in Imperative-HOL-Time in more detail. Later (in Section 10.2), I will show how a similar verification can be executed on a higher level of abstraction and I will use the machinery from Chapter 9 to synthesize an implementation for dynamic arrays.

Example 5.5.1. In this example I describe how we can prove dynamic arrays in Imperative-HOL-Time. This illustrates how amortized data structures can be verified using the framework.

An *abstract dynamic list* is represented by a pair of a carrier list bs and a fill level n. The corresponding abstract list as is the list bs restricted to the first n elements:

$$dyn\_abs\ (bs,\ n)\ as \longleftrightarrow as = take\ n\ bs \wedge n < |bs|$$

We define a function $push\_array_{fun}$ on abstract dynamic lists that doubles the length of the list if it is full and then appends an element. We prove its functional correctness:

$$dyn\_abs\ (bs,\ n)\ as \implies dyn\_abs\ (push\_array_{fun}\ x\ (bs,\ n))\ (as \cdot [x])$$

Recall that $p \mapsto_a xs$ denotes a heap containing an array at address p with content xs. Based on this, one can define an assertion

$$dyn\_array\_raw_{assn}\ (bs,\ n)\ (p,\ m) = (p \mapsto_a bs \star \uparrow(m = n))$$

relating an abstract dynamic list with a concrete *dynamic array* represented by a pair of address p and fill level m.

For the functional $push\_array_{fun}$ we define a corresponding procedure $push\_array_{impl}$ which appends an element to the back of a dynamic array, doubling the length if it is exceeded. We can now show the following raw Hoare triple, with worst-case running time linear in the fill level of the dynamic array, as we might have to double the array. The explicit numbers in the running time stem from the concrete implementation of $push\_array_{impl}$ and the cost model of time-aware Imperative-HOL.

$$n \leq |bs| \implies$$
$$<dyn\_array\_raw_{assn}\ (bs,\ n)\ p \star \$(5 * n + 9)>$$

$$\begin{array}{l} \mathit{push\_array}_{impl}\ x\ p \\ <\lambda p'.\ \mathit{dyn\_array\_raw}_{assn}\ (\mathit{push\_array}_{fun}\ x\ (bs,\ n))\ p'> \end{array}$$

We now incorporate the potential ($\Phi(bs,\ n) = 10 * n - 5 * |bs|$) and the invariant $n \leq |bs|$ into an assertion for a compound data structure *dyn_array* and prove the following Hoare triple with amortized constant running time:

$$\begin{array}{l} \mathit{dyn\_array}_{assn}\ (bs,\ n)\ p = \mathit{dyn\_array\_raw}_{assn}\ (bs,\ n)\ p \\ \qquad\qquad\qquad\qquad\qquad\qquad \star\ \$(\Phi\ (bs,\ n)) \star \uparrow(n \leq |bs|) \end{array}$$

$$\begin{array}{l} <\mathit{dyn\_array}_{assn}\ (bs,\ n)\ p \star \$19> \\ \quad \mathit{push\_array}_{impl}\ x\ p \\ <\lambda p'.\ \mathit{dyn\_array}_{assn}\ (\mathit{push\_array}_{fun}\ x\ (bs,\ n))\ p'> \end{array}$$

Note that for showing the latter amortized Hoare triple it does not suffice to employ the raw Hoare triple, rather *push_array* must be unfolded again.

As a final step we compose the refinements of abstract lists to abstract dynamic lists (*dyn_abs*) and further to dynamic arrays ($\mathit{dyn\_array}_{assn}$) and obtain the dyna_{assn}:

$$\mathit{dyna}_{assn}\ as\ p = (\exists_A bs\ n.\ \mathit{dyn\_array}_{assn}\ (bs,\ n)\ p \star \uparrow(\mathit{dyn\_abs}\ (bs,\ n)\ as))$$

where the list and fill level of the abstract dynamic array are hidden behind an existential quantifier. An assertion like $\mathit{dyna\_assn}_{assn}$ that relates a memory location with a logical representation of the data structure is called a *representation predicate*.

Then we obtain the final Hoare triple of the procedure:

$$<\mathit{dyna}_{assn}\ as\ p \star \$19>\ \mathit{push\_array}_{impl}\ x\ p\ <\lambda p'.\ \mathit{dyna}_{assn}\ (as \cdot [x])\ p'>$$

The Hoare triple serves as an interface to be used in the verification of larger programs. Note, that the user of this Hoare triple just sees a constant time operation and the amortization is conveniently hidden using the time credits in the representation predicate.

The reasoning about dynamic arrays is quite independent from how the underlying array is implemented, or rather which program semantics we use. In Section 10.2, I will show how we can abstractly model dynamic arrays and their amortized running time complexity, while still being able to synthesize a concrete implementation from it.

5.5.2 Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation

Charguéraud and Pottier [21] verified the $O(\alpha(n))$ amortized running time of an efficient implementation of the union-find data structure with union-by-rank and path compression in Coq. Here, the function α is the very slowly growing *inverse Ackermann function*. Löwenberg [98] ported that work to Isabelle/HOL and proved the same result for an implementation in Imperative-HOL-Time. He based his work on an earlier implementation by Lammich, for which I [45] proved the worst-case execution time $O(\log n)$. In the process he uncovered a running time bug. The original implementation in Imperative-HOL did not perform path compression on *union*.

Here, I state the final Hoare triples for the operations *init*, *compare* and *union* of the union find data structure:

$$<\$\, uf\_init_{time}\ n>\ uf\_init_{impl}\ n\ <\lambda r.\ is\_uf_{assn}\ r\ \{(v,\ v)\mid v \leq n\}>$$
$$uf\_init_{time} \in \Theta(n)$$

$$<is\_uf_{assn}\ u\ R\ \$\, uf\_cmp_{time}\ |Dom\ R|>$$
$$\quad uf\_cmp_{impl}\ u\ i\ j$$
$$<\lambda r.\ is\_uf_{assn}\ u\ R \star \uparrow(r \longleftrightarrow (i,\ j) \in R)>$$
$$uf\_cmp_{time} \in \Theta(\alpha(n))$$

$$<is\_uf_{assn}\ u\ R\ \$\, uf\_union_{time}\ |Dom\ R|>$$
$$\quad uf\_union_{impl}\ u\ i\ j$$
$$<\lambda r.\ is\_uf_{assn}\ r\ (per\_union\ R\ i\ j)>$$
$$uf\_union_{time} \in \Theta(\alpha(n))$$

Here, $per\_union\ R\ i\ j$ is the relation obtained when the equivalence classes of the elements i and j in the partial equivalence relation R are joined, and $is\_uf_{assn}\ r\ R$ is the representation predicate for a union-find data structure representing relation R at address r. Note that this assertion contains time credits for amortization.

In Section 10.1 I will use the union-find data structure in Kruskal's algorithm.

5.5.3 Verification of Fibonacci Heaps

A Fibonacci heap is a data structure for priority queue operations that in particular supports an amortized $\Theta(1)$ *decrease-key* operation, and thus is crucial for optimal asymptotic running time of important algorithms, such as Dijkstra's shortest path algorithm. Verifying functional correctness and the running time analysis of Fibonacci heaps seems to be at the limit of current techniques. I will quickly describe two projects in Imperative-HOL and Imperative-HOL-Time that worked towards that goal, and a verification of an extension of Separation Logic that I believe can simplify future verification attempts. To the best of my knowledge there is no verification of Fibonacci heaps in a theorem prover besides that.

First, Stüwe [98] explored the limits of inductive definitions of data structures in Separation Logic and our ad hoc refinement methodology. He modeled Fibonacci heaps without parent pointers as a functional data structure, verified all priority queue operations but *decrease-key* and *delete*, and provided imperative refinements in Imperative-HOL-Time. This project includes both the functional correctness proof and the running time analysis. Note that the *delete-min* operation is already quite involved and has amortized running time $\Theta(\log n)$. However, this approach seems to not work for the verification of the *decrease-key* and *delete* operation. Their running time relies on additional pointers into the tree structure and parent pointers for every node. Those can not be modeled with an inductive predicate in Separation Logic easily.

Griebel [39] specifically focused on verifying the functional correctness of the *decrease-key* operation.[5] He models a Fibonacci heap as an unstructured monolithic data structure in Separation Logic and describes its structure with a mathematical graph. Unfortunately this inhibits the local reasoning Separation Logic normally allows. Regardless, he succeeds in proving functional correctness for the *decrease-key* operation.

The Flow framework by Krishna et al. [76] is an extension of Separation Logic that allows to cut out an arbitrary portion of the heap and reinserting a modified portion of the heap as long as it fulfills some interface. I believe that this technique can be used to conveniently model the cutting of trees from the Fibonacci heap and allows local reasoning instead of reasoning on the graph of the whole data structure. Pöttinger [121] verified the theory of the Flow framework in Isabelle/HOL which now can be used to verify overlaid data structures like Fibonacci heaps.

Adding the running time analysis for *decrease-key* and combining the two verification projects to obtain a verification of all operations for Fibonacci heaps in Imperative-HOL-Time is obvious future work.

5.6 Related Work

We compare our work with recent advances in verification of running time analysis of programs, starting from those based on interactive theorem provers to the more automatic methods.

The most closely-related is the impressive work by Guéneau et al. [41, 42, 40] for asymptotic time complexity analysis in Coq. We now take a closer look at the similarities and differences:

- Guéneau et al. give a structured overview of different problems that arise when working informally with asymptotic complexity in several variables, then solve this problem by rigorously defining asymptotic domination (which is essentially $f \in O(g)$) with filters and develop automation for reasoning about it. We follow the same idea by building on existing formalization of Landau symbols with filters in Isabelle [31], then extend automation to also handle the two-variable case.

- While they package up the functional correctness together with the complexity claims into one predicate *specO*, we choose to have two separate theorems (the Hoare triple and the asymptotic bound).

- While their automation assists in synthesizing recurrence equations from programs, they leave their solution to the human. In contrast, we write the recurrence relation by hand, which can be highly non-obvious (e.g. in the case of median of medians selection), but focus on solving the recurrences for the asymptotic bounds automatically (e.g. using the Akra–Bazzi theorem).

- Their main examples include binary search, the Bellman–Ford algorithm, union-find [22] and the incremental cycle detection algorithm [42]. While the latter

[5] The *delete* operation can be directly implemented with *decrease-key*.

one is certainly the largest and most involved algorithm verified with Separation Logic and Time Credits, none of those examples requires applications of the Master theorem or the Akra–Bazzi method. We present several other advanced examples, including applications of the Akra–Bazzi method, and those involving amortized analysis.

- In [42] Guéneau et al. present integer time credits and how they can be used to simply the reasoning setup.

Mevel et al. [103] present the dual to time credits: time receipts. They can be used to prove lower bounds on the running time and the absence of integer overflows. It is not entirely clear to me how time receipts relate to negative integer time credits.

Wang et al. [130] present TiML, a functional programming language which can be annotated by invariants and specifically also with time complexity annotations in types. The type checker extracts verification conditions from these programs, which are handled by an SMT solver. They also make the observation that annotational burden can be lowered by not providing a closed form for a time bound, but only specifying its asymptotic behavior. For recursive functions, the generated VCs include a recurrence (e.g. $T(n-1)+4n \leq T(n)$) and one is left to show that there exists a solution for T which is additionally in some asymptotic bound, e.g. $O(n^2)$. By employing a recurrence solver based on heuristic pattern matching they make use of the Master Theorem in order to discharge such VCs. In that manner they are able to verify the asymptotic complexity of merge sort. Additionally they can handle amortized complexity, giving Dynamic Arrays and Functional Queues as examples. Several parts of their work rely on non-verified components, including the use of SMT solvers and the pattern matching for recurrence relations. In contrast, our work is verified throughout by Isabelle's kernel.

On the other end of the scale we want to mention Automatic Amortized Resource Analysis (AARA). Possibly the first example of a resource analysis logic based on potentials is due to Hofmann and Jost [60]. They pioneer the use of potentials coded into the type system in order to automatically extract bounds in the running time of functional programs. Hoffmann et al. successfully developed this idea further [57, 58, 116, 65, 119]. Carbonneaux et al. [18, 17] extend this work to imperative programs and automatically solve extracted inequalities by efficient off-the-shelf LP-solvers. While the potentials involved are restricted to a specific shape, the analysis performs well and at the same time generates Coq proof objects certifying their resulting bounds. Kavvos [67] present a method that automatically extracts recurrences from recursive functional programs, which represent the running time in terms of the size of the input. They do not handle amortization.

5.7 Recap, Limitations, and Future Work

In this chapter, I have presented a framework for verifying asymptotic time complexity of imperative programs. This is done by extending Imperative-HOL and its Separation

Logic with Time Credits. Through the case studies, I have demonstrated the ability of our framework to handle complex examples, including those involving advanced techniques of time complexity analysis, such as the Akra–Bazzi theorem and amortized analysis. I also showed that verification of amortized analysis of functional programs [109] can be converted to verification of imperative programs with little additional effort. The framework is mature enough that students with prior Isabelle knowledge can use it to perform interesting case studies with it. However, the framework has some weaknesses and limitations. In the following I will comment on them and suggest ideas for future work.

When presenting the definition of Hoare triples in Section 5.1 I have used an abstraction *abs* that maps heaps to abstract heaps which form a separation algebra. I have to admit that this was a simplification for the sake of readability, that is not present in the actual formalization. The first iteration of Lammich's as well as Zhan's Separation Logic for Imperative-HOL [85, 137] actually do not exploit the fact that the heap can be mapped to an abstract heap which forms a separation algebra. Doing so would allow to use the theory on separation algebras by Klein et al. [69] and, thus, to define the Separation Logic from algebraic principles. Instead, they both prove the basic reasoning rules for the concrete instance using a concrete heap representation. As the Separation Logic with Time Credits of Imperative-HOL-Time extends Zhan's work [137], we did the same here. The cleaner approach, however, is to use the abstraction *abs* into a separation algebra. Lammich developed a generic *wp* framework and applied it to his shallowly embedded LLVM semantics [81]. I will present that framework in Section 6.1 and I will show how it is extended with time credits. With the abstraction *abs* one can instantiate that generic framework also for Imperative-HOL (and Imperative-HOL-Time), and obtain an assertion language and Hoare triples as well as generic verification infrastructure. Unfortunately, this will not directly be compatible with the original formalizations, while they morally are isomorphic. As the difference is only at the intermediate level between concrete heaps and assertions, only minimal changes need to be applied until the interface — in form of properties of the separating connectives and the basic Hoare rules — is established. Thus, we have an engineering problem here. I think, the cleaner solution should be used in future developments, but older developments need to be ported in order to have all the algorithm formalized in the same framework.

A major limitation of the framework is its simplistic cost model that essentially only counts operations that are concerned with arrays and references. Because Imperative-HOL-Time is shallowly embedded, costly operations can be put into a functional implementation and then integrated with the `return` operator into the monadic setting. Also, the code generator will not complain about that during code extraction. Hence, strictly speaking, it does not suffice to only look at the proven Hoare triples but also the programs have to be inspected to ensure that the correct operations are counted. Restricting the language to a flattened form can be asserted by an automatic tool. The second iteration — the shallow LLVM semantics with cost model (Chapter 6) — provides such a tool: the LLVM code generator will only accept programs in monadified form using a set of allowed basic operations. Furthermore, in contrast to the cost

model of Imperative-HOL-Time for LLVM we will count the number of calls of LLVM instructions separately in order to allow for a more fine-grained analysis. Charguéraud and Pottier [21, Section 2.7] discuss different levels at which the time complexity of a program can be measured.

The time credits we use are of type *nat*. Using extended natural numbers instead has the advantage that Hoare triples from vanilla Imperative-HOL could be integrated. Adding infinitely many time credits ($\$\,\infty$) in the precondition would express that the program terminates but the running time bound is unknown. That is exactly what Imperative-HOL Hoare triples mean. The running time analysis can be provided separately — by proving $<\$\ T \star P>\ c\ <\lambda\_.\ true>$ — and be combined afterwards. It would be interesting how to automatically obtain rough running time bounds from existing formalizations which already provide termination proofs. Using integer time credits [42] avoids performing frame inference of time credits at every step. Instead, it only needs to be performed once at the end when proving that the final symbolic heap entails the post condition. Equipping Imperative-HOL-Time with integer time credits would simplify that component of the proofs. Maybe a similar result could be achieved when using *separating coimplication* as described by Bannister et al. [6, Section 6] for handling time credits.

Wimmer et al. [134] present a framework for the automatic verified memoization of recursive functions. For a recursive function f their tool is able to construct an imperative version (in Imperative-HOL) which avoids recalculating results by storing them in a lookup table. Dynamic programming algorithms can be conveniently verified with that approach. With an amortized analysis that adds potential containing time credits for each cell of the lookup table that has not yet been calculated, one could determine the total running time of a calculation from the costs of one cell given that all recursive results already are present in the lookup table. Extending their work to automatically construct a program in Imperative-HOL-Time along with a synthesized running time bound would be an interesting piece of work. Here, I briefly sketch how amortization would play into it. The assertion for the lookup table should contain time credits for the results that have not yet been calculated.

$$amor\_lookuptable_{assn}\ M\ A\ p = lookuptable_{assn}\ M\ p \star \$\,(\textstyle\sum_{a \in A - dom\ M}.\ cost\ a)$$

where $lookuptable_{assn}\ M\ p$ is a data structure representing a partial map M from parameters to results of the memoized function called with those parameters (i. e. potentially mapping a to $f\ a$), A being a superset of the interesting instances of parameters (e. g. the parameters of recursive calls of a computation of $f\ x$ for a certain x), and $cost\ a$ being the cost of determining $f\ a$ given the results of all recursive calls are already available. A computation of the value of $f\ a$ with such an assertion in the precondition amounts to looking up whether the result is already in the lookup table and returning it if it is, or recursively calculating the result and storing it in the lookup table afterward. In both cases the running time is the advertised cost of lookup in the data structure, that might be $\Theta(1)$ for an array. The accounting of the total cost is moved into initializing the amortized lookup table. By providing a tight set A the analysis can be optimized.

Both top-down (just calculating $f\ a$), or bottom-up (first calculating $f\ a$ for all $a \in A$ and then $f\ a$) can be modeled that way.

Our automation for asymptotic analysis works fine in our applications but is limited in several ways. As that component can not only be applied in our context (we will also use the component in Part III) it might be worthwhile to develop it further. First of all, we only support the one and two variable case, but I believe that there are only technical obstacles to extending it to a general multivariate case. Second, we only allow polynomials and logarithms (*polylog*) as basic functions for the asymptotic bounds. In particular, we can not automatically support α. It requires more engineering to extend our approach to more basic functions and even make it dynamically extendable. Finally, the automation could be extended to take advantage of the general theory of Landau symbols using filters.

Code extracted from Imperative-HOL algorithms typically is one order of magnitude slower (e. g. [84, §6]) compared to standard implementations in purely imperative languages like C++ or LLVM. Lammich [81] provides a shallow embedding of LLVM in Isabelle/HOL and is able to extract LLVM code whose performance is comparable with unverified reference implementations. We extend it with time credits in Chapter 6.

We only cover upper bounds on the running time. Studying lower bounds, as well as other resources such as heap space or stack space consumption are interesting topics.

5.8 Summary

- We have extended Imperative-HOL with a cost semantics, and presented a practical framework for simultaneous verification of functional correctness and running time analysis of programs in that formalism.
- I have presented a methodology that allows for modular development of programs using specifications as interfaces. Furthermore, I show how to separate the proof development into three clearly-separated parts: reasoning about the correctness of the functional abstraction, reasoning about Separation Logic with Time Credits and reasoning about asymptotic behavior of the running time bound.
- Imperative-HOL-Time provides automatic setup for the latter two parts. First, existing automation for Imperative-HOL is adapted to additionally reason about time credits. Second, a proof tactic supports determining the asymptotic complexity of time bounds.
- We provide many case studies in which we showcase the applicability of our automation.
- In the end we mention limitations of our approach and give directions for future work.

6 LLVM Semantics with a Fine-Grained Cost Model

This chapter is mostly describing work by Peter Lammich and it is a supplemented summary of Section 3 of "For a Few Dollars More — Verified Fine-Grained Algorithm Analysis Down to LLVM" (Haslbeck and Lammich [44]). The idea of using time credits *with currencies* is my main contribution to this chapter and we collaborated on working out the extension of the generic *wp* setup for cost semantics; all the preparatory work [81] and the technical implementation of the extension of the Separation Logic of LLVM with time credits was done by Peter.
As the LLVM semantics described in this chapter is necessary for understanding Chapter 9, I include this chapter to keep my thesis self-contained.

In the last chapter we have seen Imperative-HOL being extended with a cost semantics, and a framework for the verified analysis of running time of programs in that language. A drawback of Imperative-HOL is that its code export facilities only target hybrid languages such as OCaml or SML. While they do support imperative features, their extracted code of a verified algorithm still is slower than reference implementations in purely imperative programming languages. In order to obtain more competitive algorithms Lammich [81] presents an LLVM semantics with some basic reasoning infrastructure and a connection to the Isabelle Refinement Framework through the Sepref tool. That work is only concerned with proving functional correctness and termination of those programs. I will present an extension to that work in Part III of this thesis to also enable reasoning about resource consumption of programs. To lay the ground for that work, I present a cost model for Lammich's LLVM semantics and basic reasoning infrastructure for it in this chapter.

For Imperative-HOL-Time I criticized that every operation costs one unit, and one thus cannot distinguish between them. The LLVM semantics allows only valid LLVM instructions as basic operations. Instead of counting how many instructions are used, we chose a more fine-grained accounting and are counting how many of each instruction are used during a program execution. In order to integrate that into Separation Logic, I came up with the novel concept of time credits with *currencies* (Section 6.1.2).

The step from Imperative-HOL to LLVM does not only mean more efficient extracted code. The many lessons learned from Imperative-HOL also allowed for a more modular design of the verification infrastructure. I will describe the layers of abstraction that are mostly independent of the concrete program semantics. Most of the infrastructure can be reused when instantiated for some concrete program semantics.

At this point the reader has seen at least two examples of the weakest precondition calculus and Separation Logic with time credits (Section 2.4 and Chapter 5). So I think it is save to first show the abstract framework and then the concrete instance

for LLVM. In Section 6.1 I will describe the original framework by Lammich, and how time credits can smoothly be integrated. In Section 6.2 I present an overview of the shallowly embedded LLVM semantics with a cost semantics, and how the basic reasoning infrastructure is instantiated.

6.1 Basic Reasoning Infrastructure

In this section, I describe the basic reasoning infrastructure, which is mostly independent from the LLVM semantics. Rather, the LLVM semantics and its reasoning infrastructure is an instance of this general framework. In Section 6.2, we will then apply it for LLVM.

6.1.1 Basic Monad

At the basis of the formalization is a monad that provides the notions of non-termination, failure, state, and execution costs.

$\alpha\ mres = NTERM \mid FAIL \mid SUCC\ \alpha\ cost\ state$
$\alpha\ M = state \to \alpha\ mres$

Here, *cost* is a type for execution costs, which forms a monoid with operation + and neutral element 0, and *state* is an arbitrary type.

The type $\alpha\ M$ describes a program that, when executed on a state, either does not terminate (*NTERM*), fails (*FAIL*), or returns a result of type α, its execution costs, and a new state (*SUCC*).

It is straightforward to define the monad operations `return` and `bind`, as well as a recursion combinator `rec` over M. Thanks to the shallow embedding, we can also use Isabelle HOL's *if-then-else* to get a complete set of basic operations. As an example, consider the definition of the `bind` operation, in the case that both arguments successfully compute a result:

Assume $m\ s = SUCC\ x\ c_1\ s_1$ and $f\ x\ s_1 = SUCC\ r\ c_2\ s_2$
then we have `bind` $m\ f\ s = SUCC\ r\ (c_1 + c_2)\ s_2$

That is, the result x and state s_1 after the first operation m is passed into the second operation f, and the result and state after the `bind` is what emerges from f. The cost for the `bind` is the sum of the costs for both operations.

The basic monad operations do not cost anything. To account for execution costs, we define an explicit operation `consume` $c\ s = SUCC\ ()\ c\ s$. This is motivated by our approach to first formalize the functional semantics of LLVM, and only then add execution costs on top. This allows for a separation of concerns: we can phrase the functional semantics of an instruction as a monadic expression, independently from the execution costs of this instruction.

6.1.2 Time Credits with Currencies

Our reasoning infrastructure is based on Separation Logic with Time Credits [2, 22, 42]. We follow the algebraic approach of Calcagno *et al.* [14], using an earlier extension [81] of Klein *et al.* [69].

A *separation algebra* over type α consists of the neutral element $0 :: \alpha$, a *disjointness* predicate $\# :: \alpha \times \alpha \to bool$, and a binary *combination* operation $+ :: \alpha \times \alpha \to \alpha$. To guide intuition, an element h of type α is called a *heap* and describes the content of a full heap over a subset of the addresses. The expression $h_1 \# h_2$ states that the addresses covered by h_1 and h_2 are disjoint, and, for disjoint h_1 and h_2, $h_1 + h_2$ describes the combined heap content of h_1 and h_2. Note that $+$ is only defined for disjoint operands. Formally, $(\alpha, +, 0)$ is a commutative monoid, and a heap h is disjoint from the combination $h_1 + h_2$, if and only if it is disjoint from both parts:[1]

$$h \# h_1 + h_2 \iff h \# h_1 \wedge h \# h_2 \quad (\textit{if } h_1 \# h_2)$$

A separation algebra on α induces a *Separation Logic* on predicates over α, with the following connectives:

$$\uparrow\Phi\ a = \Phi \wedge a{=}0 \qquad \square = \uparrow True \qquad (\exists_A x.\ P\ x)\ a = (\exists x.\ P\ x\ a)$$
$$(P \star Q)\ a\ =\ \exists a_1\ a_2.\ a_1 \# a_2 \wedge a = a_1 + a_2 \wedge P\ a_1 \wedge Q\ a_2$$
$$P \vdash Q \quad \textit{iff} \quad \forall a.\ P\ a \implies Q\ a$$

Intuitively, the assertion $\uparrow\Phi$ holds for an empty heap if Φ holds, $\square$ describes the empty heap, and $\exists_A$ is the existential quantifier lifted to assertions. The *separating conjunction* $P \star Q$ describes a heap comprised from two disjoint parts, one described by P and the other described by Q, and entailment $P \vdash Q$ states that Q holds for every heap described by P.

Separation algebras naturally extend over product and function types, i. e., for separation algebras α, β, and any type γ, also $\alpha \times \beta$ and $\gamma \to \alpha$ are separation algebras, where the operations are lifted pointwise.

Note that $(enat, 0, \#, +)$ forms a separation algebra, with $a \# b$ being defined to always hold. Thus, also $ecost = string \to enat$, and $\alpha \times ecost$ are separation algebras, where α is a separating algebra. In particular, in Section 6.2.3 we will use $amemory \times ecost$ with $amemory$ being the original separation algebra that was already used in [81] to describe the memory of LLVM.

We define the function $\$_s\ n$ of type $ecost$ to be the resource function that uses $n :: enat$ coins of the currency $s :: string$, and write $\$_s$ as shortcut for $\$_s\ 1$.

Example 6.1.1. With that notation we can succinctly write cost expressions that contain different amounts of different coins: e. g. $t = \$_{lookup}\ 2 + \$_{add} + \$_{div}$. Here, term t has type $ecost$ and addition is lifted pointwise to functions.

For the type $\alpha \times ecost$ we naturally get a Separation Logic with Time Credits. The time credit assertion $\$$ is defined as $\$\,c = (\lambda a.\ a = (0, c))$, i. e., it describes an empty

[1] This axiom is slightly stronger than the one used by Calcagno *et al.*, but makes reasoning more intuitive, and holds for all models considered.

memory and precisely the time c. Here, the time credit c is a function from *currencies* to *amounts*. This allows for a fine-grained accounting of different instruction types. Thus, the assertion $\$\ \$_s\ n$ describes an empty heap and n time credits of currency s. Complementary, the original primitive assertions over α are lifted to describe zero time credits.

6.1.3 Weakest Precondition and Hoare Triples

The original formalization of LLVM [81] comes already with a generic VCG infrastructure. It is parameterized by a *concrete state* (*cstate*), which describes the actual memory model, and the *abstract state* (*astate*), which forms a separation algebra. The two are connected by an abstraction function $abs :: cstate \to astate$. Moreover, we require a weakest precondition predicate $wp\ c\ Q\ s$, which describes that command c, when executed on concrete state s, terminates with a result r and (concrete) state s' such that $Q\ r\ s'$ holds. Note that the type of $wp :: \gamma \to (\alpha \to cstate) \to cstate \to bool$ is very abstract and not yet specialized to the type of programs $\alpha\ M$, but we will later instantiate it with that. We require *wp* to distribute over conjunctions, i. e.,

$$wp\ c\ Q_1\ s \wedge wp\ c\ Q_2\ s \implies wp\ c\ (\lambda r\ s'.\ Q_1\ r\ s' \wedge Q_2\ r\ s')\ s$$

Finally, let $\top\!\!\top$ be an *affine top* [20], i. e., an assertion with $\square \vdash \top\!\!\top$ and $\top\!\!\top \star \top\!\!\top = \top\!\!\top$, which captures resources that can be safely discarded. Based on these, we define the *Hoare triple* $\{P\}\ c\ \{Q\}$ to hold iff:

$$\forall F\ s.\ (P \star F)\ (abs\ s) \implies wp\ c\ (\lambda r\ s'.\ (Q\ r \star \top\!\!\top \star F)\ (abs\ s'))\ s$$

Intuitively, $\{P\}\ c\ \{Q\}$ holds if, for all states that contain a part described by P, command c terminates with result r and a state where that part is replaced by a part described by $Q\ r \star \top\!\!\top$, and the rest of the state has not changed. Here, $Q\ r$ is the postcondition of the Hoare triple, and $\top\!\!\top$ describes resources that may be left over and can be discarded. The technique to quantify over the rest of the heap F in the definition of the Hoare triple is called the "baked-in frame rule" and proved successful in mechanized proofs [20, §10.2].

Note, that the framework needs the distribution of conjunction over *wp*. Monotonicity of *wp* is a corollary. It very much looks like the distributivity property of quantales. It would be interesting to examine this further and see whether the *wp* framework can be generalized to quantitative reasoning.

The generic VCG infrastructure now provides us with a syntax driven VCG with a simple frame inference heuristics.

The framework is generic enough to allow the instantiation with Imperative-HOL, Imperative-HOL-Time, the shallow LLVM semantics, and — the topic of the next section — LLVM with time. We expect that also other program semantics can be integrated, e. g. the semantics for IMP from Section 2.4.

6.2 LLVM Semantics

Now that I have described the generic VCG infrastructure, I will present the shallowly embedded LLVM semantics, and how to instantiate the generic VCG infrastructure with it. First I describe the LLVM semantics, then I add a cost model and then I show how to prove basic operations correct.

6.2.1 Shallowly Embedded LLVM Semantics

The formalization of the LLVM semantics is organized in layers. At the bottom, there is a memory model that stores deeply embedded values, and comes with basic operations for allocation/deallocation, loading, storing, and pointer manipulation. In addition, the basic arithmetic operations are defined on deeply embedded integers. These operations are phrased in the basic monad, but consume no costs. This way, we could take them unchanged from the original LLVM formalization without cost [81]. For example, the low-level load operation has the signature *raw_load :: raw_ptr → val M*. Here, *raw_ptr* is the pointer type of our memory model, consisting of a block address and an offset, and *val* is our value type, which can be an integer, a pointer, or a pair of values.

On top of the basic layer, we define operations that correspond to the actual LLVM instructions. Here, we map from deeply embedded values to shallowly embedded values, and add the execution costs.

For example, the semantics of LLVM's load instruction is defined as follows:

```
ll_load :: α ptr → α M
ll_load p = do {
    consume $load;
    r ← raw_load (the_raw_ptr p);
    checked_from_val r
  }
```

It consumes the cost[2] for the operation, and then forwards to the *raw_load* operation of the lower layer, where *the_raw_ptr* and *checked_from_val* convert between the shallow and deep embedding of values.

Like in the original formalization[3], an LLVM program is represented by a set of monomorphic constant definitions of the shape *def*, defined as follows:

```
def = proc_name var* ≡ block
block = var ← cmd; block | return var
cmd = ll_<opcode> arg* | ll_call proc_name arg* | llc_if arg block block
      | llc_while block block
arg = var | number | null | init
```

The code generator checks that the set of definitions is complete and adheres to the required shape. It then translates them into LLVM code, which merely amounts to

[2] See Section 6.2.2 for an explanation of our cost model.

[3] Actually, the only change to the original formalization is the introduction of the *ll_call* instruction, to make the costs of a function call visible.

pretty printing and translating the structured control flow by `if` and `while`[4] statements to the unstructured control flow of LLVM. A powerful preprocessor can convert a more general class of terms to the restricted shape required by the code generator. This conversion is done inside the logic, i.e., the processed program is proved to be equal to the original. Preprocessing steps include monomorphization of polymorphic constants, extraction of fixed-point combinators to recursive function definitions, and conversion of tuple constructors and destructors to LLVM's *insertvalue* and *extractvalue* instructions.

In summary, the layered architecture of the LLVM formalization allowed for a smooth integration of the cost aspect, reusing most of the existing formalization nearly unchanged.

6.2.2 Cost Model

As a cost model for running time, we chose to count how often each instruction is executed. That is, we set $cost = string \to nat$, where the string encodes the name of an instruction. It is straightforward to define *0* and *+* such that (*cost, +, 0*) forms a monoid. It is thus a valid cost model for our basic monad.

Charguéraud and Pottier [22, §2.7] discuss the adequacy of abstract cost models in a functional setting. In their classification, our abstraction is on Level 2. For a discussion of how realistic our cost model of counting LLVM instructions is, I refer to Section 3.3 in [44]. I will only point out our main design choices here.

The control flow for calling procedures and conditional branching incurs costs of currencies we call *call* and *if* respectively. By only accepting LLVM programs of the form mentioned above (with *ll_call* and *llc_if*) the code generator makes sure that each control flow construct is paid for by a `consume` of the respective currency.

The *insertvalue* and *extractvalue* instructions, which are used to construct and destruct tuple values, have no associated costs. The main reason for this design is to enable transparent use of tupled values, e. g., to encode the state of a while loop. We expect LLVM to translate the members of the tuple to separate registers anyway, such that no real costs are associated with tupling/untupling.

We define the *malloc* instruction to take cost proportional to the number of allocated elements. Note that LLVM itself does not provide memory management, and our code generator forwards memory management instructions to the *libc* implementation of the target platform. We use the *calloc* function here, which is supposed to initialize the allocated memory with zeros. While the exact costs of that are implementation-dependent, they certainly will depend on the size of the allocated block.

[4] Primitive while loops are not strictly required, as they can always be replaced by tail recursion. Indeed, our code generator can be configured to not accept while loops, and our preprocessor can automatically convert while loops to tail-recursive functions. However, the efficiency of the generated code then relies on LLVM's optimization pass to detect the tail recursion and transform it to a loop again.

6.2.3 Using the Weakest Precondition Framework

To instantiate the generic framework from Section 6.1.3 to our LLVM cost model, first we set

$$cstate = memory \times ecost$$

where *memory* is the memory type from the original LLVM formalization. The weakest precondition is defined by

$$wp :: \alpha\ M \to (\alpha \to cstate \to bool) \to cstate \to bool$$
$$wp\ m\ Q\ (s,cc) = (\exists r\ c\ s'.\ m\ s = SUCC\ r\ c\ s' \wedge c \leq cc \wedge Q\ r\ (s',\ cc - c)).$$

Intuitively, the costs *cc* stored in the state are the *credit* available to the program. The weakest precondition holds if the program runs with real costs c that are within the available credit, and Q holds for the result r, the new memory s', and the new credit, $cc - c$ which is the old credit reduced by the actually required costs. We can prove that our *wp* distributes over conjunctions.

Note that actual costs have type $string \to nat$,[5] i. e., are always finite, while the credits have type $ecost = string \to enat$, i. e., there can be infinite credits. When setting the credit to be infinite for all instruction types, we get the classical weakest precondition that requires termination, but enforces no time limit.

Our concrete state type, in particular the memory, does not form a separation algebra, as the natural memory model of LLVM has no natural notion of partial memories. Thus, we define an abstraction function that maps a concrete state to an abstract state *astate*, which forms a separation algebra:

$$astate = amemory \times ecost \qquad abs\ (m,\ c) = (abs_m\ m,\ c)$$

Again, *amemory* and abs_m is the abstract state and abstraction function from the original LLVM formalization. The costs already form a separation algebra, so we do not abstract them further.

We set $\mathbb{T}$ to describe the empty memory and any amount of time credits. This matches the intuition that a program must free all its memory, but may run faster than estimated, i. e., leave over some time credits.

Now we can instantiate the generic reasoning infrastructure.

6.2.4 Primitive Setup

Once we have defined the basic reasoning infrastructure, we have to prove Hoare triples for the basic LLVM instructions and control flow combinators. As we have added the cost aspect only at the top level of our semantics, we can reuse most of the material from the original LLVM formalization without time. Technically, we instantiate the reasoning infrastructure with a weakest precondition predicate *wpn*, which only holds for programs that consume no costs. We define:

$$wpn\ m\ Q\ s = wp\ m\ (FST \circ Q)\ (s,\ 0)$$

[5] We fixed that in the beginning of Section 6.2.2.

Here, the operator $FST\ P = \lambda(s,\ c).\ P\ s \wedge\ c = 0$ lifts assertions from the Separation Logic without time credits.

The resulting reasoning infrastructure is identical with the one of the original formalization, most of which could be reused. Only for the topmost level, i. e., for those functions that correspond to the functional semantics of the actual LLVM instructions, we lift the Hoare triples over *wpn* to Hoare triples over *wp*:

$$\{P\}\ c\ \{Q\}_{wpn} = \{FST\ P\}\ c\ \{FST \circ Q\}$$

Example 6.2.1. Recall the low-level *raw_load* and the high-level *ll_load* instruction from Section 6.2.1. The *raw_load* instruction consumes no costs, and our original LLVM formalization provides the following Hoare triple:

$$\{raw\_pto\ x\ p\}\ raw\_load\ p\ \{\lambda r.\ \uparrow(r{=}x) \star raw\_pto\ x\ p\}_{wpn}$$

This can be transferred to a Hoare triple over *wp*:

$$\{FST\ (raw\_pto\ x\ p)\}\ raw\_load\ p\ \{\lambda r.\ \uparrow(r{=}x) \star FST\ (raw\_pto\ x\ p)\}$$

which is then used to prove the Hoare triple for the program *ll_load*

$$\{\$\ \$_{load} \star pto\ x\ p\}\ ll\_load\ p\ \{\lambda r.\ \uparrow(r{=}x) \star pto\ x\ p\}$$

where $pto\ x\ p = FST\ (raw\_pto\ (to\_val\ x)\ (the\_raw\_ptr\ p))$.

6.2.5 Free for Free

Note that in our semantics, both memory allocation and memory deallocation consume costs of currencies *malloc* and *free* respectively. However, the automatic data refinement tool we are going to design (see Section 9.2.3) has to automatically insert destructors, which free memory. A destructor d that destroys an object described by assertion A is characterized in the following way:

$$destructor\ A\ d = (\forall a\ c.\ \{A\ a\ c\}\ d\ c\ \{\square\})$$

In particular, all costs required for destruction must already be contained in the assertion A. In practice, this means that we pay for the destruction of an object upon its allocation. Thus, we prove the following Hoare triples for allocation and deallocation:

$$\{\$\,\$_{malloc}\ n \star\ \$\,\$_{free} \star \uparrow(n > 0)\}$$
$$ll\_malloc\ \alpha\ n$$
$$\{\lambda p.\ range\ \{0..{<}n\}\ (\lambda\_.\ init)\ p \star malloc\_tag\ n\ p\}$$

$$\{range\ \{0..{<}n\}\ blk \star malloc\_tag\ n\ p\}\ ll\_free\ p\ \{\square\}$$

Intuitively, to allocate a block of size n, one has to pay n units of *malloc* and *1* unit of *free*. To free a block, no explicit costs have to be paid. Note that the credits for the free are stored in the *malloc_tag* assertion, along with the block ownership from the memory model:

$$malloc\_tag\ n\ p = FST\ (raw\_malloc\_tag\ n\ (the\_raw\_ptr\ p)) \star \$\,\$_{free}$$

In essence, we use amortization to pay for the deallocation of a data structure already during allocation. This is seamlessly supported because we can use time credits in our Separation Logic assertions.

Example 6.2.2. Consider an LLVM array containing a list of elements that can be represented as LLVM values (type α). The assertion for an array consists of two parts. The first part holds the content of the array. It is described by a base pointer (p), a map, and its domain — the set of offsets that are owned by the array. The second part holds the ownership of the whole block, represented by the *malloc_tag*.

$$array_{assn}\ xs\ p = range\ \{0..<|xs|\}\ (\lambda i.\ xs\ !\ i)\ p \star malloc\_tag\ |xs|\ p$$

Now, allocation *array_new* essentially is *ll_malloc* with the following Hoare triple:

$$\{\$\$_{malloc}\ n \star \$\$_{free} \star \uparrow(n > 0)\}$$
$$ll\_malloc\ \alpha\ n$$
$$\{\lambda p.\ array_{assn}\ (replicate\ n\ init)\ p\}$$

Here, the type α is assumed to have a dedicated element *init*.

The program *ll_free* is a destructor for arrays: *destructor ll_free*. Once the data structure is initiated, it can be destroyed safely at any point. The cost for freeing the element can be paid for by the time credits that are reserved for that purpose in the assertion $array_{assn}$.

6.2.6 More Infrastructure

Using the VCG and the Hoare triples for the LLVM instructions, we can now define and prove correct various data structures and algorithms.

While this works smoothly for simple data structures like arrays, it does not scale to more complex developments.

Instead of building up more involved VCG infrastructure like we did in Chapter 5 for Imperative-HOL-Time, we only provide simple data structures on this basic layer. Then, we push most reasoning to a more abstract level which we will introduce in Part III of this thesis.

6.3 Summary

- I showed how time credits can be augmented with resource currencies in order to measure different quantities at the same time. This is used to count the number of calls for each LLVM instruction in a program execution.

- The general weakest precondition setup can — in theory — be instantiated for program semantics that provide a *wp* predicate, a garbage collection assertion ($\mathbb{T}$) and an abstraction of the memory model into a separation algebra (via abstraction *abs*). The setup provides basic verification infrastructure, with a frame inference algorithm and a verification condition generator.

- We do not provide sophisticated infrastructure to support time frame inference (like in Section 5.4.3) but defer that reasoning to the refinement approach described in the following part.

Part III

Refining Resources

In the previous part we have seen how to simultaneously verify functional correctness and time complexity of imperative algorithms and data structures. The usual approach is an ad hoc refinement: first a purely functional implementation of an algorithm is proved correct in Isabelle/HOL, then it is refined to an imperative implementation that also fulfills some timing constraints. While this approach is viable for medium-sized algorithms (e. g. Guenéau's incremental cycle detection algorithm [42]) I believe that for larger developments it is necessary to modularize and separate reasoning on different levels of abstraction. In this part, I present a solution to that problem by extending stepwise refinement towards reasoning about resource consumption of programs.

Stepwise refinement allows to specify algorithms and their components, reason about algorithmic ideas on an abstract level, and add implementation details step by step. In this part, I show that stepwise refinement can not only structure reasoning about functional correct but also reasoning about the resource consumption of programs. This is joint work with Peter Lammich. We have used the Isabelle Refinement Framework (IRF) [77] and the tools in its eco-system as a blueprint and extended them for our purposes.

The standard IRF has been used to verify a whole array of algorithms and data structures. I will summarize its principal ideas, tools, and applications in Chapter 7. Furthermore, I will discuss related work both for the IRF and for other monadic approaches to reason about the running time of algorithm. Before that, I will present a review of the exposition of an algorithm analysis in an algorithm textbook in order to extract features of a *natural* presentation of algorithmic ideas.

Computation with resource usage is abstractly modeled in the nondeterministic result monad with resources (NREST[6]). Because it builds upon the IRF's *nres* monad, I will first present a summary of its relevant concepts. I will motivate the main design choices, present the model of computation and its refinement calculus, explore its theory, and present reasoning infrastructure (Chapter 8). I will show how the verification was structured with locales (Section 8.3) in the first iteration, and how resource currencies and currency refinement can be used to structure verification in a more natural way (Section 8.4).

The goal is to use the abstract modeling to verify concrete programs with their resource consumption. I present a method to synthesize concrete implementations from abstract NREST programs. As we have seen in Part II, any program semantics featuring Separation Logic can be modularly supplemented with a cost semantics. Similarly, we identify common features to be provided by a back end to support synthesis from NREST and implement synthesis for both Imperative-HOL-Time and LLVM-Time (Chapter 9).

Finally, I will present applications (Chapter 10): Kruskal's algorithm for minimum spanning trees, an abstract analysis of the amortized dynamic array data structure, and the introsort algorithm. The first of the three was verified using the first iteration of our framework and comes with a synthesized implementation in Imperative-HOL-

[6]The acronym stands for **N**ondeterministic **RES**ult monad with **T**ime for historical reasons but can be used for other resources.

Time. The latter two were verified with the framework with resource currencies and result in competitive LLVM programs with verified running time bounds.

7 The Blueprint: Algorithm Analysis and Isabelle Refinement Framework

The prevalent approach in software verification seems to be what I describe as "bottom-up": an imperative implementation is developed that it is verified to fulfill its specification. However, the direction of development of verified algorithm implementations seems to be counter-intuitive. First, the programmers have certain abstract algorithmic ideas in mind, then they realize them as concrete implementations. After that, proof engineers reverse that process again in order to prove the correctness of implementations by introducing levels of abstraction and reasoning about them. One clear drawback is the cyclic nature of this process. Insights from failing proof attempts for the concrete implementation may require modifications in the implementation. Those may in turn break existing proofs and require more work on those.

In this chapter I argue that a more natural approach to software verification is a "top-down" approach that features stepwise refinement: first we provide a specification of some operation, then present the algorithmic idea and reason that it fulfills the given specification. After that we elaborate the algorithm and add implementation details by stepwise refinement in order to obtain a deterministic algorithm that still fulfills the top-level specification. As a last step we automatically synthesize a concrete implementation from the deterministic algorithm. That synthesis also emits a proof of correctness and a verified resource bound.

Obtaining efficient implementations with verified functional correctness and quantitative properties is not the only goal here. I also want to establish a framework for effective reasoning about abstract algorithms and their quantitative properties.

The exposition of algorithmic ideas in textbooks (like CLRS [24]), papers or discussions often has the form of *"pseudocode"*: a particular form of communicating computational ideas which lies between a vague description in plain English and a verbose display of program code. Choosing the right level of abstraction is vital to effectively convey those ideas and demonstrate their correctness. A side product of the refinement approach is that algorithms can be presented and formally analyzed at different levels of abstraction.

To me, the refinement approach serves two purposes. First, it allows to express algorithmic ideas and their correctness through formal algorithm sketches. Second, it provides a methodology to obtain executable implementations from such algorithm sketches. When only functional correctness is concerned, the "top-down" refinement approach is realized by the Isabelle Refinement Framework. The content of the third part of this thesis is to describe how that framework can be extended for reasoning about resource consumption. To prepare the ground for this extension, I provide an

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Figure 7.1: The pseudocode of Kruskal's algorithm in the textbook by Cormen et al. [24].

overview of the Isabelle Refinement Framework and review its components that are relevant for this thesis in Section 7.2. I also discuss work related to the IRF and other approaches to model resource consumption in a monadic setting. Before that, I reflect on the pen-and-paper way of proving the correctness of algorithms, the role of asymptotic analysis and what features a formal treatment in a proof assistant needs to have.

7.1 Presentation of Algorithmic Ideas

Ultimately, formal proofs should be both readable for humans and computers. In order to design the verification of the analysis of imperative programs as natural as possible, we first need to study how algorithmic ideas are naturally conducted. While I would like to see a structured linguistic analysis of that matter, I could not find any relevant literature. Instead, in this section I will anecdotally present the exposition of one algorithm and the description of its analysis in a textbook. I will extract some features a natural approach needs to exhibit. In Section 10.4 I will examine the framework presented in the rest of this thesis w. r. t. these features.

Consider the description of Kruskal's algorithm in the textbook by Cormen et al. [24, Section 23.2].[1] They start by pointing out that the algorithm is an elaboration of a more general algorithm (GENERIC-MST) and use terminology established for that algorithm (e. g. *safe edge*). Then, they state how to specialize the general algorithm and use a mathematical theorem proved earlier (i. e. that *light edges* are safe) to argue for its correctness. After establishing correctness they provide intuition on how to efficiently implement the specialization (i. e. how to identify *light edges* using a disjoint-sets data structure) and present pseudocode for it (Figure 7.1).

The pseudocode is "designed to be readable by anyone who has done a little programming" [24, Preface]. It consists of control flow commands (***for***, ***if***), mathematical

[1] It might be instructive for the reader to take the time and have a look at that section before reading on.

notation ($A = \emptyset$), procedure calls of operations of abstract data types (e. g. MAKE-SET of a disjoint-sets data structure) and operations described in plain English. Note that this algorithm exhibits nondeterminism: The order of looping over the vertices (in line 2) and edges (in line 5), as well as the treatment of ties while sorting are not fixed. Both aspects are not relevant for the correctness of the algorithm and thus are safely ignored. But keep in mind that nondeterminism is important for modeling algorithm sketches at the right level of abstraction.

By referring to the pseudocode Cormen et al. argue that it indeed implements the idea of Kruskal's algorithm. Also, for reasoning about the running time complexity they go through the code line by line, replacing the costs of abstract operations by those of well-known implementations (here union-find for disjoint-sets and $O(E \log E)$ for sorting) and using asymptotic complexity for abstraction. For example: "The ***for*** loop of lines 5-8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest." Together with the results on the data structure *union-find* (established earlier in [24, §21]) they conclude that "the disjoint-set operations take $O(E\,\alpha(V))$ time" and "the total running time of Kruskal's algorithm is $O(E \log E)$". Here, note that the use of asymptotic complexity serves two different purposes.

First, the bookkeeping of concrete implementation-dependent constants is too tedious. Giving an asymptotic bound simplifies that by hiding those constants. In the example, it is even impossible to determine exactly how often FIND-SET and UNION are called. Stating the asymptotic behavior ($O(E)$) is a way of abstracting that quantity, and in the end we will only be interested in the asymptotic running time of the whole algorithm anyways. But notice that the upper bounding at this point does not necessarily have to involve asymptotics, i. e. it works also without the number of edges E going to infinity. I argue that asymptotic analysis is not the only way for hiding constants that otherwise would distract us from the main argument. Instead, the proof assistant can handle the bookkeeping of constants. Intuitively, different quantities are counted at different levels of abstraction: the number of loop iterations, the number of calls of UNION, the number of array lookups, or the number of *load* instructions. We will introduce the concept of *resource currencies* (Section 8.4) to model this intuition. This will put additional structure into the reasoning. Furthermore, asymptotic complexity allows giving a succinct and robust specification of an operation. In Section 5.2 we have seen that hiding details of implementations and running time bounds results in useful interfaces, which typically do not change for small modifications in the implementation.

The second purpose of asymptotic complexity is to make algorithm analyzes comparable. Asymptotic complexity classes of problems and their algorithms are invariant for many platforms and machine models. This allows to intuitively port an analysis of some algorithm on a specific platform to another, maintaining the complexity class. Consequently, complexity results for algorithms can easily be reused in more complex algorithms without explicitly arguing how the computational models are compatible with each other. While this is straightforward in informal proofs, it is tedious if conducted with a mechanized proof. I think for this purpose asymptotic complexity is still the right tool. Like in Chapter 5 our approach is, however, to conduct the running time

analysis with concrete constants as far as possible,[2] and to use the proof assistant to conveniently hide details. This avoids common pitfalls when working with asymptotic complexity, especially when several variables are involved (cf. [41, Section 2]) At the very end of the analysis we extract a running time bound, analyze its asymptotic behavior and compare it to other results.. Guéneau [40, §4.3] has an excellent discussion of the role of O-notation in formal complexity proofs.

> We observed that stating specifications including asymptotic complexity only makes sense above a certain level of abstraction. Once we synthesize programs for a specific machine architecture that fixes a word length and only has finite memory requiring time bounds with certain asymptotic behavior makes no sense: all running time bounds essentially are constant, with a very big constant. To the best of my knowledge there is no better solution then to thread through the constants by bookkeeping, and then rate the algorithms by their leading terms and constant factors. Asserting the asymptotic complexity claims at the abstraction level before fixing a finite machine model still indicates the complexity class of the algorithmic idea and thus of the implementation. The problem only arises when requiring a specification of an implementation to have a certain running time complexity. That will always be meaningless, because one can always prove a constant running time bound. It is not clear to me how to address this issue in the correct manner.

Let me recapitulate how the presented algorithm analysis was structured, phrase it in stepwise-refinement terminology and suggest how the informal sketch can be turned into a formal proof. First, a general algorithm (using the concept of *safe edges*) for which we already have a correctness proof was instantiated with a more concrete concept (*light edges*) yielding a first refined algorithm. Then, a second algorithm is explicitly given as pseudocode (Figure 7.1) using operations whose specifications are implied or briefly explained by mathematical notation or prose. The refinement relation between the second and first iteration is left implicit. Instead, the pseudocode is illustrated operationally with an example execution. During that and the derivation of the time complexity of Kruskal's algorithm, another implicit refinement takes place: an imaginary third algorithm is outlined which settles all the nondeterministic choices (e. g. order of traversal of vertices, ties during sorting) and replaces the used subroutines by imaginary implementations (e. g. sorting). It is left implicit that this imaginary algorithm refines the pseudocode, but it is of that algorithm we are actually determining the time consumption. Many of these refinement steps are omitted as they are straightforward and distract from the core argument, but they are necessary when turning the sketch into a formal proof.

On a higher level of abstraction what is happening here is that a computational problem (finding a minimum spanning tree) is reduced to a combination of problems (sorting, disjoint sets operations) for which we already have solutions. Once this reduction is explained and the reader is convinced, it is left as an exercise to spell out the straightforward details of combining the solutions of the subproblems. Ideally, a formal proof would reflect exactly that pattern.

[2] When analyzing an recurrence relation with the Akra–Bazzi method, we only obtain the result for the membership of some complexity class. We do not get a closed form solution in that case.

It is worth noting what kind of role the pseudocode in Figure 7.1 plays. In the imaginary refinement chain one only has to prove local properties that depend on one level up and down the chain: for functional correctness one has to show that the specification that stems from the computational problem is met, given that the subproblems are solvable. For the running time analysis, one has to count how often the respective subproblems are called, multiply them by their specified cost, add them up, and prove that they are bounded by the costs of the more abstract program. Especially for high level algorithms that only combine other algorithms this simple analysis technique suffices. Only for more involved running time analyzes we need techniques like linear recurrences or the Master Theorem.

In Section 10.1, I present the verification of Kruskal's minimum spanning tree algorithm as a case study. However, instead of following the route taken by CLRS in the example of this section, Kruskal's algorithm is shown to be an instance of the general greedy algorithm for finding a *minimum weight basis* in a matroid. Regardless, the same technique of specializing a general algorithm and refining it stepwise applies.

Let me summarize the main features a formal exposition of the analysis of algorithms needs to exhibit. First of all there needs to be a formalism to express algorithm sketches. This formalism should allow one to use precise mathematical expressions and at the same time to leave open implementation details. Those details may be postponed or just left to the reader. There should be a way of expressing qualitative properties about the result of the algorithm and quantitative properties about the execution of the algorithm. It should be possible to relate algorithm sketches at different levels of abstraction. Algorithm sketches should have a semantic and operational meaning. It should be possible to display the execution of an algorithm sketch in order to illustrate the operational meaning of the algorithm. It is desirable that the formalism is readable and executable by a human and a machine. Also the proofs for showing correctness and quantitative properties of algorithm sketches should be readable by both humans and machines.

I believe that the overall approach of presenting algorithm analyzes like in textbooks can be reproduced in a proof assistant. Stepwise refinement allows to represent algorithms on different levels of abstraction and establish refinement relations between them. It facilitates to separate concerns and split the mathematical correctness proof from implementation details.

In the next section I will review the main parts of the Isabelle Refinement Framework, a framework that allows algorithm analysis for *functional correctness*.

7.2 The Isabelle Refinement Framework

The Isabelle Refinement Framework (IRF) essentially implements the idea of top-down development of verified algorithms with stepwise refinement in Isabelle/HOL. At the top of the *refinement chain* is an abstract *specification* of some operation, which can be refined into several iterations of *monadic programs*. Eventually, if such a monadic program fulfills certain wellformedness properties, a *synthesis component* can automat-

ically generate an *implementation* in one of several *back end* semantics. From those implementations a *code generator* can export *program text* in different programming languages. The program text in turn can then be compiled by general-purpose *compilers* to obtain executable machine code.

Note that the *trusted code base*, i. e. the unverified pieces of software used, starts with the code generator. Everything above that level happens within the logic of Isabelle/HOL and thus is verified. While there are current efforts on extending the scope of trust, I focus on what happens above that line.

The formalism used to phrase specifications and monadic programs is called the *nondeterministic result monad* (*nres*). While this monad and the mentioned vertical structure is basically unchanged since the first iteration [94], the other components have evolved considerably. We will see details of that monad in Section 8.1. In the following I will give a chronological overview of the evolution of those components and of applications realized with them.

Lammich [83] pioneered a framework for refinement of monadic programs and introduced the nres monad. The framework already provided facilities for *specification refinements* and *data refinement* in the form of verification condition generators. The first case study was the verification of Hopcroft's algorithm for automata minimisation [94]. The refinement from monadic programs to executable functional programs in the logic of Isabelle/HOL was proven without tool support. However, the Isabelle Collections Framework (ICF) [89] readily provided executable *functional* data structures that could be used to refine abstract data structures like sets and finite maps. The Isabelle code generator was used to obtain program text in SML, OCaml, Haskell and Scala.

To automate the refinement of data structures from monadic programs to HOL functions, Lammich designed the synthesis component *Autoref* [77]. That tool was used to verify a series of efficient algorithms [87, 91, 64], which culminated in the first fully verified LTL model checker [35]. It was later extended by partial order reduction [11]. The journal paper by Lammich and Lochbihler [88] contains the latest description of the Autoref tool. It is still used today (e. g. [12, 10]) to obtain efficient code from verified algorithms in Isabelle/HOL.

As many efficient algorithms use imperative data structures, the next step was to design an *imperative* back end for the IRF. The *Sepref* tool [85, 84] refines monadic programs into Imperative-HOL [13], which is equipped with a Separation Logic [90]. The Isabelle code generator setup for Imperative-HOL generates program text in functional programming languages that support references and mutable arrays. Many case studies show the maturity of the framework: from verification of classical imperative algorithms and data structures [93, 82, 135, 54, 52, 92], over algorithms for the SAT problem [86, 78, 36, 9, 79], to an timed automata model checker [136]. Using a functional-imperative hybrid language as a back end is particularly suitable, as one can choose to refine performance-critical data structures imperatively but also can keep features of functional languages when it is more convenient. While using Imperative-HOL as a back end gained some additional performance, the extracted code is typically less efficient than reference implementations in purely imperative languages.

As a next step towards efficient verified algorithms, Lammich [81] presents a shallowly embedded LLVM semantics (*Isabelle-LLVM*) and modifies Sepref to synthesize Isabelle-LLVM implementations from monadic programs. A code generator turns Isabelle-LLVM programs into LLVM intermediate representation, which can be compiled by LLVM to executable code. First case studies for simple algorithms on arrays suggest a speed up of at least 1.5 in contrast to implementations extracted in SML [81]. Recently, Lammich [80] conducted a larger case study and verified the *introsort* and *pdqsort* algorithms. The extracted verified algorithms perform on par with the respective implementations from the standard library.

While the recent development of the IRF focused on the efficiency of extracted algorithms, the formalism for representing abstract algorithm stayed unchanged. I will show how the IRF can be extended to not only give functional correctness results for the implementations but also guarantees on the resource consumption by extending the nres monad. Essentially, I will describe how to apply the potential method on it, adapt the tools of the IRF accordingly and apply running time analysis to some case studies.

7.3 Related Work

In the last section I already mentioned many resources that describe or use components of the IRF. For related work to the *Autoref* framework I refer to [88, §8], and Lammich [84, §7.2] mentions related work for *Sepref*. I want to discuss related work to refinement, the nondeterminism monad *nres* and reasoning about resource consumption in a monad.

While data refinement dates back to Hoare [56], Back [3] first proposed a refinement calculus for imperative programs. There are textbooks [4, 26] that give a good overview over the field.

Schwenke and Mahony [124] combine monads with refinement and Klein et al. [23, 70] describe the nondeterminism monad used in the seL4 project. It also comes with a refinement calculus.

Weegen et al. [132] present a shallow monadic embedding of programs and a monad transformer that piggybacks a monoid, e. g. simply counting the number of steps of a program, onto an existing monad by pairing. They define a tree nondeterminism monad that essentially maintains a distribution of results and use the monad transformer to get a "nondeterministically profiled" monad. With that they verify the average-case complexity of quicksort in Coq. As presented in Chapter 3 Hölzl [62] uses the Giry monad to reason about the expected running time of programs. Eberl et al. [34] uses it to verify randomized quicksort, the average case analysis of deterministic quicksort and quantitative properties of other randomized data structures.

The use of a simple monad counting the number of steps of a program allows for a lightweight model of execution time. Here are two recent works that employ that technique. McCathy et al. [101] study the running time for functional programs, verifying e. g. mergesort, red-black trees and Braun trees in Coq. Nipkow [111] verifies the amortized complexity of root-balanced trees.

Rajani et al. [123] present a unifying type-theory for higher-order amortized cost analysis, which involves a cost monad. Rajani [122] applies type-theoretic approach to Information Flow Control and generalizes the theory to allow any commutative monoid in the cost monad.

Besides our work that I will describe in the next chapter, I am not aware of any work involving a nondeterminism monad that tracks time or resource bounds and comes with a refinement calculus.

7.4 Organization of the Rest of Part III

In the rest of the third part of this thesis I present two iterations of the combination of stepwise refinement with resource bounds. The first iteration [45] extends IRF's *nres* monad to reason about resources, measures consumption with a natural number and synthesizes programs in Imperative-HOL-Time (cf. Chapter 5). The second iteration [44] additionally uses *resource currencies* and targets LLVM-Time (cf. Chapter 6). Because both iterations affect the whole stack of the IRF, and their ideas build upon each other, I will present the material intertwined, from top to bottom, and in increasing elaboration.

In Chapter 8, I will first summarize the relevant concepts of *nres*. Then I will present a generalization of that monad together with three instances: the classical monad, the monad that measures resources with a number, and the monad with resource currencies. I will show how the main techniques for the classical *nres* monad can be adapted to treat the first and then the second iteration.

Then, in Chapter 9 I will show how the Sepref tool for synthesizing implementations can be extended. This involves showing the necessary changes for the first, followed by the second iteration.

Finally, I will present case studies for both iterations (Chapter 10) and close with a discussion of the approach and related work.

7.5 Summary

- I presented the exposition of an algorithm and its informal analysis from a textbook and extracted some features a natural treatment of algorithm analysis should exhibit.
- The *Isabelle Refinement Framework* (IRF) allows stepwise refinement of algorithms in Isabelle/HOL. It features the *nres* monad to express nondeterministic computation and provides back ends for Isabelle/HOL functions (Autoref), Imperative-HOL and LLVM (Sepref). From those back ends executable programs can be extracted by code generators.

8 NREST

The content of this chapter is joint work with Peter Lammich. This chapter is based on a combination of material from "Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL" (Haslbeck and Lammich [45]) and "For a Few Dollars More – Verified Fine-Grained Algorithm Analysis Down to LLVM" (Haslbeck and Lammich [44]) supplemented with more explanations and examples.

In this chapter I will give a gentle introduction in the nondeterministic result monad with resource bounds, which is a generalization of the *nres* monad of the Isabelle Refinement Framework. I will first introduce the *nres* monad [94], give illustrative examples and present its reasoning infrastructure. Then I will present two iterations of generalizations of *nres* that allow for reasoning about resource bounds. The first one measures resources in one number, and the second introduces the novel concept of *resource currencies* and *currency refinement*.

8.1 Modelling Nondeterministic Computation

For modeling nondeterministic computation we consider the following two aspects:

First, as a running example, consider we want to sum over all the elements of a finite set of numbers S. A natural algorithm would just pick one number at a time from the set and add it to some accumulator. At any point during the execution of this algorithm, it does not matter which number is chosen next, as long as we only choose numbers from the set and do not choose a number twice by removing it from the set. We want to specify the program by its designated property, but not with a specific result. For example, for the set S=$\{1, 2, 3\}$ the set of possible results might be $\{(1, \{2, 3\}), (2, \{1, 3\}), (3, \{1, 2\})\}$, where the first component of each result is the chosen element and the second is the rest of the set. Later we might implement the algorithm deterministically by always choosing the minimal element, i. e. in the example it chooses $(1, \{2, 3\})$. We then say the deterministic implementation *refines* the general algorithm. To sum it up, a computation should be modeled as a set of possible results.

Second, we want to model recursive computations and define recursion by a standard fixed-point construction over a flat lattice. The lattice must have a dedicated element for nontermination, which we call `fail`. Furthermore, later we will see that this element also signals failing assertions.

We now formalize the above intuition:

α *nres* = `fail` | `res` (α *set*)

A computation is either ***fail***, or ***res*** X, where X is a set of possible results.

Note that this shallow embedding makes no formal distinction between syntax and semantics. Nevertheless, we refer to an entity of type *nres*, as *program* to emphasize the syntactic aspect, and as *computation* to emphasize the semantic aspect.

We define the *refinement ordering* by lifting the subset ordering and setting ***fail*** as the top element. The *refinement ordering* corresponds to the intuition of refinement: $m \leq m'$ reads as m *refines* m', i. e., all possible results of m are also possible results of m'. To fix terminology, one can alternatively say "m implements m'", "m' is refined by m", or "m' is refined to m". The latter emphasizes, that m specifically was created in order to refine m'. Rarely, we say that m and m' are in a refinement relation. The order of the refinement should be clear from the context.

Example 8.1.1. For a simple example, consider a computation that removes an element of the set $\{1, 2, 3\}$. That computation is expressed in *nres* in the following way:

$$m' = \textbf{res}\ \{(1, \{2, 3\}), (2, \{1, 3\}), (3, \{1, 2\})\}$$

The computation m' can serve as a *specification* and can be refined by more concrete computations. For example, computation $m = \textbf{res}\ \{(1, \{2, 3\})\}$ only produces one of the possible results. Thus m refines m'.

To conveniently model actual computations, we define some combinators. We define ***spec*** P to be the computation of any result r that satisfies $P\ r$: ***spec*** $P = \textbf{res}\ \{r.\ P\ r\}$. Furthermore, we define ***return*** $x = \textbf{res}\ \{x\}$ to compute the single result x.

The combinator ***bind*** $m\ f$ models the sequential composition of computation m and f, where f may depend on the result of m:

$$\textbf{bind} :: \alpha\ nres \to (\alpha \to \beta\ nres) \to \beta\ nres$$
$$\textbf{bind}\ \textbf{fail}\ f = \textbf{fail}$$
$$\textbf{bind}\ (\textbf{res}\ X)\ f = Sup_{x \in X}\ f\ x$$

If the first computation m fails, then also the sequential composition fails. Otherwise, we consider all possible results x of m and invoke $f\ x$. The supremum essentially is the union of the set of possible results of f for all the different intermediate results of m. However, it makes the whole expression fail if one of the reachable $f\ x$ fails.

For writing larger programs conveniently we use monadic do-notation. We will postpone the proof of the monad laws [127] for now. For sequential composition we will write ***do*** $\{\ x \leftarrow m;\ f\ x\ \}$ for ***bind*** $m\ f$ and ***do*** $\{\ m;\ f\ \}$ for ***bind*** $m\ (\lambda\_.\ f)$. Larger do-blocks are joined.

Example 8.1.2. Using the combinators just introduced we can express a program that removes an element from a set and adds it to an accumulator. We pull out the respective operations following the *mañana* principle [19] of stepwise refinement: "When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later." In our case, we state a specification, use it, and provide a refinement later.

```
1    choose'_spec S = spec (λ(s, S'). s ∉ S' ∧ S' ∪ {s} = S)
2    add_spec a s = return (a + s)
3
4    add_elem' (a, S) = do {
5        (s, S') ← choose'_spec S;
6        a' ← add_spec a s;
7        return (a', S')
8      }
```

Then (*42*, {*2*, *3*}) is a possible result of program *add_elem'* for *a=41* and *S*={*1*, *2*, *3*}. We can express that by stating

$$\texttt{return}\ (42, \{2, 3\}) \leq add\_elem'\ (41, \{1, 2, 3\})$$

Observe what happens when the set S is empty. At first, one would intuitively expect that the program should fail in that case. Instead, in line 5 the computation does not have any result, i. e. $choose'_{spec}\ \emptyset = \texttt{res}\ \emptyset$. This is propagated by the (implicit) `bind` operators in line 6. The expression simplifies to the supremum over the empty set and the result of that is again the bottom element of the ordering, i. e. $\bot = \texttt{res}\ \emptyset$. Consequently, we have $add\_elem'\ (a, \emptyset) = \texttt{res}\ \emptyset$, which expresses that the computation does not have any result. Note that the computation without any result refines any computation: $\texttt{res}\ \emptyset \leq m$.

The failing computation `fail`, however, is at the other end of the refinement ordering. It is refined by any program: $m \leq \texttt{fail}$.

Typically, we use stepwise refinement to split up the refinement of a program into a chain of refinements $m \leq m' \leq \ldots \leq m_{spec}$. We call the refinement lemmas that have a specification on the right-hand side *correctness lemmas*. On the lower end of that chain often there is a deterministic program $m = \texttt{return}\ x$, i. e. an NREST program with exactly one result x. On the top end of the chain there is a specification. After proving the individual refinements, we can combine them by transitivity to a single lemma for m. In consequence, we obtain a correctness lemma for m that states that the result x satisfies the specification. The advantage of that approach is that in each individual refinement step one can focus on a specific aspect of the computation.

If we encounter $\top = \texttt{fail}$ or $\bot = \texttt{res}\ \emptyset$ in the refinement chain we have lost precision. Either the concrete program on the lower end has no result, or the specification fails and we have lost all information about the program.

Instead of silently getting programs with no results, we want to signal violated properties and use *assertions* for that purpose. Assertions fail if their condition is not met, and return *unit* otherwise:

$$\texttt{assert}\ P = \texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else fail}$$

They are also used to express preconditions of a program. A Hoare-triple for program m, with precondition P, postcondition Q is written as a refinement condition:

$$m \leq \texttt{do}\ \{\ \texttt{assert}\ P; \texttt{spec}\ Q\ \}$$

Note that the above refinement condition is equivalent to: $P \implies m \leq \texttt{spec}\ Q$. That is, one can pull the assertion of a specification into the premises. This is the way to express preconditions in specifications.

Example 8.1.3. To continue the example above, we now guard $choose'_{spec}$ with an assertion and use it in a safe implementation for *add_elem*:

$$
\begin{array}{l}
choose_{spec}\ S = \texttt{do}\ \{ \\
\quad \texttt{assert}\ (S \neq \emptyset); \\
\quad choose'_{spec}\ S \\
\}
\end{array}
$$

$$
\begin{array}{l}
add\_elem\ (a,\ S) = \texttt{do}\ \{ \\
\quad (s,\ S') \leftarrow choose_{spec}\ S; \\
\quad a' \leftarrow add_{spec}\ a\ s; \\
\quad \texttt{return}\ (a',\ S'); \\
\}
\end{array}
$$

Furthermore, we can now give a specification for the running example of summing the numbers of a set.

$$sum\_set_{spec}\ S = \texttt{do}\ \{\ \texttt{assert}\ (\textit{finite}\ S);\ \texttt{spec}\ (\lambda r.\ r = (\textstyle\sum_{s \in S} s))\ \}$$

Here, we demand that the set we want to sum over is finite.

We use Isabelle/HOL's *if-then-else* to model branching and define a recursion combinator $\texttt{rec} :: ((\alpha \to \alpha\ nres) \to \alpha \to \alpha\ nres) \to \alpha \to \alpha\ nres$ via a fixed-point construction [74], to get a complete set of basic combinators. Then, we can define a derived combinator for a while loop:

$$
\begin{array}{l}
\texttt{while} :: (\alpha \to bool) \to (\alpha \to \alpha\ nres) \to \alpha \to \alpha\ nres \\
\texttt{while}\ b\ c\ s = \texttt{rec}\ (\lambda R\ s.\ \texttt{if}\ b\ s\ \texttt{then do}\ \{\ \texttt{assert}\ (b\ s);\ s \leftarrow (c\ s);\ R\ s\ \} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else return}\ s)\ s
\end{array}
$$

Example 8.1.4. Now we can finally write down the algorithmic idea of the running example:

$$
\begin{array}{l}
sum\_set\ S = \texttt{do}\ \{ \\
\quad s \leftarrow \texttt{return}\ (0,\ S); \\
\quad (a',\ S') \leftarrow \texttt{while}\ (\lambda(a,\ S).\ S \neq \emptyset)\ (\lambda(a,\ S).\ add\_elem\ (a,\ S))\ s; \\
\quad \texttt{return}\ a' \\
\}
\end{array}
$$

The program starts with the parameter set S, draws an element from the set and adds it to the accumulator until the set is empty, and finally returns the accumulator. Here, the tuple $(a,\ S)$ of accumulator and set of remaining numbers constitute the state that is changed during the execution of the `while` loop. It has to be threaded through the loop explicitly.

We now would like to prove that the program terminates and fulfills its specification.

$$sum\_set\ S \leq sum\_set_{spec}\ S$$

Before we turn to how to prove such refinement lemmas, let me introduce the last concept of modeling: data refinement.

8.1.1 Data Refinement

A typical use case of refinement is to implement an *abstract* data type by a *concrete* data type. In our running example, we could implement finite sets of numbers by distinct lists. Towards an executable implementation, instead of abstractly summing over the set of numbers we could iterate over the list and add up the elements.

We define a *refinement relation* R^{set}_{list} between distinct lists and finite sets. A concrete computation m that yields distinct lists then refines an abstract computation m' that yields sets if every possible concrete result is related to a possible abstract result. Formally, $m \leq \Downarrow_D R\ m'$, where the subscript D stands for *data refinement*. The operator $\Downarrow_D$ is defined, for arguments R and m', by the following two rules.

$$\Downarrow_D R\ (\texttt{res}\ X) = \texttt{res}\ (Sup_{x \in X}\ \{c\ .\ (c,\ a) \in R\}) \qquad \Downarrow_D R\ \texttt{fail} = \texttt{fail}$$

Here, we use the supremum to aggregate all abstract results that are related to a concrete result. It essentially is the union over the set of concrete results that are abstracted by all the results of the abstract computation.

Example 8.1.5. For an abstract program $m = \texttt{res}\ \{\ \{3,\ 1,\ 2\}\ \}$ we will straightforwardly have the concretized program $\Downarrow_D R^{set}_{list}\ m = \texttt{res}\ \{[1,\ 2,\ 3],\ [2,\ 3,\ 1],\ \ldots\}$. The refinement relation R^{set}_{list} is formally defined in the following way:

$$(xs,\ S) \in R^{set}_{list} \longleftrightarrow (set\ xs = S \wedge distinct\ xs)$$

where *set xs* is the set of elements in the list *xs*, and the predicate *distinct* characterizes lists that have no duplicates. Note that to *build* this *relation* we used an abstraction function (*set*) and an invariant (*distinct*). Refinement relations often take that form and we have $R^{set}_{list} = br\ set\ distinct$ with $br\ \alpha\ I = \{(c,\ a).\ I\ c \wedge a = \alpha\ c\}$. However, not every set corresponds to a list in this relation, nor does every list implement a set. Infinite sets, like $\{i \mid odd\ i\}$, do not have a corresponding list, and the list $[0,\ 0]$ is not distinct and thus does not implement a set.

The refinement relation R^{set}_{list} describes how the container data structure is refined. Additionally, the data contained in the set could be refined. For simplicity, we do not consider this here and refer to [88, §4.2.1] for a principled approach to combine refinement relations of data and data structures.

Instead, we simply refine the contained numbers by themselves. This is achieved by using the identity relation *Id*, which is denoted by R^{nat}_{nat} for natural numbers. We write $(n,\ n') \in R^{nat}_{nat}$ to express that n refines the the natural number n', which is equivalent to $n = n'$. Data refinement with relation *Id* has no effect: $\Downarrow_D Id\ m = m$.

Example 8.1.6. Using data refinement we can now refine our running example to use lists instead of sets. In Figure 8.1 we have the abstract programs working on sets on

```
1  take_first xs = do {                          choose_spec S = do {                          1
2    assert (xs ≠ []);                             assert (S ≠ ∅);                             2
3    return (hd xs, tl xs)                         spec (λ(s, S'). s ∉ S' ∧ S' ∪ {s} = S)      3
4   }                                            }                                             4
5                                                                                              5
6  add_one2 (b, xs) = do {                       add_one (a, S) = do {                         6
7    (x, xs') ← take_first xs;                     (s, S') ← choose_spec S;                    7
8    b' ← add_spec b x;                            a' ← add_spec a s;                          8
9    return (b', xs')                              return (a', S')                             9
10 }                                             }                                             10
11                                                                                             11
12 sum_set2 xs = do {                            sum_set S = do {                              12
13   t ← return (0, xs);                           s ← return (0, S);                          13
14   (b', xs') ← while (λ(b, xs). xs ≠ [])         (a', S') ← while (λ(a, S). S ≠ ∅)           14
15                (λ(b, xs). add_one2 (b, xs))                  (λ(a, S). add_one (a, S))     15
16                t;                                            s;                            16
17   return b'                                     return a'                                   17
18 }                                             }                                             18
```

(a) **(b)**

Figure 8.1: Two algorithms whose components are in refinements

the right, and their refinements working on lists on the left. In line 3 of the latter, the expressions *hd xs* and *tl xs* return the first element and respectively all but the first element of the list *xs*.

Let list *xs* implement set *S*, i. e. $(xs,\ S) \in R^{set}_{list}$. Then, we would like to establish the following refinements:

$$take\_first\ xs \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ (choose_{spec}\ S)$$
$$one\_add_2\ (a,\ xs) \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ (one\_add\ (a,\ S))$$
$$sum\_set_2\ xs \leq \Downarrow_D R^{nat}_{nat}\ (sum\_set\ S)$$

where the operator $\cdot \times \cdot$ is the product on relations.

In Section 8.1.3, I will present an approach to prove specification refinements, which have a specification on the right-hand side, like the first of the three propositions above. The latter two refinements are between two programs that have the same structure. I will show how to handle them in a structured and automated way in Section 8.1.6.

8.1.2 Notation for Refinement

Like in the last example, we will often see propositions of the form

$$(x,\ x') \in R \implies m\ x \leq \Downarrow_D S\ (m'\ x)$$

It states that m refines m' w. r. t. relation R for the arguments and S for the result. Some of those propositions may include additional preconditions. To write them more conveniently, we use the following notation[1]:

$$(m,\ m') \in [\lambda x\ x'.\ pre\ x\ x']\ R \to S \\ = \forall x\ x'.\ pre\ x\ x' \implies (x,\ x') \in R \implies m\ x \leq \Downarrow_D S\ (m'\ x')$$

If the precondition is always true, we just write $(m,\ m') \in R \to S$. For the sake of readability, we will identify curried and uncurried function and write

$$(m,\ m') \in R_1 \to \ldots \to R_n \to S$$

for programs with n arguments that are refined by $R_1, \ldots, R_n$.

The above form of those propositions is called the *parametric form*. It brings to mind relational parametricity by Wadler [128].

Example 8.1.7. Using this notation we can rephrase the refinements of our example in the following way:

$$(take\_first,\ choose_{spec}) \in R^{set}_{list} \to (R^{nat}_{nat} \times R^{set}_{list}) \\ (one\_add_2,\ one\_add) \in R^{nat}_{nat} \to R^{set}_{list} \to (R^{nat}_{nat} \times R^{set}_{list}) \\ (sum\_set_2,\ sum\_set) \in R^{set}_{list} \to R^{nat}_{nat}$$

The last one reads: if the parameters are related by R^{set}_{list} then the result of sum_set_2 refines the result of sum_set w. r. t. relation R^{nat}_{nat}.

Also the correctness lemma from Example 8.1.4 can be phrased in the more convenient notation. We have to add the trivial data refinements Id and R^{nat}_{nat}.

$$(sum\_set,\ sum\_set_{spec}) \in Id \to R^{nat}_{nat}$$

Before we turn to proving those refinements, let us put them together and obtain the correctness for sum_set_2. The data refinement lemma for sum_set_2 above can be combined with the correctness lemma for sum_set. We first combine the theorems concretely and then see how refinements in the parametric form can be combined in general.

By transitivity and monotonicity of data refinement we obtain:

$$(xs,\ S) \in R^{set}_{list} \implies sum\_set_2\ xs \leq \Downarrow_D Id\ (sum\_set_{spec}\ S)$$

Now, we can unfold sum_set_{spec} and R^{set}_{list}, replace S with $set\ xs$ and thus remove the precondition *finite* $(set\ xs)$, which we get from sum_set_{spec}. Finally, we massage the lemma into the following form.

$$[\![\ distinct\ xs \wedge sorted\ xs\ ]\!] \implies sum\_set_2\ xs \leq \texttt{return}\ (\textstyle\sum_{x \in set\ xs} x)$$

It is remarkable that the set S can be discarded from the final lemma. More interestingly, if we have a close look at the program sum_set_2, we see, that it actually also sums all elements from the list even if it is not distinct. That means we have chosen the abstraction with a set — which determines at least the distinctness property —

[1] This notation was first described in [82, §2.2].

too coarse to capture all program behavior of sum_set_2.

We have to keep in mind that the choice of the specification at the top end as well as all the data abstractions along the refinement chain restrict what properties we can show for the program at the bottom end.

At this point we have a deterministic program together with a proof for termination and a correctness lemma. We could now use the back ends described in Section 7.2 to extract executable program code.

In general, we would like to compose two refinements in parametric form. Consider three programs m_3, m_2 and m_1 with refinement lemmas we can prove:

$$[\![(m_3,\ m_2) \in R_2 \to S_2;\ (m_2,\ m_1) \in R_1 \to S_1]\!] \\ \Longrightarrow\ (m_3,\ m_1) \in (R_2\ O\ R_1) \to (S_2\ O\ S_1)$$

Here, $R_2\ O\ R_1$ is the composition of relations. The identity relation is the neutral element for that operation: $Id\ O\ R = R\ O\ Id = R$.

Example 8.1.8. In our example combining the refinement lemmas for sum_set_2 and sum_set we obtain:

$$(sum\_set_2,\ sum\_set_{spec}) \in (R^{set}_{list}\ O\ Id) \to (R^{nat}_{nat}\ O\ R^{nat}_{nat})$$

It simplifies to the same correctness lemma for sum_set_2 as before.

This completes the exposition on how to model computation in *nres*, set up the refinement chain, and collapse it into a final correctness lemma. Now we turn to how we actually can prove the refinement lemmas.

8.1.3 Specification Refinement

A recurring pattern are refinement lemmas with a specification on the right-hand side. Those lemmas are called *correctness lemmas* or *specification refinements*. In this section I present a technique for proving those. We want to prove that a compound program m refines some specification that is guarded with a precondition and may involve a data refinement:

$$m \le \Downarrow_D R\ (\texttt{do}\ \{\ \texttt{assert}\ P;\ \texttt{spec}\ Q\ \})$$

We can always first pull the precondition to the premises, and push the data refinement into the specification. The following two rules formalize that intuition.

$$(P \Longrightarrow m \le \Downarrow_D R\ (\texttt{spec}\ Q)) \Longrightarrow m \le \Downarrow_D R\ \texttt{do}\ \{\ \texttt{assert}\ P;\ \texttt{spec}\ Q\ \} \\ (m \le \texttt{spec}\ (\lambda c.\ \exists a.\ (c,a) \in R \wedge Q\ a)\ ) \Longrightarrow m \le \Downarrow_D R\ (\texttt{spec}\ Q)$$

Because the general from can be transformed in that way, in the following we assume to work on lemmas of the form $m \le \texttt{spec}\ Q$. We can come up with refinement rules for the basic combinators:

$$Q\ x \implies \texttt{return}\ x \leq \texttt{spec}\ Q$$
$$P \wedge (P \implies Q\ ()) \implies \texttt{assert}\ P \leq \texttt{spec}\ Q$$
$$(\forall x.\ P\ x \implies Q\ x) \implies \texttt{spec}\ P \leq \texttt{spec}\ Q$$

The latter rule together with transitivity of the refinement ordering gives us a *consequence rule*:

$$m \leq \texttt{spec}\ P \implies (\forall x.\ P\ x \implies Q\ x) \implies m \leq \texttt{spec}\ Q$$

It can be used if we already have a specification refinement lemma for some program m, and want to weaken the post condition P to Q.

In order to combine programs, we need a rule for sequential composition. We can prove the following rule for the **bind** operator:

$$m \leq \texttt{spec}\ (\lambda x.\ f\ x \leq \texttt{spec}\ Q) \implies \texttt{bind}\ m\ f \leq \texttt{spec}\ Q$$

It reads: the compound program refines specification Q if the computation m does not fail and only produces results x, which run on f fulfill specification Q. This allows for a weakest precondition calculus, and the rule for **bind** resembles the well-known **bind** rule for weakest preconditions.

Example 8.1.9. Now we can tackle the refinement lemma of *take_first* and $choose_{spec}$. Remember we can assume that xs refines S. Then, the goal is:

$$(xs,\ S) \in R_{list}^{set} \implies take\_first\ xs \leq \Downarrow_D(Id \times R_{list}^{set})\ (choose_{spec}\ S)$$

Let us follow that proof step by step. First, we unfold the definitions of the specification, pull the precondition into the premises, and push the data refinement into the specification. We obtain the following goal:

$$\begin{aligned}(xs,\ S) \in R_{list}^{set} &\implies S \neq \emptyset \\ &\implies take\_first\ xs \leq \texttt{spec}\ (\lambda(x,\ xs).\ \exists(s,\ S').\ ((x,\ xs),\ (s,S')) \in (Id \times R_{list}^{set}) \\ &\qquad\qquad \wedge\ \ s \notin S' \wedge S' \cup \{s\} = S)\end{aligned}$$

Let us write $\texttt{spec}\ Q_{cs}$ for the right-hand side of the current goal, and forget the premises for now. Next, we unfold the definition of the program *take_first* and apply the rules from above. As the outer most operator is **bind** we use its rule and obtain the goal:

$$\texttt{assert}\ (xs \neq []) \leq \texttt{spec}\ (\lambda\_.\ \texttt{return}\ (hd\ xs,\ tl\ xs) \leq \texttt{spec}\ Q_{cs})$$

After applying the rule for **assert** and the rule for **return** we are left with two goals, which do not contain any monadic programs:

$$(xs,\ S) \in R_{list}^{set} \implies S \neq \emptyset \implies xs \neq []$$
$$(xs,\ S) \in R_{list}^{set} \implies S \neq \emptyset \implies xs \neq [] \implies Q_{cs}\ (hd\ xs,\ tl\ xs)$$

Proving those goals is in the realm of standard Isabelle/HOL tactics. Either by providing hand-crafted lemmas for R_{list}^{set}, or simply unfolding its definition and hammering through.

In that way, we can automatically reduce *specification refinements* of monadic programs to showing verification conditions. The tool used for that purpose is called

refine_vcg. It contains the simplification rules and the introduction rules for all the combinators. Typically, it suffices to unfold the specification and the program in question. Then, the bind rule always pulls the first command to the front, for which an introduction rule must exist. If that first command c in the current goal $c \leq \texttt{spec}\ Q$ is not one of the standard combinators, the tool requires a rule of the following form $S \implies c \leq \texttt{spec}\ P$. Then the consequence rule can be used and the goal is transformed into $S \wedge (\forall x.\ P\ x \implies Q\ x)$.

Until now we only considered straight-line programs. Let us see how to prove a specification refinement for a program that involves a while loop. Consider *sum_set* from Figure 8.1b. We want to prove the lemma $sum\_set\ S \leq sum\_set_{spec}\ S$.

For the combinator `while` we will not be able to prove the refinement automatically. A standard approach is to indicate an invariant I that is preserved throughout the execution of the loop and to specify a well-founded relation on the states of the loop iterations to ensure termination. The following rule for the `while` combinator allows that:

```
1   ⟦ wf R; I s;
2     ∀s. I s ⟹ b s ⟹ c s ≤ spec (λs'. I s' ∧ (s', s) ∈ R);
3     ∀s. I s ⟹ ¬ b s ⟹ Q s ⟧
4     ⟹ while b c s ≤ spec Q
```

In order to prove that a `while` loop terminates and fulfills a specification we can provide an invariant and a well-founded relation and prove the following four premises. First, the relation R needs to be well-founded. Second, the invariant has to hold initially. Third, the invariant must be preserved by the loop body, and the state needs to be strictly decreasing along the well-founded relation R. Finally, the invariant needs to imply the postcondition when exiting the loop.

Example 8.1.10. Let us execute *refine_vcg* for the goal $sum\_set\ S \leq sum\_set_{spec}\ S$. The definition of program *sum_set* is in Figure 8.1b, and the specification sum_set_{spec} is defined in Example 8.1.3.

Routinely, we first unfold the specification and obtain the precondition *finite* S as a premise. Then we unfold the program and apply the standard rules for `bind` and `return` until the `while` combinator is the top-most symbol. At that point the goal looks as follows:

$$\begin{aligned}&\texttt{while}\ (\lambda(a,\ S).\ S \neq \emptyset)\\ &\qquad (\lambda(a,\ S).\ add\_one\ (a,\ S))\ (0,\ S)\\ &\leq \texttt{spec}\ (\lambda(a',\ S').\ \texttt{return}\ a' \leq \texttt{spec}\ (\lambda r.\ r = \textstyle\sum_{s\in S} s))\end{aligned}$$

The well-founded relation R is defined such that it makes sure that the cardinality of the set S decreases. The invariant is $I\ (a',\ S') \longleftrightarrow S' \subseteq S \wedge a' + \sum_{s\in S'} s = \sum_{s\in S} s$. Applying the above rule, we have to prove the four goals that stem from the `while` rule. The goals in line 1 make sure that R is well-founded — we ignore it in this presentation — and that the invariant holds initially. The second line involves proving that the loop body does the right thing, i. e. it preserves the invariant and follows the well-founded relation. After unfolding the definition of *add_one* the tactic *refine_vcg* routinely

converts the goal into a verification condition not involving monadic programs. The final goal stemming from line 3 expresses that the invariant implies the postcondition in case the computation exits the loop. After applying *refine_vcg* one we obtain the final goal:

$$I\,(a',\,S') \implies S' = \emptyset \implies a' = \sum_{s \in S} s$$

This lemma can be proved by unfolding the invariant I and applying Isabelle/HOL's automatic tactics.

Provided with the invariant and the well-founded relation that ensures termination, the tactic *refine_vcg* automatically traverses the program and reduces the specification refinement to goals that are free from monadic programs. Those can then be solved with standard Isabelle/HOL tactics or interactive proof.

The tactic *refine_vcg* can solve refinement lemmas of a very restricted form. In the following we will introduce pointwise reasoning which helps to automatically prove general refinements between monadic programs.

8.1.4 Pointwise Reasoning

To prove equality and refinement of *nres* programs we introduce pointwise reasoning.[2] As *nres* programs are in essence sets of results, for showing a refinement $\texttt{res}\ X \leq \texttt{res}\ Y$ we need to show the set inclusion $X \subseteq Y$. This is in turn equivalent to the *pointwise* first-order goal $\forall x \in X.\ x \in Y$.

First, we define the predicate *nofail* to characterize non-failing computations, and *inres* m x to signal that x is a valid result of m:

$$\mathit{nofail}\ m = (m \neq \texttt{fail})$$

$$\mathit{inres}\ \texttt{fail}\ x = \mathit{True}$$
$$\mathit{inres}\ (\texttt{res}\ X)\ x = (x \in X)$$

An alternative way of writing this is $\mathit{inres}\ m\ x \longleftrightarrow (\texttt{return}\ x \leq m)$.

Those definitions help us to prove equalities and refinements of programs by reducing them to proof obligations which typically can be discharged by automatic first-order provers:

$$(\mathit{nofail}\ m \implies \mathit{nofail}\ m') \wedge (\forall x.\ \mathit{inres}\ m\ x \implies \mathit{inres}\ m'\ x) \implies m \leq m'$$

It spells out the intuition we already had about the refinement ordering: m refines m', when it only fails if m' also does, and if every result of m is also an result of m'.

As the ordering is antisymmetric, we can also reduce equivalences to pointwise proof obligations. Two computations are equal if they either both fail or have the same results:

[2] That is not a new idea: setup for pointwise reasoning was already implemented by Lammich for *nres*. We describe it here, because it was never explained in any publication, and we will build reasoning infrastructure for the resource types described in the next sections upon those ideas.

$(\mathit{nofail}\ m = \mathit{nofail}\ m') \wedge (\forall x.\ \mathit{inres}\ m\ x = \mathit{inres}\ m'\ x) \implies m = m'$

We typically compare programs that are composed with the combinators introduced earlier. We can prove simplification lemmas for *inres* for different combinators:

$\mathit{inres}\ (\texttt{res}\ X)\ x = x \in X$
$\mathit{inres}\ (\texttt{return}\ x)\ y = (x{=}y)$
$\mathit{inres}\ (\texttt{spec}\ P)\ x = P\ x$
$\mathit{inres}\ (\texttt{assert}\ P)\ () = P$
$\mathit{inres}\ (\texttt{if}\ P\ \texttt{then}\ m_1\ \texttt{else}\ m_2)\ x = (\texttt{if}\ P\ \texttt{then}\ \mathit{inres}\ m_1\ x\ \texttt{else}\ \mathit{inres}\ m_2\ x)$

Specifically, for the supremum operator we get the following simplification rule, which still is first-order:

$\mathit{inres}\ (\mathit{Sup}\ S)\ x = (\exists m{\in}S.\ \mathit{inres}\ m\ x)$

That is, x is a valid result of an aggregation of several computations S, if there is a computation m in that set which produces x. In the case where S is empty, the supremum gets $\texttt{res}\ \emptyset$. Consequently, the left-hand side is false, as is the right-hand side. We use the supremum in the definition of bind as well as in the definition of data refinement. The *inres* rule for supremum is used in the proofs for the *inres* rules for those two concepts.

With the rules above established, we can prove the rule for $\texttt{bind}$ automatically:

$\mathit{inres}\ (\texttt{bind}\ m\ f)\ y = (\mathit{nofail}\ m \implies (\exists x.\ \mathit{inres}\ m\ x \wedge \mathit{inres}\ (f\ x)\ y))$

It is instructive to follow that proof in detail. For the case that m does not fail, i. e. $m = \texttt{res}\ X$, let us simplify both sides with the rules from above:

$$\begin{aligned}\mathit{inres}\ (\texttt{bind}\ (\texttt{res}\ X)\ f)\ y &= \mathit{inres}\ (\mathit{Sup}_{x\in X}\ f\ x)\ y\\ &= (\exists m{\in}\{f\ x \mid x \in X\}.\ \mathit{inres}\ m\ y)\\ &= (\exists x{\in}X.\ \mathit{inres}\ (f\ x)\ y)\\ &= (\exists x.\ x \in X \wedge \mathit{inres}\ (f\ x)\ y))\end{aligned}$$

$$\begin{aligned}&(\mathit{nofail}\ (\texttt{res}\ X) \implies (\exists x.\ \mathit{inres}\ (\texttt{res}\ X)\ x \wedge \mathit{inres}\ (f\ x)\ y))\\ &\quad= (\exists x.\ \mathit{inres}\ (\texttt{res}\ X)\ x \wedge \mathit{inres}\ (f\ x)\ y))\\ &\quad= (\exists x.\ \mathit{inres}\ (\texttt{res}\ X)\ x \wedge \mathit{inres}\ (f\ x)\ y))\\ &\quad= (\exists x.\ x \in X \wedge \mathit{inres}\ (f\ x)\ y))\end{aligned}$$

Many lemmas involving *nofail*, *inres* and arbitrary programs composed of the above combinators can be solved by the pointwise approach. First the simplification rules are applied to get rid of the monadic programs, then strong first-order automation is applied to prove the lemmas automatically. It would be interesting to characterize the subset of refinements that can be solved that way. At least they suffice for proving the monad laws and monotonicity lemmas.

Before we turn to those let us phrase pointwise reasoning rules for data refinement:

$\mathit{nofail}\ (\Downarrow_D R\ m) = \mathit{nofail}\ m$
$\mathit{inres}\ (\Downarrow_D R\ m)\ x = (\mathit{nofail}\ m \implies (\exists x'.\ (x,\ x') \in R \wedge \mathit{inres}\ m\ x'))$

A concretized program $\Downarrow_D R\ m$ fails exactly if the abstract program m fails, and x is the result of a $\Downarrow_D R\ m$ program, if there is a result x' of the abstract program that abstracts the concrete result x.

In general, with pointwise reasoning we can convert a refinement proof obligation into a first-order formula, which can be solved using standard Isabelle/HOL tools. In practice, this works well for refinements that do not involve recursion or looping constructs.

Monad Laws With the pointwise reasoning setup at hand, we can automatically prove the monad laws [127]:

$$\texttt{bind}\ (\texttt{return}\ x)\ f = f\ x$$
$$\texttt{bind}\ m\ (\lambda x.\ \texttt{return}\ x) = m$$
$$\texttt{bind}\ (\texttt{bind}\ m\ f)\ g = \texttt{bind}\ m\ (\lambda x.\ \texttt{bind}\ (f\ x)\ g)$$

The combinator `return` is a left and right-identity for `bind`, and `bind` is associative. This now gives us the formal justification to use do-notation for writing down monadic programs in a declarative style (cf. Figure 8.1).

8.1.5 Monotonicity Reasoning

In order to prove refinement between two programs that have the same structure we can use monotonicity reasoning.

Example 8.1.11. Consider the programs in Figure 8.1 and the refinements we would like to establish for Example 8.1.7. We have already seen how we can prove the specification refinement $(take\_first,\ choose_{spec}) \in R^{set}_{list} \to (R^{nat}_{nat} \times R^{set}_{list})$.

Now we want to prove the refinement between add_one_2 and add_one, and sum_set_2 and sum_set. We observe that both pairs of programs have the same structure. Only some data and some operations are refined.

In order to refine a compound program, we would like to first, refine its components, and then combine the refinements of the parts to obtain a refinement of the compound program. To that end, we need some monotonicity rules for the combinators we use.

In particular for sequential composition we can prove the following rule using the pointwise reasoning:

1 $\quad [\![\ m \le \Downarrow_D R'\ m';$
2 $\quad\quad \forall x\ x'.\ (x,\ x') \in R' \wedge nofail\ m' \wedge inres\ m'\ x' \implies f\ x \le \Downarrow_D R\ (f'\ x')\ ]\!]$
3 $\quad\quad \implies \texttt{bind}\ m\ f \le \Downarrow_D R\ (\texttt{bind}\ m'\ f')$

That is, if we can refine program m' with data refinement relation R' (line 1) and for all its concrete results we have a refinement for f' with refinement relation R (line 2), we also have a refinement for the compound computation with data refinement relation R (line 3). The *inres* precondition in the second premise is used to transport information about the intermediate variable x from the execution of m.

For `return` we have the following rule:

$$(x,\ y) \in R \implies \texttt{return}\ x \leq \Downarrow_D R\ (\texttt{return}\ y)$$

As observed before, we have $\Downarrow_D Id\ m = m$ and thus $m \leq \Downarrow_D Id\ m$ is correct.

Example 8.1.12. Let us execute this process for the refinement of *add_one*. This will be quite verbose, but bear with me: I think it is insightful to see this process at least once in full detail.

We want to prove the following refinement lemma:

$$one\_add_2\ (b,\ xs) \leq \Downarrow_D(Id \times R^{set}_{list})\ one\_add\ (a,\ S)$$

Remember that we have the preconditions $(b,\ a) \in R^{nat}_{nat}$ and $(xs,\ S) \in R^{set}_{list}$. For the components of the programs we have the following two rules:

$$(xs,\ S) \in R^{set}_{list} \implies take\_first\ xs \leq \Downarrow_D(Id \times R^{set}_{list})\ (choose_{spec}\ S)$$
$$(x,\ x') \in Id \wedge (y,\ y') \in Id \implies add_{spec}\ x\ y \leq \Downarrow_D Id\ (add_{spec}\ x'\ y')\}.$$

We apply the monotonicity approach for the refinement lemma. First, we apply the monotonicity rule for **bind**. The first goal will be $take\_first\ xs \leq \Downarrow_D ?R\ (choose_{spec}\ S)$, where $?R$ is not known. But looking into our store of refinement rules, we only have the one for *take_first* mentioned above. We can match R with $(R^{nat}_{nat} \times R^{set}_{list})$ and prove the goal using the rule. Note that for applying that rule we did not have to prove that the precondition in $choose_{spec}\ S$, i. e. $S \neq \emptyset$. Instead, we get the precondition from the abstract program "for free". It must be proven further up the refinement chain and can be safely assumed here. This leaves us with the second goal:

$$\begin{aligned}&[\![\ ((x,\ xs'),\ (s,\ S')) \in (R^{nat}_{nat} \times R^{set}_{list});\\ &\quad nofail\ (choose_{spec}\ S);\ inres\ (choose_{spec}\ S)\ (s,\ S')\ ]\!]\\ &\implies \texttt{do}\ \{\ b' \leftarrow add_{spec}\ b\ x;\ \texttt{return}\ (b',\ xs')\ \}\\ &\qquad \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ (\texttt{do}\ \{\ a' \leftarrow add_{spec}\ a\ s;\ \texttt{return}\ (a',\ S')\ \})\end{aligned}$$

Here, note that we have to retrieve the information how the intermediate results s and S' are connected to the input set S from the premises $inres\ (choose_{spec}\ S)\ x'$ and $nofail\ (choose_{spec}\ S)$. For that, we need to unfold the definition of the specification and then simplify with the simplification rules for *inres*. In the end we obtain the necessary information $S \neq \emptyset \wedge s \notin S' \wedge S' \cup \{s\} = S$. The goal simplifies to:

$$\begin{aligned}&[\![\ ((x,\ xs'),\ (s,\ S')) \in (R^{nat}_{nat} \times R^{set}_{list});\\ &\quad S \neq \emptyset;\ s \notin S';\ S' \cup \{s\} = S\ ]\!]\\ &\implies \texttt{do}\ \{\ b' \leftarrow add_{spec}\ b\ x;\ \texttt{return}\ (b',\ xs')\ \}\\ &\qquad \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ (\texttt{do}\ \{\ a' \leftarrow add_{spec}\ a\ s;\ \texttt{return}\ (a',\ S')\ \})\end{aligned}$$

Now we apply the **bind** rule again, use the rule for add_{spec}, and obtain:

$$\begin{aligned}&[\![\ ((x,\ xs'),\ (s,\ S')) \in (R^{nat}_{nat} \times R^{set}_{list});\ (b',\ a') \in R^{nat}_{nat};\\ &\quad S \neq \emptyset;\quad s \notin S';\quad S' \cup \{s\} = S;\\ &\quad nofail\ (add_{spec}\ a\ s);\ inres\ (add_{spec}\ a\ s)\ a'\ ]\!]\\ &\implies \texttt{return}\ (b',\ xs') \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ (\texttt{return}\ (a',\ S'))\end{aligned}$$

In a final step we again retrieve the information from the *inres* term, and use the monotonicity rule for **return**. The final goal is:

$$[\![\ ((x,\ xs'),\ (s,\ S')) \in (R^{nat}_{nat} \times R^{set}_{list});\ (b',\ a') \in R^{nat}_{nat};\ S \neq \emptyset;\ s \notin S';\ S' \cup \{s\} = S;\ a' = a + s\ ]\!] \Longrightarrow\ ((b',\ xs'),\ (a',\ S')) \in (R^{nat}_{nat} \times R^{set}_{list})$$

The goal again does not contain any monadic programs and can be discharged with standard tactics. Observe that actually the information from the *inres* is not really needed in this example. This is the case because none of the used refinement lemmas for the components had a precondition. In general, this is not the case.

We have now established the proof for the refinement lemma:

$$[\![\ (b,\ a) \in R^{nat}_{nat};\ (xs,\ S) \in R^{set}_{list}\ ]\!] \Longrightarrow\ one\_add_2\ (b,\ xs) \leq \Downarrow_D(R^{nat}_{nat} \times R^{set}_{list})\ one\_add\ (a,\ S)$$

Because this notation is quite verbose, we can use the notation introduced earlier and write:

$$(one\_add_2,\ one\_add) \in R^{nat}_{nat} \to R^{set}_{list} \to (R^{nat}_{nat} \times R^{set}_{list})$$

It expresses that if the parameters are data refined with refinement relation R^{set}_{list} and R^{set}_{list} respectively, then the result of one_add_2 refines the result of *one_add* via the refinement relation $R^{nat}_{nat} \times R^{set}_{list}$. This is exactly the goal we mentioned in Example 8.1.7.

We can also prove similar lemmas for other higher order combinators, like `rec`:

$$[\![\ (x,\ x') \in R;\ \forall x\ x'.\ [\![\ (x,\ x') \in R\ ]\!] \Longrightarrow\ b\ x = b'\ x';\ \forall x\ x'.\ [\![\ (x,\ x') \in R;\ b\ x;\ b'\ x'\ ]\!] \Longrightarrow\ f\ x \leq \Downarrow_D R\ \ (f'\ x')\ ]\!] \Longrightarrow\ \texttt{while}\ b\ f\ x \leq \Downarrow_D R\ (\texttt{while}\ b'\ f'\ x')$$

To prove that a concrete `while` loop refines an abstract one with a data refinement R, we have to show three things. First, the initial states must be in the data refinement relation. Second, if the abstract and concrete states are related with R the guards coincide. And finally, if the loop is entered and the parameters are related by R, the results of the loop body are related by the refinement relation R.

Example 8.1.13. It is instructive to also look at the refinement proof for *sum_list* and *sum_set*. We want to prove: $(sum\_list,\ sum\_set) \in R^{set}_{list} \to R^{nat}_{nat}$. We again observe that the structure is the same, so we can start applying the monotonicity rule for bind and use the standard monotonicity rule for return. We obtain the goal where the while operator is the top most combinator:

$$[\![\ (t,\ s) \in (Id \times R^{set}_{list});\ nofail\ (\texttt{return}\ (0,\ S));\ inres\ (\texttt{return}\ (0,\ S))\ s\ ]\!] \Longrightarrow\ \texttt{while}\ (\lambda(b,\ xs).\ xs \neq [])\ (\lambda(b,\ xs).\ add\_one_2\ (b,\ xs))\ t \leq \Downarrow_D(Id \times R^{set}_{list})\ (\texttt{while}\ (\lambda(b,\ xs).\ S \neq \emptyset)\ (\lambda(a,\ S).\ add\_one\ (a,\ S))\ s)$$

After applying the rule we need to show its three premises hold. The refinement relation holds initially. Then, if xs and S are related they both are empty at the same time and thus the guards coincide. Finally, for the loop body, we essentially have to show $(add\_one_2,\ add\_one) \in R^{set}_{list} \to R^{set}_{list}) \to (R^{set}_{list} \times R^{set}_{list})$, which we already did in Exercise 8.1.12. This finishes the proof.

8.1.6 Lockstep Refinement

We often refine a compound program by refining some of its components. We have just seen two examples. Now I present the general form.

Let A and C be two structurally equal programs (i. e., they have the same structure of combinators *if*, *rec*, *bind*, etc.), and let A_i and C_i be the pairs of corresponding basic components, for $i \in \{1,\ldots,n\}$. Provided with refinement lemmas $(C_i,\ A_i) \in [\Phi_i]\ R_i' \to R_i$ for each of those pairs,[3] an automatic procedure walks through the program and establishes a refinement $(C,\ A) \in R_0 \to R_n$. Note that the data refinements R_i can be different for each component i, and R_0 relates the parameters of A and C. This process generates verification conditions for ensuring the preconditions Φ_i, which can be discharged automatically or via interactive proof. To combine the refinement lemmas of the components, we employ the monotonicity lemmas for the combinators, as described above. The tactic *refine_rcg* automates that process.

In our example, the refinement lemmas for the components had no preconditions, because they were encoded as assertions in the abstract programs. This simplifies the automatic process, but shifts proof burden to the refinements of the components.

8.1.7 How to Structure Verifications Through Refinement

How can the stepwise refinement approach be used to structure an algorithm verification? A typical approach to verify some operation *prog* is to start by stating a *specification* $prog_{spec}$ = `do { assert` P; `spec` Q `}` with a precondition P and postcondition Q. Then, one can come up with an abstract algorithm $prog_1$ that should implement that specification. That algorithm consists of control flow structures (like conditional branches, *while* loops, recursion combinator, *bind*, and *return*) and other operations (e. g. A_1) that either are specifications themselves or already come with a refinement lemma w. r. t. some specification (A_{spec}). The first lemma to show is $prog_1 \leq prog_{spec}$. Lemmas of that form are called *specification refinements*. They can be discharged with the verification condition generator *refine_vcg* as described in Section 8.1.3. This kind of proof involves proving the correctness of some algorithmic idea at the right level of abstraction such that implementation details do not distract from the main argument.

Typically the subprograms A_i specify only the necessary information for the correctness argument of $prog_1$. Implementation details irrelevant for the correctness proof should be left out. For example, a subprogram might specify to return a non-empty subset of some given set, but does not specify how many or which elements to return, or what implementation for sets to use.

Once the correctness of the algorithmic idea is established, one carries on refining the components. For example, we would like to choose the size of the set for A_1 and come up with a refinement A_2 that returns a singleton subset. Establishing the refinement $A_2 \leq A_1$ can be tackled with the same methods. Assume we proved that fact for all the sub operations in $prog_1$. By replacing the abstract operations with the concrete

[3] The refinement relations R_i' and R_i relate the parameters and respectively the result of those components.

ones, we obtain another program $prog_2$ with the same structure. Now we can use the automation *refine_rcg* for *lockstep refinement* together with the refinement lemmas for the components to obtain the refinement $prog_2 \leq prog_1$.

We can carry on with that approach, also using *data refinement*, for example, choosing the implementation of red-black trees (with refinement relation R^{set}_{rbtree})for the set data structure ($prog_3 \leq \Downarrow_D R^{set}_{rbtree}\ prog_2$) yielding a new program $prog_3$ and proving the refinement lemma with the verification condition generator for lockstep refinement.

In the end, we can use transitivity of the refinement relation to collapse the refinement chain and obtain the result that the third iteration of the algorithm refines the initial specification: $prog_3 \leq \Downarrow_D R_{rbtree}\ prog_{spec}$

8.1.8 Recap

I have presented IRF's nondeterministic result monad *nres*. We have seen how it can be used to abstractly model computation and use a stepwise refinement approach to prove termination and functional correctness. I have introduced the main mechanisms to reason about *nres* programs. This lays the ground for the extension to also reasoning about the resource consumption of such programs. In the rest of this part I will show how the IRF can be extended that way. In the rest of this chapter I will present the extension of *nres* that allows reasoning about resource consumption.

8.2 The General NREST Monad

In this section I will introduce the general NREST monad. It subsumes two iterations of the NREST monad and the original one. I will first state the motivation and which design principle we would like to express. Then, I will show how the concepts and the automatic tactics from *nres* can be extended to reason about resources, and will equip this with some examples.

We want to extend *nres* and keep the modeling of nondeterministic computation, which proved efficient for the stepwise refinement approach. Furthermore, we need the dedicated element `fail` to model nontermination and failing assertions. We add one more design feature.

We also want to model bounds on the resource cost of computation. For nondeterministic computation a result can be reached over several computation paths. We aggregate the respective resource cost of a result into one value by using the supremum over the cost of all paths that reach that result. We discuss alternatives at the end of this chapter. This design choice leads to a natural refinement ordering: a program refines another program if it computes a subset of results that consume less resources.

We now formalize the above intuition:

$(\alpha, \gamma)\ NREST = \texttt{fail} \mid \texttt{res}\ (\alpha \to \gamma\ option)$

A computation is either `fail`, or `res` M, where M is now a partial function from possible results to resources.

The NREST-monad is a conservative extension of the *nres* monad. If we set γ to the type *unit* that only contains the element () we obtain the type (α, *unit*) *NREST*. We call this type NREST-bool. As *unit option* is isomorphic to *bool*, the partial function $\alpha \to \gamma$ *option* is isomorphic to the type α *set*. Finally, NREST-bool is isomorphic to *nres*. In the following you can always think in that instance in order to guide your intuition and connect it to the concepts known from the previous section. We will also phrase some of the above examples in NREST-bool to introduce the notation. We now will lift the known combinators and introduce new ones that involve resources.

We can lift any ordering on γ to the *refinement ordering* on NREST, by first lifting the ordering to *option* with *None* as the bottom element, then pointwise to functions and finally to (α, γ) *NREST*, setting `fail` as the top element. The *refinement ordering* corresponds to the intuition of refinement: $m \leq m'$ reads as *m refines m'*, i. e., m has less possible results than m', computed with less resources.

Example 8.2.1. For a simple example, let us revisit Example 8.1.1 from *nres* and see how we would model it in NREST-bool. Consider a computation that removes an element of the set {*1, 2, 3*}. That computation is expressed in the following way:

$$m' = \texttt{res}\ [(1, \{2, 3\}) \mapsto (), (2, \{1, 3\}) \mapsto (), (3, \{1, 2\}) \mapsto ()]$$

In general $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ has type $\alpha \to \beta$ *option* and denotes a partial function that maps $x_i :: \alpha$ to $y_i :: \beta$. The element () is the only inhabitant of the type *unit*. The computation m' essentially represents a set of possible results. It again can serve as a *specification* and can be refined by more concrete computations. For example, computation $m = \texttt{res}\ [\ (1, \{2, 3\}) \mapsto ()]$ only produces one of the possible results. Thus m refines m'. This is not very thrilling until now, but bear with me. The examples of NREST-bool should introduce you to the right notation, but you can already think about other instances that measure resources with a number for example.

In general, we require the resources γ to have a complete lattice structure, such that we can form suprema over the (possibly infinitely many) paths that lead to the same result. Moreover, when sequentially composing computations, we need to add up the resources. This naturally leads to a monoid structure (γ, +, *0*), where *0*, intuitively, stands for no resources. We call a type γ *resource type*, if it forms a complete lattice and has a monoid structure.

In order to conveniently model actual computations, we define some combinators for NREST. We define `spec` P T to be the computation of any result r that satisfies $P\ r$ using $T\ r$ resources: `spec` P T = `res` (λr. `if` $P\ r$ `then` *Some* ($T\ r$) `else` *None*). By abuse of notation, we write `spec` x T for `spec` (λx. $r = x$) ($\lambda\_$. T). Furthermore, we define `return` x = `res` $[x \mapsto 0]$ to compute the single result x without using any resources.

Now we will add a combinator that actually uses resources. The `elapse` m t combinator adds the (constant) resources t to all results of m:

```
elapse :: (α, γ) NREST → γ → (α, γ) NREST
elapse fail t = fail
```

$$\texttt{elapse}\ (\texttt{res}\ M)\ t = \texttt{res}\ (\lambda x.\ \texttt{case}\ M\ x\ \texttt{of}\ None \Rightarrow None \mid Some\ t' \Rightarrow Some\ (t + t'))$$

If the computation m fails, also $\texttt{elapse}\ m\ t$ fails. Otherwise, the partial function gets t added to the resource cost of every possible result.

The combinator $\texttt{bind}\ m\ f$ models the sequential composition of computation m and f, where f may depend on the result of m. It extends the original combinator by propagating the costs of the first computation.

$$\texttt{bind} :: (\alpha, \gamma)\ NREST \to (\alpha \to (\beta, \gamma)\ NREST) \to (\beta, \gamma)\ NREST$$
$$\texttt{bind}\ \texttt{fail}\ f = \texttt{fail}$$
$$\texttt{bind}\ (\texttt{res}\ M)\ f = Sup\ \{\texttt{elapse}\ (f\ x)\ t \mid x\ t.\ M\ x = Some\ t\}$$

If the first computation m fails, then also the sequential composition fails. Otherwise, we consider all possible results x with resources t of m, invoke $f\ x$, and add the cost t for computing x to the results of $f\ x$. The supremum aggregates the cases where f yields the same result, via different intermediate results of m. It also makes the whole expression fail if one of the reachable $f\ x$ fails.

For writing larger programs conveniently, we use monadic do-notation as for *nres*. We will later show the monad laws for the instances we will use.

Assertions are used and defined as before: they fail if their condition is not met, and return *unit* otherwise.

$$\texttt{assert}\ P = \texttt{if}\ P\ \texttt{then}\ \texttt{return}\ ()\ \texttt{else}\ \texttt{fail}$$

We now add resource consumption to our specifications. A Hoare-triple for program m, with precondition P, postcondition Q and resource usage t is written as a refinement condition:

$$m \leq \texttt{do}\ \{\ \texttt{assert}\ P;\ \texttt{spec}\ Q\ (\lambda\_.\ t)\ \}$$

Example 8.2.2. Specifications follow that pattern. The comparison of two list elements at a cost of t can be specified by:

$$\begin{aligned} &idxs\_cmp_{spec}\ t\ xs\ i\ j = \texttt{do}\ \{ \\ &\quad \texttt{assert}\ (i < |xs| \wedge j < |xs|); \\ &\quad \texttt{spec}\ (xs!i < xs!j)\ (t\ (xs,\ i,\ j)) \\ &\}\end{aligned}$$

where $xs\ !\ i$ is the ith element of list xs. Instead of fixing the cost for specifications, we pass them as parameter t. This allows us to refine different instances of abstract data types (here lists) by different concrete data structures with different costs.

Example 8.2.3. The specification for adding up the elements of a set S can now be expressed in NREST:

$$sum\_set_{spec}\ T\ S = \texttt{do}\ \{\ \texttt{assert}\ (finite\ S);\ \texttt{spec}\ (\textstyle\sum_{s \in S} s)\ (T\ S)\ \}$$

Note that the resource bound $T :: \beta \to \alpha \to \gamma$ in general may depend on the parameters (β) and the result (α) of the specified computation.

We use Isabelle's *if-then-else* to model branching and define a recursion combinator $\texttt{rec} :: ((\alpha \to (\alpha,\ \gamma)\ NREST) \to \alpha \to (\alpha,\ \gamma)\ NREST) \to \alpha \to (\alpha,\ \gamma)\ NREST$ via a fixed-point construction [74], to get a complete set of basic combinators. The `while` combinator can be derived as for *inres* but has a different type.

$$\begin{aligned}&\texttt{while} :: (\alpha \to bool) \to (\alpha \to (\alpha, \gamma)\ NREST) \to \alpha \to (\alpha, \gamma)\ nres\\ &\texttt{while}\ b\ c\ s = \texttt{rec}\ (\lambda R\ s.\ \texttt{if}\ b\ s\ \texttt{then do}\ \{\ \texttt{assert}\ (b\ s);\ s \leftarrow (c\ s);\ R\ s\ \}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\texttt{else return}\ s)\ s\end{aligned}$$

Example 8.2.4. Let us revisit the *sum_set* example from before and see how we can integrate resource consumption in it. In Figure 8.2 you can see the programs adapted to NREST. The overall structure stays the same, we can use do-notation to conveniently write down programs, and the combinators `assert` and `while` are used as before.

However, every specification (line 1 and 6) is parameterized with a resource cost, e. g. T_c and T_a. For programs that combine subprograms we can then express their resource consumption dependent on those. For a non-empty set S one could prove

$$add\_one\ T_c\ T_a\ (a,\ S) \leq \texttt{spec}\ (\lambda\_.\ True)\ (T_c\ S + T_a)$$

In the naive approach, those cost functions need to be threaded through all programs that use the specifications. Because this makes writing programs quite verbose, we explored two approaches to simplify this.

In the first iteration (Section 8.3), we observed that a program like *sum_set* always lives in an *environment* where certain basic operations are "assumed". That is, those operations are only specified but not yet provided with an implementation. In the example those basic operations are $choose_{spec}$ and add_{spec}. They come each with a resource cost. Instead of writing them out, we assume them as free variables in a logical context and compound resource bounds will depend on those variables. This context then represents the "environment". To later use the environment one has to instantiate the free variables with concrete bounds and one obtains bounds on compound programs by inserting them.

One could argue that given those environments one does not need the refinement calculus on the NREST monad at all. Instead, one can do ad hoc refinement with the context mechanism. Programs that are not yet implemented are just assumed as free variables together with some properties about them. Then, compound programs use those subprograms and establish correctness with the help of those properties. In order to use the compound program in the end, one needs to provide implementations for the the assumed subprograms.

In a second iteration (Section 8.4), we use *resource currencies* to avoid explicitly using logical contexts. Instead, specifications (like add_{spec}) typically cost one coin of a certain currency (e. g. named *add*). Compound programs then have costs comprised by several coins of different currencies. When refining a specification, one typically provides a compound program that refines the specification by using a combination of several subprograms. The cost of that compound program is in terms of the coins needed for those subprograms. In that process the abstract coin needs to be exchanged into the cost of the compound program. We call that process *currency refinement*, as

```
1  choose_spec T_c S = do {
2    assert (S ≠ ∅);
3    spec (λ(s, S'). s ∉ S' ∧ S' ∪ {s} = S) (λ_. T_c S)
4  }
5
6  add_spec T_a a s = spec (a + s) (λ_. T_a)
7
8  add_one T_c T_a (a, S) = do {
9    (s, S') ← choose_spec T_c S;
10   a' ← add_spec T_a a s;
11   return (a', S')
12 }
13
14 sum_set T_c T_a S = do {
15   s ← return (0, S);
16   (a', S') ← while (λ(a, S). S ≠ ∅)
17               (λ(a, S). add_one T_c T_a (a, S))
18               s;
19   return a'
20 }
```

Figure 8.2: The *sum_set* example with resources.

typically a single coin of some program is exchanged into several coins of "smaller" subprograms. Those coins are "refined" in that sense. This way feels kind of natural, and there we also have some kind of "environment" in the form of the currency system used for each program. By using this approach it is not necessary to explicitly use those logical contexts and the full power of stepwise refinement can be unleashed.

Example 8.2.5. Let us have a look at another example. Say we have specifications for lookup and compare and want to implement the specification $idxs\_cmp_{spec}$.

$lookup_{spec}\ t\ xs\ i = \texttt{do}\ \{\ \texttt{assert}\ (i < |xs|);\ \texttt{spec}\ (xs\ !\ i)\ (\lambda\_.\ t)\ \}$
$cmp_{spec}\ t\ a\ b = \texttt{spec}\ (a < b)\ (\lambda\_.\ t)$

Here, the term $xs\ !\ i$ denotes the ith element of list xs. We can now come up with the following refinement for $idxs\_cmp_{spec}$ from Example 8.2.2:

```
idxs_cmp xs i j = do {
    xsi ← lookup_spec () xs i ;
    xsj ← lookup_spec () xs j ;
    cmp_spec () xsi xsj
  }
```

In the *nres* monad, we would now prove $idxs\_cmp\ xs\ i\ j \leq idxs\_cmp_{spec}\ (\lambda\_.\ ())\ xs\ i\ j$ by applying the verification condition generator *refine_vcg* and solving the trivial ver-

ification conditions automatically. In the next section I will describe how *refine_vcg* can be adapted to work for NREST.

The main components of *NREST* are the same as for *nres*, just with a different type. I will briefly show how they can be adapted in the general case, and will provide the adapted reasoning infrastructure in the next two sections.

Data Refinement Because NREST is defined on partial functions to the resource type, instead of sets, we have to adapt the definition of data refinement.

$$\Downarrow_D R\ (\texttt{res}\ M) = \texttt{res}\ (\lambda c.\ Sup\ \{M\ a \mid a.\ (c,\ a) \in R\}) \qquad \Downarrow_D R\ \texttt{fail} = \texttt{fail}$$

We can also use the same notation for data refinements:

$$(m,\ m') \in [pre]\ R \to S$$

expresses that if precondition *pre* on the abstract parameters holds, program m refines m' w. r. t. relation R for the arguments and S for the result.

Specification Refinement We will comment on the automation for specification refinement in the next section. We will have to generalize the weakest precondition calculus (Section 8.3.4), but once that is established the automation works analogously.

Lockstep Refinement Refinement of structurally equal programs works analogously for NREST as for *nres*. We have to prove monotonicity lemmas for all the combinators. For example we have the following monotonicity rule for `elapse`:

$$m \le \Downarrow_D R\ m' \implies t \le t' \implies \texttt{elapse}\ m\ t \le \Downarrow_D R\ (\texttt{elapse}\ m'\ t')$$

Lock step refinement with *refine_rcg* works exactly the same as for *nres*. Only when introducing resource currencies and currency refinement for NREST-ecost, we will have to adapt the mechanism. This will be explained in Section 8.4.3.

Pointwise Reasoning For NREST we can define *nofail* and *inres* in a similar way as for *nres*.

$$nofail\ m = (m \neq \texttt{fail})$$

$$\begin{aligned} &inres :: (\alpha,\ \gamma)\ NREST \to \alpha \to bool \\ &inres\ \texttt{fail}\ x = True \\ &inres\ (\texttt{res}\ M)\ x = (M\ x \neq None) \end{aligned}$$

The predicate *inres* for *NREST* has the same property as for *nres*:

$$inres\ m\ x \longleftrightarrow \texttt{return}\ x \le m$$

As refinement on NREST does not only mean refinement of results but also resources, we will have to extend the pointwise reasoning approach in order to prove arbitrary NREST refinements (cf. Section 8.3.2).

We have already mentioned that NREST collapses into *nres* if set γ=*unit*. Next, I will introduce instances of NREST that actually reason about resources. First, the resource type of extended natural numbers measures resource consumption with one single number (Section 8.3). We do not only want to model computation but also prove refinement and correctness lemmas. I will present how the reasoning infrastructure of *nres* can be generalized to *NREST*. Finally, we use resource functions whose domain is a set of *currencies* to measure resource consumption in a fine-grained manner (Section 8.4). Further ideas for resource types will be discussed in Section 8.5.

8.3 Measuring Resources with a Number

Now let us add non-trivial resources to the refinement. In the first iteration of this work [45] concerning refinement with resources, the resource type was fixed to extended natural numbers ($enat = \mathbb{N} \cup \{\infty\}$), measuring the resource consumption with a single number. We will refer to this instance by *NREST-enat*.

We will first give an example to illustrate the expressivity of the formalism, and then delve into how to extend the techniques from the previous section.

Example 8.3.1. Let us equip the programs from Example 8.2.5 with resource costs and fix cost parameters t_{lookup} and t_{cmp}:

$$
\begin{array}{l}
idxs\_cmp\ xs\ i\ j = \texttt{do}\ \{ \\
\quad xs_i \leftarrow lookup_{spec}\ t_{lookup}\ xs\ i; \\
\quad xs_j \leftarrow lookup_{spec}\ t_{lookup}\ xs\ j; \\
\quad cmp_{spec}\ t_{cmp}\ xs_i\ xs_j \\
\}
\end{array}
$$

The cost for *idxs_cmp* can be deduced as a function in the costs of the compound computation. Remember that in NREST-enat we use a logical context that fixes cost parameters. In that context, we define operations like *idxs_cmp* and we drop those implicit parameters for presentation purposes.

Example 8.3.2. For another example we specify the operation of appending an element to a list:

$$push\_list_{spec}\ T\ x\ xs = \texttt{res}\ [xs \cdot [x] \mapsto T\ xs]$$

Here, the term $xs \cdot ys$ denotes appending of two lists. The operation *push_list* is specified with a parameter T that represents the running time of the operation, here parameterized with the input list *xs*. For an implementor, this leaves open the possibility to provide an implementation whose time consumption depends on *xs*, e. g. on its length. We see later in Example 9.1.1 how we can refine this abstract specification with a concrete implementation.

We can refine the data in that example using dynamic lists instead of lists like in Example 5.5.1. We build the following refinement relation and can show a correctness lemma for $push\_array_{fun}$:

```
1  kruskal = do {
2    l ← obtain_sorted_edge_list;
3
4
5    (djs₀, fl₀) ← initState;
6    (djs', fl') ← nfold l (λ(a, w, b) (djs, fl). do {
7      assert (a ∈ Dom djs
8              ∧ b ∈ Dom djs);
9      b ← res [¬djs_cmp djs a b ↦ t_it];
10     if b then do {
11       assert ((a, w, b) ∉ set fl);
12       add_edge djs a b fl
13     } else
14       return (djs, fl)
15     }) (djs₀, fl₀);
16     return fl'
17 }
```

(a) A further refinement for Kruskal's algorithm, where an additional disjoint sets data structure is passed around.

```
1  mwb_greedy = do {
2    l ← spec (λL. sorted_wrt w L
3       ∧ distinct L ∧ set L = E)
4           (λ_. t_sc);
5    B₀ ← res [∅ ↦ t_eb];
6    B' ← nfold l (λe B. do {
7      assert (e ∉ B ∧ indep B
8           ∧ e ∈ c ∧ B ⊆ E);
9      b ← res [indep (B ∪ {e}) ↦ t_it];
10     if b then do {
11
12       res [B ∪ {e} ↦ t_i]
13     } else
14       return B
15   }) B₀;
16   return B'
17 }
```

(b) The greedy algorithm to construct a minimum weight basis of a matroid in the NREST monad.

Figure 8.3

$$R^{list}_{dynlist} = br\ (\lambda(bs,\ n).\ take\ n\ bs)\ (\lambda(bs,\ n).\ n < |bs|)$$
$$(\lambda(xs,\ n)\ x.\ \texttt{res}\ [push\_array_{fun}\ (xs,\ n)\ x \mapsto T\ xs],\ push\_list_{spec}\ T)$$
$$\in Id \to R^{list}_{dynlist} \to R^{list}_{dynlist}$$

Example 8.3.3. As a running example for this section, we consider the formalization of Kruskal's algorithm. To illustrate the expressive power of NREST we present the abstract algorithm in Figure 8.3b: the greedy algorithm to construct a minimum weight basis for a matroid. Note that, details of matroids will be introduced later (Section 10.1) and are irrelevant here. It suffices to think of a minimum weight basis being a generalization of minimum spanning trees. This abstract algorithm will later be instantiated for the cycle matroid, which yields the skeleton of Kruskal's algorithm. Already on this abstract level we can structure the algorithm and prove the functional correctness of the algorithmic idea, as well as its running time parameterized over the running times of the abstract operations it performs.

In line two the algorithm obtains a list of the elements of the carrier set E (later this will be the set of edges of an undirected graph) sorted w. r. t. some weight function w. Starting from an empty independent set, we iteratively add elements if they leave the set B independent (i. e. create no cycle in the graph case). Here, we use the *nfold* combinator to iterate over the list l. It has the following type:

$\beta\ list \to (\beta \to \alpha \to (\alpha,\ enat)\ NREST) \to \alpha \to (\alpha,\ enat)\ NREST$

For all operations that may cost time, we reserve some time parameter of type *nat*: here t_{sc}, t_{eb}, t_{it} and t_i stand for sorted carrier list time, empty basis time, independence test time and insertion time.

Besides the time parameters, the algorithm has further implicit arguments: The carrier set c describes all possible elements of the matroid, the weight function w assigns each of those elements a weight, and the set E is a subset of carrier set c that describes the possible elements in the problem instance.

We can give the specification for this algorithm, and state the refinement theorem:

$mwb\_greedy \le \texttt{spec}\ min\_weight\_basis\ (\lambda\_.\ t_{sc} + t_{eb} + |E| * (t_{it} + t_i))$

where the predicate *min_weight_basis* characterizes subsets of E that form a minimum weight basis of the current problem instance. After building up theory for NREST with resources, we will see how to prove such a refinement in a mechanized way. That is the subject of the next section.

Example 8.3.4. I now illustrate an effect that stems from our decision to aggregate the resource usage of different computation paths that lead to the same result. Consider the admittedly artificial program

$\texttt{res}\ (\lambda n{::}nat.\ Some\ n);\ \texttt{return}\ 0$

It first chooses an arbitrary natural number n with cost n and then returns the result *0*. That is, there are arbitrarily many paths that lead to the result 0, consuming arbitrarily many costs. The supremum of this is ∞, such that the above program is equal to $\texttt{elapse}\ (\texttt{return}\ 0)\ \infty$. Note that none of the computation paths actually attains the aggregated resource usage. We will come back to this effect in the next section and later in Section 9.2.4.

We now want to establish the monad laws and monotonicity lemmas similarly to *nres*.

For building convenient infrastructure to reason about basic equalities and refinements we now extend the pointwise reasoning to also cope with resources. For checking the refinement $m \le m'$ it does not suffice to check that any result in m is also a result of m', we also need to check that m uses at most as much resources as m'. In the following we will show that a straightforward extension of the pointwise reasoning setup does not work in general. Nevertheless, we can remedy this with a light modification for the special case *enat*.

8.3.1 Pointwise Reasoning - Naive Approach

Say we define a new predicate that expresses that a computation m can compute a result x with resource usage $t :: enat$:

$inres'_\tau :: (\alpha,\ enat)\ NREST \to \alpha \to enat$
$inres'_\tau\ \texttt{fail}\ x\ t = False$
$inres'_\tau\ (\texttt{res}\ M)\ x\ t = (\texttt{case}\ M\ x\ \texttt{of}\ None \Rightarrow False \mid Some\ r \Rightarrow t \le r)$

The predicate is *False* if the computation fails, the computation does not compute the result x or it reserves less costs then t.

For the usual combinators we can prove similar rules for $inres'_\tau$ to the ones for *inres*. Only for the supremum we run into a problem for the first time. We would like to establish

$$inres'_\tau\ (Sup\ S)\ x\ t = (\exists m{\in}S.\ nofail\ m \wedge inres'_\tau\ m\ x\ t).$$

But there is one case when the equality does not hold: if an infinite cost $t = \infty$ is attained by the result x for program *Sup S* but all computations in S have finite resource consumption for x. Then, there does not exist any m in S that can compute x with consumption $t = \infty$. This case can occur if S is an infinite set, as we have just seen in Example 8.3.4.

8.3.2 Pointwise Reasoning - Special Case

Alternatively, we rule out the infinite case by defining $inres_\tau$ such that the resource consumption must be finite, i. e. x is a natural number:

$$inres_\tau :: (\alpha,\ enat)\ NREST \to \alpha \to nat$$
$$inres_\tau\ \texttt{fail}\ x\ t = False$$
$$inres_\tau\ (\texttt{res}\ M)\ x\ t = (\texttt{case}\ M\ x\ \texttt{of}\ None \Rightarrow False \mid Some\ r \Rightarrow enat\ t \leq r)$$

Note that we need the coercion in $enat\ t \leq r$, as r is possibly infinite and t is a finite number. Observe, that this refines *inres*: we can obtain it via the following equality.

$$inres\ m\ x = \exists t{::}nat.\ inres_\tau\ m\ x\ t$$

Now we actually can prove $inres_\tau\ (Sup\ S)\ x\ t = (\exists m{\in}S.\ nofail\ m \wedge inres_\tau\ m\ x\ t)$ because we can follow the argument from above. Only, the failing case $t{=}\infty$ is ruled out. At the same time we can still reduce refinement lemmas to pointwise form:

$$(nofail\ m \implies nofail\ m') \wedge (\forall x\ t.\ inres_\tau\ m\ x\ t \implies inres_\tau\ m'\ x\ t) \implies m \leq m'$$

A program m refines a program m' when for any result x if m can compute the result with t resources, then also m' can. The extraordinary fact is that it suffices to know that implication for finite t, which makes it enough to define $inres_\tau$ on natural numbers.

Proof. To prove the lemma we can work off the easy cases and need to show for any result x of m, that program m' consumes at least as much resources. Let x be such a result and t_x and ${t_x}'$ be the resource costs of m and m' for it. We need to show $t_x \leq {t_x}'$. If m consumes a finite amount ($t_x < \infty$) we can use the second premise and are done. If m consumes $t_x = \infty$ many resources for result x, we need to prove that also m' does, i. e. ${t_x}' = \infty$. We can do this by showing that ${t_x}'$ is at least as large as any arbitrarily chosen natural number τ. Let us fix that τ. Then, we can use the second premise, as we know that the antecedent is true for any finite t. Unfolding the definition of $inres_\tau$ shows that ${t_x}'$ needs to be at least τ and ultimately needs to be ∞. □

I think it is remarkable that this technique works. We need to keep in mind that this construction only works for an infinite domain without a top element augmented with a limit top element. Unfortunately, we cannot expect to use that construction for arbitrary resource types γ, but it works for our purposes.

8.3.3 Monad Laws, Data Refinement, and Lockstep Refinement

With the adjusted pointwise reasoning infrastructure in place, the monad laws of NREST-enat are proven automatically.

Data refinement is orthogonal to introducing the counting of resource consumption, as it only acts on the domain of the maps, not on their values. The monotonicity lemmas for all combinators can be lifted, augmented with data refinement and proven correct with the pointwise reasoning setup. I repeat the rule for **bind** again here for programs m and m' of type (_, *enat*) *NREST*.

$$\begin{array}{l}[\![\ m \le \Downarrow_D R'\ m';\\ \quad (\forall x\ x'.\ (x,\ x') \in R' \wedge \mathit{inres}\ m\ x \wedge \mathit{inres}\ m'\ x' \implies f\ x \le \Downarrow_D R\ (f'\ x'))\ ]\!]\\ \quad \implies \mathbf{bind}\ m\ f \le \Downarrow_D R\ (\mathbf{bind}\ m'\ f')\end{array}$$

Note that for the rule we use *inres* in the precondition of the second premise. It is useful and often necessary to recover facts about the intermediate results x and x'. For example, if $m = \mathbf{return}\ b$ is the guard of a branching it can be recovered with the rule $\mathit{inres}\ (\mathbf{return}\ b)\ x \implies x = b$. Recovering facts about the costs of m for x has not been necessary in our use cases. That is why we use the more abstract *inres* instead of inres_τ for the rule. The rule for **bind** looks exactly the same as for *nres*. It only has a different type.

In consequence, also lockstep refinement works unaltered: the refinement condition generator traverses two structurally equal programs, equipped with refinements of the basic operations and combines them with the monotonicity lemmas just mentioned.

Example 8.3.5. Consider the two programs in Figure 8.3. The concrete program *kruskal* is a specialized minimum weight basis algorithm for the cycle matroid, where the elements of the matroid are edges in an undirected graph, represented by a tuple (a, w, b) of its end nodes a and b and weight w. Programs *obtain_sorted_edge_list* and *add_edge* are compound programs. We want to show the following refinement relation:

$$\mathit{kruskal} \le \Downarrow_D\ R^{graph}_{list}\ \mathit{mwb\_greedy}$$

where R^{graph}_{list} relates a set of abstract edges in the graph with a list of edge tuples representing them. The relation R^{edge}_{tuple} relates abstract edges with edge tuples. The above lemma can be proved by lockstep refinement using the *refine_rcg* component. In that process several intermediate refinement relations are used, e. g. $((\mathit{djs}, \mathit{fl}), B) \in R^{edgeset}_{djs}$ which relates the abstract edge set B to the list of edges *fl* and its corresponding disjoint-sets forest *djs*. The main part of this refinement proof is to show that testing independence when we add an edge (a, w, b) (i. e. checking cycle-freedom) can be implemented by comparing the equivalence classes of a and b.

Note that *add_edge* has to do two things: update the disjoint-sets forest and add the edge tuple to the list. We specify this program abstractly, and reserve time t_{iu} and t_{il} for the two actions. In the refinement proof we need to prove that $t_{iu} + t_{il} \leq t_i$. Similarly, the sum of the costs in *obtain_sorted_edge_list* must be smaller than t_{sc}.

The VCG for refinement *refine_vcg* simulates the two programs side by side, using the monotonicity lemmas to split the problem into smaller parts, and showing the refinements of those smaller parts. I omit the formal statement of the monotonicity rule for *nfold*. Similarly to the rule for `while`, the results of two *nfold* programs are in a refinement relation if the start state is in a refinement relation and the loop body preserve that.

One verification condition that is emitted by *refine_rcg* is the following.

$$[\![\ ((djs,\ fl),\ B) \in R^{edgeset}_{djs};\ ((a,\ w,\ b),\ e) \in R^{edge}_{tuple}\ ]\!]$$
$$\Longrightarrow\ add\_edge\ djs\ a\ b\ fl \leq \Downarrow_C R^{graph}_{list}\ (\texttt{res}\ [B \cup \{e\} \mapsto t_i])\}$$

where djs_graph_{rel} is motivated as above. It involves proving that *add_edge* refines the insertion of the edge e into B, while maintaining the disjoint-sets data structure *djs* and incurring no more than t_i cost.

In order to introduce a more succinct notation for such refinement lemmas, we have set up similar notation for NREST as we have seen for *nres* at the end of Example 8.1.12. The above refinement can be written as:

$$(\lambda((djs,\ fl),\ (a,w,b)).\ add\_edge\ djs\ a\ b\ fl,\ \lambda(B,\ e).\ \texttt{res}\ [B \cup \{e\} \mapsto t_i])$$
$$\in R^{edgeset}_{djs} \to R^{edge}_{tuple} \to R^{graph}_{list}$$

It expresses, that if the parameters are in a refinement relation, the results of *add_edge* and the specification are in a refinement relation. Keep in mind, that this lemma lives in the context where the time parameters have been fixed.

8.3.4 Specification Refinement in NREST-enat

For proving refinement lemmas of the form $m \leq \texttt{res}\ \Phi$ in NREST-enat we again need to find rules for all the combinators. Here, the type of Φ is $\alpha \to enat\ option$. In order to come up with meaningful rules for these combinators we first need to generalize the goal.

Instead of asking only *whether* a program satisfies the specification, we also ask "*how much*" it satisfies the specification. That is, we ask for the difference of the resources specified and actually used, denoted by *gwp m* Φ. The *generalized weakest precondition gwp* has type $(\alpha,\ enat)\ NREST \to (\alpha \to enat\ option) \to \alpha \to enat\ option$. It brings the weakest prepotential (Section 2.3) and the weakest preexpectations (Section 3) to mind.

For *gwp* we have the following equality:

$$m \leq \texttt{res}\ \Phi \Leftrightarrow Some\ 0 \leq gwp\ m\ \Phi$$

To get some intuition let us fix the resource to be time. Then, *gwp m* Φ is the *latest* feasible time at which we can start m to still match the deadline Φ. If there is no

feasible starting time ($gwp\ m\ \Phi = None$), m does not fulfill the specification Φ. If it has some value t, this is the latest feasible starting time of all computation paths in m to reach the deadline Φ.

Before we give the definition of *gwp*, let us explore what we can do with it. We can prove the following equality for the **bind** operator:

$$gwp\ (\textbf{bind}\ m\ f)\ \Phi = gwp\ m\ (\lambda y.\ gwp\ (f\ y)\ \Phi)$$

Intuitively it says: The latest starting time for the compound computation **bind** m f to satisfy Φ is the latest starting time for m in order to meet the latest starting time such that $f\ y$ meets the specification Φ.

To determine $gwp\ m\ \Phi$, we need to consider the differences between the specified and the actual resource consumption time for every result of m and take the most conservative one:

$$gwp\ m\ \Phi = Inf_r\ minus\ \Phi\ m\ r$$

Operation $minus :: (\alpha \to enat\ option) \to (\alpha, enat)\ NREST \to \alpha \to enat\ option$ formalizes "taking the difference". We have the following cases:

- m fails: then m may never be executed and thus there is no valid latest starting time, i. e. $minus\ \Phi\ m\ r = None$.
- $m = \textbf{res}\ M$ and $M\ r = None$: as M will never produce the result r it can be ignored, i. e. the result is the top element: $Some\ \infty$.
- $m = \textbf{res}\ M$ and $M\ r = Some\ t$ and $\Phi\ r = None$: r is specified to not be obtained, but when starting m we obtain r. Thus, there is no valid starting time for m: $minus\ \Phi\ m\ r = None$.
- $m = \textbf{res}\ M$ and $M\ r = Some\ t$ and $\Phi\ r = Some\ t'$: if more time is needed than specified ($t > t'$), there is no valid latest starting time and we return $None$, otherwise the difference is returned ($Some\ (t' - t)$).

We can get some more intuition when unfolding *gwp* in the above equality:

$$\begin{aligned} & m \le \textbf{res}\ \Phi \\ \iff & Some\ 0 \le gwp\ m\ \Phi = Inf_r\ minus\ \Phi\ m\ r \\ \iff & \forall r.\ Some\ 0 \le minus\ \Phi\ m\ r \end{aligned}$$

The infimum is just a compact version of saying that the difference of Q and m on *any* result r is non-negative.[4] By abusing notation and following the intuition of minus one can restate the last line as "$\forall r.\ m\ r \le Q\ r$". In essence it says, that c meets specification Q if and only if for any r the time that it takes to calculate r for m is at most the time that Q reserved for that result.

Instead of solving problems of the form $m \le \textbf{res}\ Q$, we solve problems of the more general form $Some\ t \le gwp\ c\ Q$. This general form allows us to state syntax-directed

[4] As we have seen in Chapter 4, infimum on Booleans is the forall quantifier.

rules in a uniform way, which would not be possible without the definition of *gwp*. Compare the **bind** rule in the last section.

From the equality for *gwp* on **bind** we can derive an introduction rule for **bind**:

$$\mathit{Some}\ t \le \mathit{gwp}\ m\ (\lambda y.\ \mathit{gwp}\ (f\ y)\ Q) \implies \mathit{Some}\ t \le \mathit{gwp}\ (\texttt{bind}\ m\ f)\ Q$$

For the other combinators we have:

$$(\forall r \in M.\ \mathit{Some}\ (t + M\ r) \le Q\ r) \implies \mathit{Some}\ t \le \mathit{gwp}\ (\texttt{res}\ M)\ Q$$
$$\mathit{Some}\ t \le Q\ x \implies \mathit{Some}\ t \le \mathit{gwp}\ (\texttt{return}\ x)\ Q$$
$$(\forall x.\ P\ x \implies \mathit{Some}\ (t + t'\ x) \le Q\ x) \implies \mathit{Some}\ t \le \mathit{gwp}\ (\texttt{spec}\ P\ t')\ Q$$
$$\mathit{Some}\ (t + t') \le \mathit{gwp}\ M\ Q \implies \mathit{Some}\ t \le \mathit{gwp}\ (\texttt{elapse}\ M\ t')\ Q$$

Example 8.3.6. Let us prove the refinement for *idxs_cmp* from Example 8.3.1:

We now can prove the following refinement lemma using the verification condition generator.

$$2 * t_{lookup} + t_{cmp} \le T \implies \mathit{idxs\_cmp}\ xs\ i\ j \le \mathit{idxs\_cmp}_{spec}\ T\ xs\ i\ j$$

One verification condition will check whether the specification allots enough resources for the algorithm. It can be discharged by the premise.

Operation *nfold* has the following type and the following *gwp* rule:

$$\mathit{nfold} :: \beta\ \mathit{list} \to (\beta \to \alpha \to (\alpha,\ \mathit{enat})\ \mathit{NREST}) \to \alpha \to (\alpha,\ \mathit{enat})\ \mathit{NREST}$$

$$\begin{array}{ll}
1 & [\![\ I\ []\ l_0\ s_0; \\
2 & \quad (\forall x\ l_1\ l_2\ s.\ l_0 = l_1 \cdot [x] \cdot l_2 \wedge I\ l_1\ ([x] \cdot l_2)\ s \\
3 & \qquad \implies \mathit{Some}\ 0 \le \mathit{gwp}\ (f\ x\ s)\ (\mathit{emb}\ (I\ (l_1 \cdot [x])\ l_2)\ t_{body})); \\
4 & \quad (\forall s.\ I\ l_0\ []\ s \implies \mathit{Some}\ (t + t_{body} * |l_0|) \le Q\ s)\ ]\!] \\
5 & \quad \implies \mathit{Some}\ t \le \mathit{gwp}\ (\mathit{nfold}\ l_0\ f\ s_0)\ Q
\end{array}$$

Here, $\mathit{emb}\ P\ t = (\lambda x.\ \texttt{if}\ P\ x\ \texttt{then}\ \mathit{Some}\ t\ \texttt{else}\ \mathit{None})$, *nfold* is defined in a straightforward manner and the invariant I is a predicate that takes as its first argument the list of already processed elements, then the list of elements still to be processed and finally a state s. Here the amount t_{body} is a constant resource cost for any call of the loop body. For showing that $\mathit{nfold}\ l_0\ f\ s_0$ meets its specification Q with slack resource t, one has to show the three premises. First, the invariant I has to hold initially (line 1). Then, the body preserves the invariant and takes at most t_{body} time steps (line 2-3). Finally, the invariant in the end implies the desired specification (line 4). As we fold over a finite list, a termination argument is not required.

With the above rules and analogous rules for **assert** and the combinators **if** and **case**, we construct a syntax-directed verification condition generator that exhaustively applies those rules.

Example 8.3.7. After annotating the loop in the program *mwb_greedy* (Figure 8.3b) with $\mathit{body}_{time} = t_{it} + t_i$ and a suitable invariant $I = \lambda l_1\ l_2\ T.\ I_{mwb}\ (T,\ \mathit{set}\ l_2)$ (where $I_{mwb}(T, E)$ implies $\mathit{min\_weight\_basis}\ T$ for the whole carrier set E), we run the VCG on the refinement theorem of Example 8.3.3 and obtain eleven verification conditions.

One of those is the invariant preservation of the first branch of the if-expression, i. e. when adding an element e:

$$\textit{sorted\_wrt}\ w\ l \wedge \textit{distinct}\ l \wedge \textit{set}\ l = E \wedge l = l_1 \cdot [e] \cdot l_2 \wedge \textit{indep}\ (T \cup \{e\}) \\ \wedge\ I_{mwb}(T,\ \textit{set}\ ([e] \cdot l_2)) \implies I_{mwb}(T \cup \{e\},\ \textit{set}\ l_2)$$

This verification condition is one of the central ones in the correctness proof and can be discharged with an interactive proof.

8.3.5 Splitting Resources from Functional Correctness

We can disregard resource usage and only focus on refinement of functional correctness, and then add resource usage analysis later. This is useful to separate the concerns of functional correctness and resource usage proof. I will describe a practical example in a case study later (Section 10.3.5). I only present an alternative way to prove the refinement between *idxs_cmp* and $\textit{idxs\_cmp}_{spec}$ in Example 8.3.6 here:

Example 8.3.8. For functional correctness, we use the specification $\textit{idxs\_cmp}_{spec}\ (\infty)$ and a program $\textit{idxs\_cmp}_\infty$ similar to *idxs_cmp* but with all the costs replaced by ∞. Proving the refinement $\textit{idxs\_cmp}_\infty\ xs\ i\ j \le \textit{idxs\_cmp}_{spec}\ xs\ i\ j\ (\infty)$ only requires showing verification conditions that correspond to functional properties and termination.[5] In particular, assertions and annotated invariants in the concrete program have to be proved. Proof obligations on resource usage, however, collapse into the trivial $t \le \infty$. For the same reason, we get $\textit{idxs\_cmp}\ xs\ i\ j \le \textit{idxs\_cmp}_\infty\ xs\ i\ j$ from a lockstep refinement, and by transitivity obtain

$$\textit{idxs\_cmp}\ xs\ i\ j \le \textit{idxs\_cmp}_{spec}\ \infty\ xs\ i\ j$$

Next, we prove $\textit{idxs\_cmp}\ xs\ i\ j \le_n \texttt{spec}\ (\lambda\_.\ \textit{True})\ (2 * t_{lookup} + t_{cmp})$. Here, the refinement relation $m \le_n m' \equiv \textit{nofail}\ m \implies m \le m'$ assumes that the concrete program does *not* fail. This has the effect that, during the refinement proof, assertions and annotated invariants in the concrete program can be assumed to hold rather than need to be proven. As a result, we can focus on the resource usage proof.

Finally, the two refinements can be combined to obtain

$$\textit{idxs\_cmp}\ xs\ i\ j \le \textit{idxs\_cmp}_{spec}\ xs\ i\ j\ (2 * t_{lookup} + t_{cmp})$$

To prove refinements $m \le_n \texttt{res}\ \Phi$, we prove $m \le \texttt{res}\ \Phi$ with the additional premise *nofail m*. Alternatively, we can reduce the goal to show $\textit{Some}\ 0 \le \textit{gwp}_n\ m\ \Phi$, where $\textit{gwp}_n\ m\ \Phi = (\texttt{if}\ \textit{nofail}\ m\ \texttt{then}\ \textit{gwp}\ m\ \Phi\ \texttt{else}\ \textit{Some}\ \top)$. Similarly to the rules for *gwp*, we can state the syntax-directed rules for all the combinators also for $\textit{gwp}_n$ and prove them using the respective rules for *gwp*. Feeding the verification condition generator with those rules and the introduction rule $\textit{Some}\ 0 \le \textit{gwp}_n\ m\ \Phi \implies m \le_n \texttt{res}\ \Phi$ obtains an automatic proof procedure for those goals.

[5] In essence, the same verification conditions as in the analogous refinement lemma in the Boolean NREST have to be proved.

8.3.6 Organize Refinements with Locales

When writing down algorithms in NREST-enat (e. g. Kruskal in Figure 8.3a), we need to specify the resource consumption of all the basic operations. We do not want to fix those costs in advance. On the one hand, because we may not yet know the exact costs or may want to change it in the future, such that a modular interface is preferable. On the other hand, we may want to reuse algorithms in different instances with different data structures which have different resource costs.

In the first iteration our solution is to keep those cost expressions as parameters. In Isabelle/HOL we realize that by forming a locale for any set of abstract operations. Then, every abstract operation X_{spec} fixes the resource usage with a parameter X_{time}. Here is an example:

Example 8.3.9. The program *idxs_cmp* from Example 8.3.6 uses two basic operations: the lookup and the the compare operation. The program *idxs_cmp* can be defined in a locale that has the two parameters t_{cmp} and t_{lookup}.

When we want to use the program *idxs_cmp* in another context that refines the used data structures, we have to instantiate those parameters. That is, we have to give values for those parameters. We may later want to implement the list by an array with cost *2* for lookup and fix the elements of the list to be natural numbers with a compare operation that incurs cost *1*. We can instantiate the locale with $t_{cmp} = 2$ and $t_{lookup} = 1$ and automatically obtain a refinement lemma:

$$idxs\_cmp\ xs\ i\ j \leq idxs\_cmp_{spec}\ (\lambda\_.\ 5)\ xs\ i\ j$$

In order to later refine *idxs_cmp* we have to refine its basic operations with implementations that at most use *2* and *1* resource respectively.

The numbers we insert here are the concrete costs that correspond to the costs in the concrete semantics of the program we want to synthesize later. If we choose too small values here, it does not render the algorithm analysis unsound, but it will make it impossible to synthesize concrete implementations from it. We will see that mechanism in Section 9.1.5 when we synthesize programs in Imperative-HOL-Time from NREST-enat programs.

Already in the *nres* monad, contexts were used to structure the refinement and make arguments implicit instead of carrying them around for each definition of an algorithm. Using locales and stepwise refinement makes the verification of larger algorithm developments involving resource analysis feasible. I will mention more details when I present the complete verification of Kruskal's algorithm in Section 10.1.

While this approach works efficiently, it is not satisfying in the sense that it does not feel natural. In Section 7.1, we argued that we naturally would say things like "this algorithm uses a UNION subroutine at this point" and would carry on analyzing how often the subroutine will be called in that algorithm. One would even analyze the asymptotic complexity of how often the subroutine is called dependent on the size of the input. Only later, one would refine how the subroutine UNION was implemented and how much it costs.

In the next section I will introduce *resource currencies* that allow to measure the resource usage of a program not only by one number, but by the amount of several currencies. One currency could be a "UNION coin" $\$_{union}$. By allowing to refine coins with *currency refinement*, details on what a specific coin really costs can be added later. Those coins are like money, in that they only represent a means of exchange. The final value is the costs in the concrete semantics it is ultimately exchanged for. The next section explains it.

8.4 Fine-Grained Resources

We have now seen NREST with resource type being *unit* and *enat*. In this section I will introduce *resource currencies* that allow to measure resource usage in distinct coins. In particular, the ability to exchange coins into coins of different currencies allows to refine the resource usage stepwise without the tedious usage of locales and placeholder variables.

Recall that we call γ a resource type if it has a complete lattice and a monoid structure. If γ is a resource type, so is $\eta \to \gamma$, as those structures can be lifted pointwise. Intuitively, such resources consist of *coins* of different *resource currencies* η, the *amount* of coins being measured by γ.

Example 8.4.1. In the following we use the resource type $ecost = string \to enat$. The string describes the name of a currency, whose amount is measured by an extended natural number. Arbitrary resource usage is modeled by the value ∞. I will refer to this instance as *NREST-ecost*.

Note that, while the resource type *string→enat* guides intuition, most of the theory works for general resource types of the form $\eta \to \gamma$ or even just γ.

Recall the function $\$_s\ n$ to be the resource function that uses $n :: enat$ coins of the currency $s :: string$, and write $\$_s$ as shortcut for $\$_s\ 1$.

A program that sorts a list in $O(n^2)$ can be specified by:

$$sort_{spec}\ xs = \texttt{spec}\ (\lambda xs'.\ sorted\ xs' \wedge mset\ xs' = mset\ xs)\ (\$_q\ |xs|^2\ +\ \$_c)$$

That is, a list xs can result in any sorted list xs' with the same elements, and the computation takes (at most) quadratically many q coins in the list length, and one c coin, independently of the list length. Intuitively, the q and c coins represent the constant factors of an algorithm that implements that specification and are later elaborated by exchanging them into several coins of more fine-grained currencies, corresponding to the concrete operations in the algorithm, e. g., comparisons and memory accesses. Abstract currencies like q and c only *"have value"* if they can be exchanged to meaningful other currencies and finally pay for the resource costs of a concrete implementation.

Example 8.4.2. Let us consider another example. How would we model the running example $idxs\_cmp_{spec}$ using currencies (Example 8.2.2)? We specify the comparison of two list elements at a price of one *idxs_cmp* coin with the following term:

$$idxs\_cmp_{spec}\ (\$_{idxs\_cmp})\ xs\ i\ j$$

We do not need to form a context with a parameter for the operation cost but can use the currency refinement mechanism to later refine those costs.

For reasoning effectively about programs with resource type *ecost*, we need to build up the same infrastructure as for the simpler cases. We will present that theory in the following, introduce *currency refinement* and complicates the lockstep refinement case.

8.4.1 Pointwise Reasoning

We now want to lift the pointwise reasoning from earlier to the usage of currencies.

Intuitively, showing refinement for one currency should be as hard as showing refinement for several currencies. We just can check the refinement for all the currencies separately, i. e. perform another pointwise reasoning.

We follow that intuition and project computations to one currency. The program $\Pi\ c\ m$ has the same results as m, but only talks about the resource c.

$\Pi :: string \to (\alpha,\ ecost)\ NREST \to (\alpha,\ enat)\ NREST$
$\Pi\ c\ \mathtt{fail} = \mathtt{fail}$
$\Pi\ c\ (\mathtt{res}\ M) = \mathtt{res}\ (\lambda x.\ \mathtt{case}\ M\ x\ \mathtt{of}\ None \Rightarrow None \mid Some\ T \Rightarrow Some\ (T\ c))$

We can decide a refinement of a program in NREST-ecost by reducing it to normal resources:

$m \le m' \longleftrightarrow (\forall c.\ \Pi\ c\ m \le \Pi\ c\ m')$

To reduce basic refinement lemmas to be solvable by first-order provers, we first apply the above lemma and then the introduction rule for using pointwise reasoning. This yields a goal involving terms like $inres_\tau\ (\Pi\ c\ m)\ x\ t$ where t is of type *nat*.

In order to solve those goals, we need simplification rules, that pull the combinators in m through the projection. We prove lemmas for all the basic combinators:

$nofail\ (\Pi\ c\ m) = nofail\ m$
$\Pi\ c\ (\mathtt{return}\ x) = \mathtt{return}\ x$
$\Pi\ c\ (Sup\ A) = Sup\ \{\Pi\ c\ m \mid m \in A\}$
$\Pi\ c\ (\mathtt{elapse}\ m\ T) = \mathtt{elapse}\ (\Pi\ c\ m)\ (T\ c)$
$\Pi\ c\ (\mathtt{bind}\ m\ f) = \mathtt{bind}\ (\Pi\ c\ m)\ (\lambda x.\ \Pi\ c\ (f\ x))$

Especially the lemma for the supremum is important. For it to hold, it is necessary that projection is *continuous*, which indeed is the case.

Example 8.4.3. Let us prove the first monad law for $f :: \beta \to (\alpha,\ ecost)\ NREST$. The goal is

$\mathtt{bind}\ (\mathtt{return}\ x)\ f = f\ x$

In order to reduce the goal from NREST-ecost to NREST-enat it can be rewritten to

$\forall c.\ \Pi\ c\ (\mathtt{bind}\ (\mathtt{return}\ x)\ f) = \Pi\ c\ (f\ x)$

Then, it can be simplified to

$\forall c.\ \mathtt{bind}\ (\mathtt{return}\ x)\ (\lambda x.\ \Pi\ c\ (f\ x)) = \Pi\ c\ (f\ x)$

Finally, the goal is solved by using the first monad law for type $\beta \to (\alpha,\ enat)\ NREST$.

Monad laws, Monotonicity Lemmas and Data Refinement With the pointwise reasoning in place for *ecost*, the monad laws and monotonicity lemmas including data refinement can be proved automatically. We can use and prove the original monotonicity rule for `bind` that uses *inres* as a premise to pull information about the intermediate results x.

8.4.2 Currency Refinement

The new feature that is possible through using currencies is to be able to exchange them. Consider we want to refine Example 8.2.2 into a program that first accesses the elements and then compares them.

Example 8.4.4. We refine $idxs\_cmp_{spec}$ ($\$_{idxs\_cmp}$) from Example 8.4.2 by executing two list lookups and a compare operation. We provide a specification for the list lookup and a program *idxs_cmp*.

$$list\_get_{spec}\ T\ xs\ i = \texttt{do}\ \{\ \texttt{assert}\ (i < |xs|);\ \texttt{spec}\ (xs\ !\ i)\ T\ \}$$

$$\begin{aligned}
&idxs_cmp\ xs\ i\ j = \texttt{do}\ \{\\
&\quad \texttt{assert}\ (i < |xs| \wedge j < |xs|);\\
&\quad xsi \leftarrow list_get_{spec}\ \$_{lookup}\ xs\ i;\\
&\quad xsj \leftarrow list_get_{spec}\ \$_{lookup}\ xs\ j;\\
&\quad \texttt{return}\ (xsi <_{\$_{cmp}} xsj)\\
&\}
\end{aligned}$$

where $\texttt{return}\ (a <_T b)$ returns the result $a < b$ incurring cost T. We will use similar notation for other binary operators.

Note that *idxs_cmp* and $idxs\_cmp_{spec}$ use different, incompatible currency systems. To compare them, we need to exchange coins: one *idxs_cmp* coin will be traded for two *lookup* coins and one *less* coin.

To make that happen we introduce the *currency refinement* $\Downarrow_C E\ m$. Here, the *exchange rate* $E :: \eta_a \to \eta_c \to \gamma$ specifies for each abstract currency $c_a :: \eta_a$ how many of the coins of the concrete currency $c_c :: \eta_c$ are needed. Note that, in general, one abstract coin may be exchanged into multiple coins of different currencies. For a resource type γ that provides a multiplication operation $(*)$ we define the operator $\Downarrow_C$ with the following two rules:

$$\begin{aligned}
&\Downarrow_C E\ (\texttt{res}\ M) = \texttt{res}\ (\lambda\ r.\ \texttt{case}\ M\ r\ \texttt{of}\ None \Rightarrow None \mid\\
&\qquad\qquad Some\ t \Rightarrow Some\ (\lambda c_c.\ \textstyle\sum_{c_a}\ t\ c_a * E\ c_a\ c_c))\\
&\Downarrow_C E\ \texttt{fail} = \texttt{fail}
\end{aligned}$$

The refined computation has the same results as the original. To get the amount of a concrete coin c_c for some result r with resource function t, we sum over all abstract coins c_a, the amount of abstract coins needed in the original computation ($t\ c_a$) weighted by the exchange rate ($E\ c_a\ c_c$).

Example 8.4.5. Let $m\ x\ y = \texttt{res}\ [(x + y)\ /\ 2 \mapsto \$_{avg}]$ be a deterministic program that calculates the average of two numbers at the cost of one *avg* coin. We want to implement the operations in two steps: first taking the sum, then dividing by two. Let E be the exchange rate that spends one *add* and one *div* coin for each *avg* coin: $E = \uparrow\downarrow[avg{:=}\$_{add} + \$_{div}]$. Here, $+$ is lifted to functions in a pointwise manner and $\uparrow\downarrow[c_0{:=}t_0, \ldots, c_n{:=}t_n]$ denotes a function that maps the elements c_i to t_i and all other elements to *0*.

When applying the exchange rate E to m, we expect the cost for the result to be $\$_{add} + \$_{div}$. We can calculate what we intuitively expect:

$$\begin{aligned}
\Downarrow_C E\ m &= \texttt{res}\ [(x + y)\ /\ 2 \mapsto (\lambda c_c.\ \textstyle\sum_{c_a}\ (\$_{avg})\ c_a * E\ c_a\ c_c)] \\
&= \texttt{res}\ [(x + y)\ /\ 2 \mapsto (\lambda c_c.\ \textstyle\sum_{c_a}\ (\texttt{if}\ c_a = avg\ \texttt{then}\ 1\ \texttt{else}\ 0) * E\ c_a\ c_c)] \\
&= \texttt{res}\ [(x + y)\ /\ 2 \mapsto (\lambda c_c.\ \textstyle\sum_{c_a}\ (\texttt{if}\ c_a = avg\ \texttt{then}\ E\ avg\ c_c\ \texttt{else}\ 0))] \\
&= \texttt{res}\ [(x + y)\ /\ 2 \mapsto (\lambda c_c.\ E\ avg\ c_c)] \\
&= \texttt{res}\ [(x + y)\ /\ 2 \mapsto (\lambda c_c.\ (\$_{add} + \$_{div})\ c_c)] \\
&= \texttt{res}\ [(x + y)\ /\ 2 \mapsto \$_{add} + \$_{div}]
\end{aligned}$$

For the summation in the definition of currency refinement to make sense, there must be only finitely many abstract coins c_a with $t\ c_a * E\ c_a\ c_c \neq 0$. This can be ensured by restricting the resource functions t of the computation to use finitely many different coins, or by restricting the exchange rate E accordingly. The latter can be checked syntactically in practice.

Example 8.4.6. For refining the specification $idxs\_cmp_{spec}$ we can use the exchange rate $E_1 = \uparrow\downarrow[idxs\_cmp{:=}\ \$_{lookup}\ 2 + \$_{less}]$, which does the correct exchange for currency *idxs_cmp* and is zero everywhere else. We can now prove:

$$idxs\_cmp\ xs\ i\ j \leq \Downarrow_C E_1\ (idxs\_cmp_{spec}\ \$_{idxs\_cmp}\ xs\ i\ j)$$

For convenience, we introduce currency refinement into the parametric notation. The above refinement can be written as:

$$(idxs\_cmp,\ idxs\_cmp_{spec}\ \$_{idxs\_cmp}) \in \{E_1\}\ Id \to Id \to Id \to Id$$

In this particular case, one could alternatively pull the currency refinement into the specification and apply the exchange rate E_1 to the coin *idxs_cmp*.

In order to execute two exchanges E_1 and E_2 one after the other, we introduce the composition $E_1 \circ_C E_2$ on exchange rates. In that way two currency refinements can be combined into one:

$$\Downarrow_C E_2\ (\Downarrow_C E_1\ m) = \Downarrow_C (E_1 \circ_C E_2)\ m$$

8.4.3 Lockstep Refinement Revisited

During lockstep refinement we decompose the two structurally equal programs with the monotonicity rules, collect the refinement conditions, and — this is new now — have to ensure the correct exchange rate.

Note that, while the data refinements R_i can be different for each component i, the exchange rate E must be the same for all components. Currently, before a refinement step, we have to manually align the exchange rates for every component refinement lemma, which usually involves writing down and proving a specialized version of the lemma. Let us illustrate this process with an example.

Example 8.4.7. Say we want to refine *idxs_cmp* from the running example. We choose to refine the $list\_get_{spec}$ with an array lookup, and the compare operation is refined by a specific compare program, with the following two refinement lemmas:

$$list\_get_{spec}\ (\$_{ptr\_offset} + \$_{load})\ xs\ i \leq \Downarrow_C E_{alu}\ (list\_get_{spec}\ \$_{list_{get}}\ xs\ i))$$
$$\texttt{return}\ (a <_{\$_{uword\_less}} b) \leq \Downarrow_C E_{uwl}\ (\texttt{return}\ (a <_{\$_{cmp}} b))$$

Here, $E_{alu} = \uparrow\downarrow[list\_get{:=}\$_{ptr\_offset} + \$_{load}]$ and $E_{uwl} = \uparrow\downarrow[cmp{:=}\$_{uword\_less}]$. Let us denote the program with the same structure as *idxs_cmp* but using the refined subprograms by $idxs\_cmp_{arr}$.

In the naive approach that is implemented now, we need to lift the two refinements to not work with E_{alu} and E_{uwl} respectively, but with the exchange rate that executes both exchanges simultaneously:

$$E_{a\&u} = \uparrow\downarrow[list\_get{:=}\$_{ptr\_offset} + \$_{load},\ cmp{:=}\$_{uword\_less}]$$

This involves writing down the above refinement lemmas again and then proving them. Proving them is easy in the case of specifications, because often specifications only use one coin of one currency, but for more complicated programs that is already problematic. Once we found the right exchange rate and adapted the refinement lemmas of the parts, we can automatically prove the refinement of the compound program.

$$idxs\_cmp_{arr}\ xs\ i\ j \leq \Downarrow_C\ E_{a\&u}\ (idxs\_cmp\ xs\ i\ j)$$

A more elaborate way collects constraints on the exchange rate and solves them afterward to obtain a unified exchange rate (like $E_{a\&u}$).

For that we can use the fact that the currency refinement using the supremum of two exchange rates yields a bigger computation. We can incorporate that fact into the monotonicity rule for **bind**:

$$[\![\ m \leq \Downarrow_C E_1\ m';\quad (\forall x.\ f\ x \leq \Downarrow_C E_2\ (f'\ x));\quad E = sup\ E_1\ E_2\ ]\!]$$
$$\Longrightarrow\ \texttt{bind}\ m\ f \leq \Downarrow_C E\ (\texttt{bind}\ m'\ f')$$

Using that fact, we can execute the lockstep refinement normally, then obtaining the following final refinement lemma:

$$idxs\_cmp_{arr}\ xs\ i\ j \leq \Downarrow_C E_{syn}\ (idxs\_cmp\ xs\ i\ j)$$

where $E_{syn} = (sup\ (sup\ E_{alu}\ E_{alu})\ E_{uwl})$. In a next step the synthesized exchange rate E_{syn} can be flattened and simplified to obtain $E_{a\&u}$. That process would be automatic and needs less user interaction. Care has to be taken when simplifying those exchange rates, because in general exchange rates might refine the same currencies. Then, one has to take the maximum per currency.

This works as long as there are refinements including currency refinement for all the subprograms. Often it is the case that only some of the operations in a program get refined, but others just stay the same. Say we have some operation A that stays the same, i. e. we would use the trivial $A \leq A$ as the refinement lemma. But we rather need a currency refinement lemma like $A \leq \Downarrow_C E\ A$ with a suitable exchange rate E. Choosing the identity exchange rate $E_{id} = (\lambda c.\ \$_c\ 1)$, which maps every coin of currency c to one coin of currency c, makes $\Downarrow_C E_{id}$ get the identity function and the refinement lemma trivial to prove. Unfortunately, this would overlap with some other exchange rate, e. g. $sup\ E_{alu}\ E_{id}$ will not be $E_{id}(list\_get{:=}\$_{ptr\_offset} + \$_{load})$. Instead, it will simplify to $E_{id}(list\_get{:=}\$_{ptr\_offset} + \$_{load} + \$_{list\_get})$, which has the superfluous *list_get* coin.

That means, choosing E_{id} is too coarse and we need to find a more restricted version for the currency refinement lemma that does not necessarily overlap with other exchanges. We need to choose this exchange rate E such that it exchanges all the currencies that occur in A idempotently and such that it sets all the other currencies to *0*. For that to happen, we need to calculate what currencies are used in A. Calculating that amounts to a fixpoint computation, in case A contains some instance of the `rec` combinator. While this should be doable, we have not yet implemented a solution for that problem.

Nonetheless, while the current approach of finding exchange rates involves manual work, it still is feasible. Careful design of the refinement steps avoiding too many calculations of *combined exchange rates* works well.

8.4.4 Refining Specifications with Exchange Rates

To refine specifications, nothing really changes in contrast to NREST-enat. We can prove the exact same *gwp* rules by reducing them from NREST-ecost to NREST-enat by projection. Some lemmas refining specifications often contain a currency refinement: $m \leq \Downarrow_C E\ (\texttt{spec}\ Q\ T)$. For those cases we define $\downarrow_C E$ to execute an exchange rate E not on an NREST program but on a resource expression, and we get $\Downarrow_C E\ (\texttt{spec}\ Q\ T) = \texttt{spec}\ Q\ (\lambda x.\ \downarrow_C E\ (T\ x))$. Then, we can prove the above goal by routinely proving the inequality $Some\ 0 \leq gwp\ m\ (emb\ Q\ (\lambda x.\ \downarrow_C E\ (T\ x)))$. Recall that, $emb\ Q\ T = (\lambda x.\ \texttt{if}\ Q\ x\ \texttt{then}\ Some\ (T\ x)\ \texttt{else}\ None)$.

8.4.5 Stepwise Refinement with Currencies

Let us compare the structuring with resource currencies with the approach with locales. Consider the following example.

Example 8.4.8. In Example 8.3.9 in NREST-enat we analyzed *idxs_cmp* by inserting the final costs of the compare and lookup operation and then have obtained the following final result:

$$idxs\_cmp\ xs\ i\ j \leq idxs\_cmp_{spec}\ (\lambda\_.\ 5)\ xs\ i\ j$$

In this chapter we have used a more fine-grained analysis and do not exchange all operation costs into a uniform currency. We combine the two refinement steps we already proved (Example 8.4.7 and 8.4.6) with transitivity of the refinement relation and monotonicity of currency refinement. Then we pull the exchange rate into the specification and simplify the term:

$$\begin{aligned} idxs\_cmp_{arr}\ xs\ i\ j &\leq \Downarrow_C\ E_{a\&u}\ (idxs\_cmp\ xs\ i\ j) \\ &\leq \Downarrow_C\ E_{a\&u}\ (\Downarrow_C E_1\ (idxs\_cmp_{spec}\ (\$_{idxs\_cmp})\ xs\ i\ j)) \\ &= \Downarrow_C\ (E_1 \circ_C\ E_{a\&u})\ (idxs\_cmp_{spec}\ (\$_{idxs\_cmp})\ xs\ i\ j) \\ &= idxs\_cmp_{spec}\ (\downarrow_C\ (E_1 \circ_C\ E_{a\&u})\ \$_{idxs\_cmp})\ xs\ i\ j \\ &= idxs\_cmp_{spec}\ (\$_{ptr\_offset}\ 2 + \$_{load}\ 2 + \$_{uword\_less})\ xs\ i\ j \end{aligned}$$

By combining the results in the end, we get the result that the $idxs\_cmp_{arr}$ refines the specification and we obtain a fine-grained upper bound on the resource consumption. We can write it in the parametric notation:

$$\begin{aligned} &(idxs\_cmp_{arr},\ idxs\_cmp_{spec}\ (\$_{ptr\_offset}\ 2 + \$_{load}\ 2 + \$_{uword\_less})) \\ &\quad \in Id \to Id \to Id \to Id \end{aligned}$$

On that level we could now sum over all the currencies to obtain a combined upper bound on all operations. Then, we can apply asymptotic analysis on that expression or on the individual currencies and obtain that $idxs\_cmp_{arr}$ has running time complexity $\Theta(1)$.

When we reconsider the discussion about how to structure refinement in the classical approach (Section 8.1.7) we can now add the approach of using *resource currencies*. When defining a specification $prog_{spec}$, a new currency is invented (called *prog*) and it is assumed that each call of that program incurs one coin of it. Similarly, for all the subprograms that are used in the first iteration $prog_1$. When proving the first refinement we would have to come up with an exchange rate E_1 to prove $prog_1 \leq \Downarrow_C E_1\ prog_{spec}$. With that correctness lemma, we have proved two properties about $prog_1$. Regarding functional correctness, it refines the specification. Regarding resource consumption, we have established an upper bound in the form of a resource expression in the currencies of its subprograms ($E_1\ \$_{prog}$). This is very much like the role of the pseudocode in the observation in Section 7.1 (cf. Figure 7.1): we can argue for the correctness of the algorithm sketch and give its running time dependent on its parts.

We can now carry on along the refinement chain, refining the components and combining the concrete subprograms into a more concrete refinement $prog_2$. Then, we can obtain a refinement lemma for that concrete program by monotonicity, with the additional caveat to merge exchanges rates during that process. In the end, the refinement chain can be collapsed like in the above example to obtain a final correctness theorem.

It is not yet clear to me which approach — NREST-enat with locales or NREST-ecost with resource currencies — is more usable. There is also a third possibility to use NREST-ecost without currency refinement but instead using locales like for NREST-enat. In theory, NREST-ecost supersedes NREST-enat: set $\eta = unit$ instead of $\eta = string$. Practically, both approaches seem to be equally expressive, as instead of using currencies we could just fix constants for them and talk about weighted sums all

the time. Then, projecting to a certain currency amounts to setting its constant to *1* and the constants of all the other currencies to *0*. The currency refinement mechanism, however, feels more natural, as it allows to speak about resource costs at different levels of abstraction. In Chapter 9, we will see that NREST-ecost integrates nicely with the fine-grained cost analysis of LLVM-Time. That would not be possible for NREST-enat.

8.5 Discussion of Alternatives and Other Resources

In this section I discuss alternative definitions of the NREST monad and list some ideas for other resource types γ that might be worth studying.

8.5.1 Alternatives to NREST

In the beginning of this chapter we stated our motivations and design goals for NREST. Besides choosing partial function that map results to an resource element, there are alternatives. In the following, I want to explain why we took our decision the way we did:

A result set and a resource An alternative would be to define an NREST program being a set of results together with a resource usage:

$$(\alpha,\ \gamma)\ NREST_1 = \texttt{fail} \mid \texttt{res}\ (\alpha\ set \times \gamma)$$

Here the problem is that this is way too coarse. Say we have a two stage process that starts with a set of two natural numbers. In each step the program would choose nondeterministically a number and would do work that uses resources linear in that number. Say we start with a set $\{1, 2\}$ then the result after the first step would be $\texttt{res}\ (\{\ \{1\}, \{2\}\ \},\ 2)$, as there are two possibilities which element was removed from the set, and the upper bound on the both outcomes would be *2*. After the second step the result must be $\texttt{res}\ (\{\ \emptyset\ \},\ 4)$, as in both cases the remaining element is removed, but again the upper bound on the running time of that second step is 2, which yields a total running time of 4, which is not tight. In essence, one sees that one can not use nondeterminism effectively when assigning all results the same resource usage.

A set of pairs Another alternative is that one regards the resource usage just as part of the result. Then instead of the result being of type α, we intuitively want to choose type $\alpha \times \gamma$. Observe that $(\alpha \times \gamma)\ set$ is isomorphic to $\alpha \to \gamma\ set$. For later presentation it makes sense to choose the latter. So we define the following alternative to NREST:

$$(\alpha,\ \gamma)\ NREST_2 = \texttt{fail} \mid \texttt{res}\ (\alpha \to \gamma\ set)$$

On the one hand, this definition certainly allows to model the above situation adequately. Depending on which number out of $\{1, 2\}$ was chosen we can specify a different resource consumption and in the end model a tight running time of *3*.

On the other hand, the refinement relation cannot just be the natural subset relation, because we would like to have $\{(x, 3), (x, 4)\} \leq \{(x, 4)\}$, in order to allow refinement with programs with less resource consumption. So before using the subset relation we have to "bake-in" the upper bounding by taking the lower closure and lifting it to the function:

$$lower\_closure\ S' = \{s \mid s' \in S' \wedge s \leq s'\}$$
$$\texttt{res}\ M \leq \texttt{res}\ M' \longleftrightarrow (\forall x.\ M\ x \subseteq lower\_closure\ (M'\ x))$$

So a program $m = \texttt{res}\ M$ refines a program $m' = \texttt{res}\ M'$ if for all results x the set of possible resource costs is bounded by some possible resource bound for element x in M'. When we first considered the design choice on how to model NREST, we dropped that approach because it simply did not feel natural and the alternative approach with maps to single γ elements worked out more smoothly: The refinement ordering in NREST is a simple lifting which directly results in a complete lattice structure and allows for a generalized weakest precondition (*gwp*). In the following I present some results of an effort trying to use the "set of pairs" approach.

First, we note that the ordering defined with the *lower_closure* is not antisymmetric, and thus $NREST_2$ does not have a complete lattice structure. An example would be $a = \texttt{res}\ [()\mapsto\{(1,1),(0,1)\}]$ and $b = \texttt{res}\ [\ () \mapsto \{(1,1),(1,0)\}]$: both $a \leq b$ and $b \leq a$ hold but $a = b$ does not. In order to fix this problem we introduce a quotient type γ *dclosed* for downwards closed sets over γ, where we identify all elements of γ *set* that are equivalent under application of *lower_closure*. With that we define a new variant for NREST:

$$(\alpha, \gamma)\ NREST_3 = \texttt{fail} \mid \texttt{res}\ (\alpha \to \gamma\ dclosed)$$

For this we can define a sensible refinement ordering that gives rise to a complete lattice structure on NREST. Note here, that the definition of the supremum on NREST does not stem from the resource type γ being a complete lattice, but because the set underlying the type γ *dclosed* gives rise to the complete lattice on subsets. Thus, we can even drop the requirement that γ is a complete lattice and only require it to be an order.

We do not need to use *enat* instead of *nat* to obtain a complete lattice. In particular we needed the element ∞ to model the supremum over infinitely many finite numbers (cf. Example 8.3.4). In the current setting *nat dclosed* already allows for infinite sets and their downwards closure. Also, that an result cannot be reached, which was earlier modeled by the partial map or more technically by *None*, is modeled easily now: simply by using the element in α *dclosed* that corresponds to the the empty set.

For a resource type γ that provides a neutral element *0* and addition $+$ with a monoid structure, we further can define the monadic operators `return`, `bind` and `elapse` as expected. For the application of $+$ in `bind`, it must be lifted from γ to γ *dclosed*, which is straightforward.

The pointwise reasoning setup can be adapted and even simplified considerably. We define $inres_\tau :: (\alpha, \gamma)\ NREST_3 \to \alpha \to \gamma \to bool$ uniformly for all γ even for liftings to functions. We can prove all the simplification rules correct and finally prove that

$NREST_3$ fulfills the monad laws. Note that γ needs to fulfill some properties for $+$ and the ordering $\leq$. We have collected those properties but have not yet tried to minimize them or further investigate them.

Instead, we tried to adapt the next concept needed for the framework and failed in doing so. In order to prove specification refinements we would like to adapt the generalized weakest precondition *gwp* (cf. Section 8.3.4) to $NREST_3$. Remember, we want to prove refinements of the form $m \leq$ `res` M. Intuitively, the idea is to find the margin between the computation m and the deadline M. To determine that margin in NREST we used a difference operator (cf. Section 8.3.4). It is not clear how to define that difference operator if there are more than one extreme points in the deadline.

For example, let the resource cost be measured in two currencies, denoted as pairs. Then, consider the element of type $(nat \times nat)$ *dclosed* that corresponds to the set of resources $\{ (2, 0), (0, 2) \}$. It may represent the deadline M_x for some element x. Similarly, let m_x be the downwards closed set for $\{(1, 0), (0, 1)\}$ representing the resource cost of m for element x. In order to obtain *gwp* we would need to take the difference between those two downwards closed sets. It is not clear to me how to define that in a sensible way. One could enforce the existence of exactly one greatest element in the downwards closed sets by a side condition, and pull that through the calculus. Or even use the subtype of γ *dclosed* that has only one extreme element. It would be interesting how that approach would relate to NREST.

In NREST, however, we are forced to aggregate the cost into one element. We would obtain $(2, 2)$ and $(1, 1)$ respectively. In that case, the difference operator is well-defined, and we would obtain $(2, 2) - (1, 1) = (1, 1)$ to be the margin. I have to note that the overapproximation of $\{ (2, 0), (0, 2) \}$ to $(2, 2)$ *does* cause a problem, which we will treat in Section 9.2.4. But that problem does not occur during the usage of stepwise refinement. It only surfaces at the very end of the refinement chain, and can be handled by considering a special case. Luckily, only that special case, i. e. the resource bound only having one extreme point, occurs in our case studies. Furthermore, I expect that further case studies that stem from the analysis of standard algorithms will also fall into that special case.

It seems that measuring the resource consumption in NREST with one greatest element instead of a set of elements does give up some precision. However, it simplifies the calculus and avoids proving side conditions in every step of reasoning. Only at the last step one has to ensure a special case, which seems to be the natural one. Experience shows that this approach works fine. But there are still some open questions.

8.5.2 Other resources

Here, I want to briefly name ideas for other resources types that would fit into the NREST monad.

- Consider $\gamma = bool$: I think with that type one could signal, that some event has not yet happened during a nondeterministic computation. Let $False \leq True$, then *False* might represents that one can ensure that something did not yet occur, and

True says, that something might have happened or rather one does not know. The same domain could be used to model Information Flow Control (IFC). The NREST monad maintains a confidentiality label (*True* for public, and *False* for private) for each result. Those labels get combined during sequential composition. Rajani [122, Part II] discusses an application of a type-theory for higher-order amortized analysis to IFC that seems to be related.

- For any resource type γ one could study the same type with the flipped order. Already for $\gamma = bool$, that would change the semantics to *False* representing "arbitrary" resource cost and *True* representing that something *did* occur. For *enat* and *ecost* flipping the order would mean to consider lower bounds. Still the "most conservative" cost is aggregated, i. e. the infimum over all paths or all abstractions. Through a `bind` one still would add up the lower bounds, to get a new lower bound.

- Combining resources can not only be done with lifting to functions. Also lifting to pairs is feasible. So one could realize intervals on the running time or other resources by maintaining a pair of the original resource and the resource with flipped order.

- Another idea is (α, *string set*) *NREST*, that is any computation has some output string, Supremum is just union, addition is just concatenation lifted to sets. The minus operation could be prefixes, i. e. if the specification Q asks for $\{abc\}$ and the program m gives $\{c, bc\}$ then the minus could be $\{a, ab\}$.

- I tried to come up with a domain for maximum heap or stack size by using a pair (m, c) denoting the maximum and current usage. I succeeded in forming a complete lattice and a monoid structure on that domain. But I could not prove that the plus operation distributes with the supremum. It would be interesting to considered other non-monotone domains.

- The unit interval $[0,1]$ with multiplication is a complete lattice and a monoid with neutral element 1. Compare this to Chapter 4 and the discussion about quantales.

8.6 Summary

- The IRF provides the *nres* monad to model nondeterministic computation. It allows *stepwise program refinement*, as well as *data refinement*. Reasoning infrastructure is provided with the *pointwise reasoning* technique, the *refine_vcg* tactic for *specification refinement*, and the *refine_rcg* tactic for *lockstep refinement*. The *parametric form* allows for succinct notation of refinement lemmas.

- The nondeterminism monad with resources (NREST) models nondeterministic computation with resource consumption. It has at least three meaningful instances: with $\gamma = unit$ we obtain *NREST-bool* which is isomorphic to the classical

nres monad modeling nondeterministic result; with $\gamma = enat$ (*NREST-enat*) we can model nondeterministic result with worst case running time; with $\gamma = ecost$ (*NREST-ecost*) we can also model the consumption of different resources at the same time with so called *resource currencies*.

- All three instances come with the possibility for data refinement, automation for lockstep refinement and specification refinement.
- The *generalized weakest precondition gwp* generalizes the Boolean version and allows stating syntax-directed rules that constitute a verification condition generator.
- The introduction of *resource currencies* allows for fine-grained resource analysis and allows stepwise refinement of resource usage without using contexts.
- NREST can be used to formulate abstract algorithmic ideas, effectively reason about them and further refine those algorithms by adding implementation details later.

9 Synthesis of Implementations

As a next step let us consider how to automatically synthesize programs from abstract algorithms in the NREST monad. In Part II I have presented two semantics of programming languages with a cost model. In this chapter I will show how we can hook up the abstract NREST-monad with both Imperative-HOL-Time and LLVM-Time, and automatically generate implementations that preserve correctness and resource usage claims from the abstract algorithms. This is again joint work with Peter Lammich [45, 44].

The Sepref tool [85] serves as a blueprint for the synthesis. It refines *nres* to vanilla Imperative-HOL. I will present two adaptations: The first iteration hooks up NREST-enat with Imperative-HOL-Time (Section 9.1), the second one hooks up NREST-ecost with LLVM-Time (Section 9.2). While adapting the tool required many but rather straightforward modifications, the underlying theory contains some interesting ideas and insights. The core of the tool is the *translation phase*, where the concrete program is synthesized. In the presentation of the two Sepref instances with resource consumption I focus on that phase as the other phases can be adapted in a straightforward manner. For the description of the other phases I refer to [92, §4.2].

9.1 NREST-enat with Imperative-HOL-Time

Portions of this section appear in the paper "Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL" (Haslbeck and Lammich [45]).

Before we delve into the synthesis phase, reconsider some specifics of the Imperative-HOL-Time semantics from Chapter 5. Let us have a look at the definition of a Hoare triple:

```
1  <P> c <λr. Q r> = (∀h n. abs (h, n) ⊨ P
2                      ⟹ (∃h' t r. (c, h) ⇒ (r, h', t)
3                            ∧ abs (h', n − t) ⊨ Q r ⋆ true
4                            ∧ t ≤ n))
```

Here, *abs* abstracts a pair of an Imperative-HOL-Time heap and time credits into an element of a separation algebra. The assertion *true* is true for any heap, thus enables garbage collection of both time credits *and* heap elements. The Hoare triple $<P>\ c\ <\lambda r.\ Q\ r>$ denotes that procedure c started from a heap satisfying P terminates with a return value r in a resulting heap that satisfies $Q\ r \star true$. In particular it states that the starting heap holds enough time credits n in order to pay for the cost t of executing the procedure c (see line 4).

Note that the time cost t and the amount of time credits n are natural numbers and not extended natural numbers as the costs in the NREST-enat monad. The term $(c,\ h) \Rightarrow (r,\ h',\ t)$ expresses that program c started from heap h takes t steps to produce result r and final heap h'.

I will now present the *synthesis predicate* that connects the two monads. Then, I show how to prove synthesis rules for basic operations and organize a library of such. I will show how the synthesis process composes synthesis rules to synthesize compound algorithms, and how to extract final Hoare triples from those compound synthesis predicates. Finally, I illustrate the process with an example verification of dynamic arrays and I show how they can be used.

9.1.1 Synthesis Predicate

The "**H**eap-monad to **N**on-determinism **R**efinement" predicate $hnr\ \Gamma\ m_\dagger\ \Gamma'\ A\ m$ intuitively expresses that the concrete program $m_\dagger$ computes a concrete result that relates, via the *refinement assertion* A, to a result in the abstract program m, using at most the resources specified by m for that result. A refinement assertion describes how an abstract variable is refined by a concrete value on the heap. It can also contain time credits. The assertions Γ and Γ' constitute the heaps before and after the computation and typically are a separating conjunction of refinement assertions for the respective parameters of $m_\dagger$ and m. Formally, we define:

```
1  hnr :: assn → α Heap → assn → (β → α → assn) → (β, enat) NREST → bool
2  hnr Γ c Γ' A m =
3    nofail m ⟹
4      (∀h n. abs(h, n) ⊨ Γ ⟹
5        (∃h' t r. (c, h) ⇒ (r, h', t)
6          ∧ (∃t_a r_a. abs (h', (n + t_a)−t) ⊨ Γ' ⋆ A r_a r ⋆ true
7            ∧ elapse (return r_a) (enat t_a) ≤ m
8            ∧ t ≤ n + t_a)))
```

If the abstract program m does not fail, procedure c started from a heap satisfying Γ produces a heap satisfying Γ' and a result r which relates to an abstract result r_a via refinement assertion A. The abstract result r_a is a valid result of m and has at least t_a time units reserved for it. Together with the time credits on the heap n, this pays for the execution cost t (line 8).

In particular, the execution cost t is paid for by the time units t_a specified by the abstract program and by time credits n that are hidden in the data structures on the heap. One can see that amortized data structures seamlessly integrate into the framework: only amortized running time costs are visible to the abstract algorithm, while the actual running time and potential is hidden in the implementation.

Note that the advertised cost t_a is a natural number. If the abstract program m reserves infinitely resources for a result r_a that abstract the result of c, there will always be a suitable value for t_a as the running time t of c is known to be finite.

Furthermore, the heap assertion *true* in the post-heap allows to garbage-collect arbitrary portions of the heap: this includes excess time credits as well as arbitrary data structures from the heap.

Example 9.1.1. Let us look at an example and appreciate the role of amortization. We already have seen the verification of dynamic arrays with operation *push_array* in Example 5.5.1, also we have seen the definition of $push\_list_{spec}$ in Example 8.3.2 I repeat both here. First, the Hoare triple for *push_array* is:

$$\begin{array}{l}<dyna_{assn}\ xs\ p \star id_{assn}\ x\ x_\dagger \star \$19> \\ \quad push\_array_{impl}\ x_\dagger\ p \\ <\lambda p'.\ inv\ dyna_{assn}\ xs\ p \star inv\ (id_{assn}\ x\ x_\dagger) \star dyna_{assn}\ (xs \cdot [x])\ p'>\end{array}$$

I have added the assertion $id_{assn}\ x\ x_\dagger$ that describes the empty heap and the pure fact $x = x_\dagger$. For further structuring of Hoare triples the pre-heap should contain a *refinement assertion* relating the abstract and concrete version of each parameter of the operation. Additionally some time credits are needed for the advertised costs. The post-heap should contain a refinement assertion for all the parameters that stay unchanged and a separating conjunct for the result (here $dyna_{assn}\ (xs \cdot [x])\ p'$). Here, both input parameters get eliminated, i. e. together they will be contained in the result. For parameters that get destroyed in the operation an *invalidated* assertion will be added. It represents the empty heap but records that the parameter once existed. This allows to retrieve pure facts from them. Formally we write:

$$inv\ A\ x\ x_\dagger = \uparrow(\exists h.\ h \models A\ x\ x_\dagger)$$

In particular data not stored on the heap and properties about them can be recovered. For example $h \models id_{assn}\ x\ x_\dagger \implies x = x_\dagger$.

Second, the specification $push\_list_{spec}$ is defined as:

$$push\_list_{spec}\ T\ x'\ xs = \texttt{res}\ [xs \cdot [x'] \mapsto T\ xs]$$

We can now come up with a synthesis rule that links the implementation with the abstract operation. It is obtained by unfolding the above Hoare triple and integrating the specification for $push\_list_{spec}$. Let us first look at the unfolded version:

$$\begin{array}{l}
nofail\ (\texttt{res}\ [xs \cdot [x] \mapsto T\ xs]) \implies \\
\quad (\forall h\ n.\ abs\ (h,\ n) \models dyna_{assn}\ xs\ p \star nat_{assn}\ x\ x_\dagger \implies \\
\qquad (\exists h'\ t\ r.\ (push_array_{impl}\ x_\dagger\ p,\ h) \Rightarrow (r,\ h',\ t) \\
\qquad\quad \wedge\ (\exists t_a\ r_a.\ abs\ (h',\ (n + t_a) - t) \models inv\ dyna_{assn}\ xs\ p \star inv\ nat_{assn}\ x\ x_\dagger \\
\qquad\qquad\qquad\qquad\qquad\qquad \star\ dyna_{assn}\ r_a\ r \star true \\
\qquad\qquad \wedge\ \texttt{elapse}\ (\texttt{return}\ r_a)\ (enat\ t_a) \leq \texttt{res}\ [xs \cdot [x] \mapsto T\ xs] \\
\qquad\qquad \wedge\ t \leq n + t_a)))
\end{array}$$

Let the heap h and time credits n that entail the assertion on the pre-heap. This means, that n contains time credits corresponding to the potential stored in the amortized data structure dynamic array. Observe that n does not yet contain the advertised cost *19* of the Hoare triple. We have to provide at least as much credits to apply the Hoare triple above and denote the extra time credits needed by t_a. Once we applied

the Hoare rule, we know that $push\_array_{impl}$ terminates after t steps with concrete result r and post-heap h'. That post heap h' together with the remaining time credits $(n + t_a) - t$ entails the post assertion. It contains what is left of the input parameters and a separating conjunct that relates some abstract result r_a with the concrete result r via some refinement assertion. We now have conditions on the abstract result r_a and the advertised running time t_a. Line 6 expresses that the abstract result r_a is the specification $xs \cdot [x]$ and that t_a is at most $T\ xs$. But it serves as an upper bound for the choice of t_a. Line 7 states that $n + t_a$ must be able to pay for the real cost t. Thus it serves as a lower bound for t_a. The existence of an abstract t_a in that interval can be made sure by requiring the additional premise $19 \leq T\ xs$.

In that technique we use the time credits from the amortized data structure together with the advertised costs expressed in the monadic program to pay for the cost of the concrete implementation.

We can now fold the definition of *hnr* and obtain the synthesis rule for $dyna_{assn}$:

$$\begin{aligned} 19 \leq t\ xs \implies\ & hnr\ (dyna_{assn}\ xs\ p \star id_{assn}\ x\ x_\dagger)\ (push\_array_{impl}\ x_\dagger\ p) \\ & (inv\ dyna_{assn}\ xs\ p \star inv\ id_{assn}\ x\ x_\dagger)\ dyna_{assn}\ (push\_list_{spec}\ t\ x\ xs) \end{aligned}$$

Here, $dyna_{assn}\ xs\ p$ is the representation predicate relating the pointer p pointing to an dynamic array with the abstract list xs it represents. Typically, for each pair of abstract and concrete parameters there is an assertion on the pre and post-heap. The synthesis predicate expresses that if the parameters of the implementation $push\_array_{impl}$ refine the parameters of the abstract operation $push\_list_{spec}$, the resulting dynamic array refines the abstract result. The abstract operation expects the reserved time in parameter t dependent on the abstract parameters. The synthesis rule only holds if there is enough time reserved in the abstract algorithm. The premise ensures that.

In order to validate that this definition makes sense, observe what we can prove for it: First, we can extract Hoare triples from synthesis rules and vice versa if the synthesis rules have a specific form. Second, this definition enables us to prove soundness of synthesis rules for all the combinators in particular for `bind`. Before we see those combinator rules let us examine the connection to Hoare triples.

9.1.2 Connection to Hoare Triples

For programs with specifications of the special form `spec` $P\ (\lambda\_.\ t)$, we can extract a standard Hoare triple from a valid synthesis predicate and vice versa:

$$\begin{aligned} & hnr\ \Gamma\ c\ \Gamma'\ A\ (\texttt{spec}\ P\ (\lambda\_.\ t)) \\ & \quad \longleftrightarrow\ <\Gamma \star \$\,t>\ c\ <\lambda r_\dagger.\ \Gamma' \star (\exists_A\ r.\ A\ r\ r_\dagger \star \uparrow(P\ r))> \end{aligned}$$

While during reasoning the abstract time bound needs to depend on the result (e. g. in order to prove the synthesis rule for bind correct), when proving the running time of an algorithm, in most cases the final running time only depends on the input parameters.

This equivalence has two consequences. First, after synthesis of a program from an abstract algorithm, we can extract a Hoare Triple that only uses concepts from the underlying program semantics but is independent from the theory about NREST. NREST

only serves as a means of structuring the proofs. The algorithm as well as its properties are expressed in terms of the semantics of the language and its Separation Logic. Second, we can prove synthesis predicates for basic abstract operations, by providing concrete implementations, setting up the appropriate synthesis rule, converting it into a Hoare triple and using infrastructure of the target monad to prove those.

Example 9.1.2. The specification $push\_list_{spec}$ has that form:

$$push\_list_{spec}\ T\ x'\ xs = \texttt{spec}\ (\lambda r.\ r = xs \cdot [x'])\ (\lambda\_.\ T\ xs)$$

So we can retrieve the Hoare triple from the synthesis rule we just derived.

$$\begin{aligned}&<dyna_{assn}\ xs\ p \star id_{assn}\ x\ x_\dagger \star \$19>\\ &\quad push\_array_{impl}\ x_\dagger\ p\\ &<\lambda r_\dagger.\ inv\ dyna_{assn}\ xs\ p \star inv\ id_{assn}\ x\ x_\dagger\\ &\qquad \star (\exists_A\ r.\ dyna_{assn}\ r\ r_\dagger \star \uparrow(r = xs \cdot [x'])) >\end{aligned}$$

That Hoare triple simplifies to the one we just had for $push\_array_{impl}$. Or conversely we could have proved the synthesis rule from the Hoare triple.

What prevents us from defining the synthesis predicate relative to a Hoare triple is the fact that the running time needs to be able to depend on the result. While the classical — non-negative — time credits need to be stated in the pre-heap, integer time credits may solve the problem. Negative time credits in the post-heap are equivalent to positive time credits in the pre-heap but additionally may depend on the result of the computation. Using integer time credits instead might simplify the synthesis predicate considerably.

In the following we will present how we organize basic abstract operations, and provide implementations with concrete data structures and respective synthesis rules.

9.1.3 Organizing Abstract Operations

Synthesis rules have the following general form:

$$\begin{aligned}&P\ (x_{1\dagger},\ldots,x_{\dagger n})\ (x_1,\ldots,x_n) \implies\\ &\quad hnr\ (A_1\ x_{1\dagger}\ x_1 \star \ldots \star A_n\ x_{n\dagger}\ x_n)\ (X_{impl}\ (x_{1\dagger},\ \ldots\ ,x_{n\dagger}))\\ &\qquad (A_1{}^{p_1}\ x_{1\dagger}\ x_1 \star \ldots \star A_n^{p_n}\ x_{n\dagger}\ x_n)\ A\ (X\ (x_1,\ \ldots\ ,x_n))\end{aligned}$$

Here $p_i \in \{k,\ d\}$ signals whether the data structure is kept or destroyed during the computation. In particular, $A_i^k = A_i$ and $A_i^d = inv\ A_i$. The synthesis rule is only valid subject the premise P.

Once we have ensured that more restricted structure of the synthesis rules, we can intro a more succinct notation.[1] With that notation the rule above can be written in the following way:

$$(X_{impl},\ X_{spec}) \in [P]\ A_1^{p_1} \to \ldots \to A_n^{p_n} \to A$$

[1] The notation is introduced in [84, §5.1].

The notation is inspired by relational parametricity rules. It follows the same ideas as the notation for refinements from Section 8.1.2. It shows that if the parameters of the implementation X_{impl} and the specification X_{spec} are related by the refinement assertions A_i, then the result is related by the resulting refinement assertion A. The superscripts of the refinement assertions indicate whether the parameter will be kept on the heap or destroyed during the operation. Those synthesis rules can also have additional preconditions.

Example 9.1.3. We can write the synthesis rule of the above example in that notation:

$$(push\_array_{impl},\ push\_list_{spec}\ T)$$
$$\in [\lambda\_\ (xs,\_).\ 19 \leq T\ xs]\ {dyna_{assn}}^d \times {id_{assn}}^d \to dyna_{assn}$$

Both parameters — the dynamic array and the element to push to it — are destroyed during the execution of the algorithm, but the resulting heap contains the new dynamic array after pushing the element. As the added element was a pure element, one can retrieve the information about it in the post-heap. The allotted time bound T only depends on the first abstract parameter, and the lower bound for it stemming from the implementation in Imperative-HOL is enforced by the precondition.

Stating synthesis rules in this more restricted form simplifies their usage for the Sepref tool. Furthermore, it stresses how they can be composed with refinements. Consider the following example:

Example 9.1.4. In Example 5.5.1 we proved the Hoare triple for $push\_array_{impl}$ mentioned earlier in this section in several steps. First proving the implementation correct w. r. t. an implementation on dynamic lists. Then abstracting the dynamic list to a list. We now want to illustrate the composition of a synthesis rule and an NREST-enat refinement.

From the intermediate Hoare triple from Example 5.5.1 we obtain the following synthesis rule:

$$(push\_array_{impl},\ \lambda(xs,\ n)\ x.\ \texttt{res}\ [push\_array_{fun}\ (xs,\ n)\ x \mapsto 19])$$
$$\in {dyn\_array_{assn}}^d \to Id^d \to dyn\_array_{assn}$$

In Example 8.3.2 we have obtained the following correctness lemma involving a data refinement:

$$(\lambda(xs,\ n)\ x.\ \texttt{res}\ [push\_array_{fun}\ (xs,\ n)\ x \mapsto T\ xs],\ push\_list_{spec}\ T)$$
$$\in R^{list}_{dynlist} \to Id \to R^{list}_{dynlist}$$

If we combine the refinement assertion dyn_array_{assn} and the refinement relation $R^{list}_{dynlist}$ we obtain the compound refinement assertion $R^{list}_{dynlist}$:

$$dyn\_array_{assn} \circ_{AR} R^{list}_{dynlist} = dyna_{assn}$$

The operator is defined by $(A \circ_{AR} R)\ a\ c = (\exists_A b.\ A\ b\ c \star \uparrow(\ (b,\ a) \in R))$.

Note that for historic reasons (already in the Separation Logic for Imperative-HOL [85]) assertions take the abstract argument as the first argument. In contrast to that, the refinement relation (e. g. $R^{list}_{dynlist}$) takes a pair where the abstract element is the second component. This inconsistency sometimes leads to confusion. For this thesis I decided to stay truthful to the Isabelle theories and to not switch the order in presentation. Future work consolidating the IRF should streamline this inconvenience.

Remember that $R^{list}_{dynlist}$ and *dyn_abs* are related in the following way:

$$(c,\ a) \in R^{list}_{dynlist} \longleftrightarrow dyn\_abs\ c\ a$$

To check that the composition is equal to the definition of dyna_{assn}, here again see the of $dyna_{assn}$, which was:

$$dyna_{assn}\ as\ p = (\exists_A bs\ n.\ dyn\_array_{assn}\ (bs,\ n)\ p \star \uparrow(dyn\_abs\ (bs,\ n)\ as))$$

So finally we can compose the above synthesis rule and the correctness lemma to obtain a new synthesis rule with the more abstract result:

$$\begin{aligned}&(push\_array_{impl},\ \lambda(xs,\ n)\ x.\ push\_list_{spec}\ (\lambda.\ 19))\\ &\quad \in (dyn\_array_{assn} \circ_{AR} R^{list}_{dynlist})^d \to (id_{assn} \circ_{AR} Id)^d\\ &\qquad \to (dyn\_array_{assn} \circ_{AR} R^{list}_{dynlist})\end{aligned}$$

Which also simplifies to the synthesis rule for $push\_array_{impl}$ we already proved in the example earlier this Section.

As a general pattern, the correctness theorem for an NREST-enat program X can be combined with a refinement lemma in the following way:

$$\begin{aligned}&[\![\ (X_{impl},\ X\ T) \in [\lambda x_\dagger\ x.\ T_\dagger\ x_\dagger \le T\ x]\ A_1^{p_1} \to \ldots \to A_n^{p_n} \to A;\\ &\quad (X\ T,\ X_{spec}\ T') \in [\lambda x\ x'.\ T\ x \le T'\ x']\ R_1 \to \ldots \to R_n \to R\ ]\!]\\ &\quad \Longrightarrow\ (X_{impl},\ X_{spec}\ T') \in [\lambda x_\dagger\ x'.\ T_\dagger\ x_\dagger \le T'\ x']\\ &\qquad (A_1 \circ_{AR} R_1)^{p_1} \to \ldots \to (A_n \circ_{AR} R_n)^{p_n} \to (A \circ_{AR} R)\end{aligned}$$

The parameter $T_\dagger$ is an implementation-dependent bound for the running time of the program X_{impl}. When modeling the abstract program X we need to choose a time bound T for it. This happens by filling the time parameters of the locale it lives in. The precondition of the first premise ensures that the time bound T is at most $T_\dagger$. The time bound T' for the specification X_{spec} depends on the time parameters of the locale. When establishing the second premise we have to prove that T' is at most T. The combination of those two preconditions gives the precondition of the resulting synthesis rule. In the original framework, the *FCOMP tool* [82, §3.3.1] automates the combination of synthesis rules and *nres* refinements. It was adapted for our purposes.

In Example 5.5.1 dynamic arrays were verified in Imperative-HOL-Time and connected to a list representation. Now, in Example 9.1.4 we have only used the Imperative-HOL-Time implementation relative to a dynamic list representation and composed it with a data refinement in NREST-enat between dynamic lists and lists. The overall goal is to move as much reasoning from the concrete program semantics (here Imperative-HOL-Time) to the abstract modeling in NREST. In Section 10.2 we will see that also

abstract	operations	running time	concrete
matrix	create; lookup, update	$O(n^2)$; $O(1)$	array
set/map	create; insert, delete, update	$O(1)$; $O(\log n)$	red-black tree
		$O(n)$; $O(1)$	array
list	create, append; lookup, update	$O(1)^*$; $O(1)$	dynamic array
disjoint sets	create; union, find	$O(n)$; $O(\alpha(n))^*$	union-find

Table 9.1: The abstract data structures with abstract operations that we provide implementations for in the TIICF. Amortized running time bounds are marked with an asterisk (*).

the amortized reasoning for the push operation of dynamic lists can be conducted on the NREST level.

The Isabelle Imperative Collections Framework Table 9.1 lists abstract data structures with their abstract operations and the implementations we currently provide in the *Timed Imperative Isabelle Collections Framework* (TIICF).

9.1.4 The Synthesis Process

With basic synthesis rules for the abstract operations in place, we want to automatically synthesize programs from abstract algorithms that combine those operations. This process extends surprisingly seamless from the original Sepref tool to reasoning about resources.

The translation works by symbolically executing the abstract program, thereby synthesizing a structurally similar concrete program. During the symbolic execution, the relation between the abstract and concrete variables is modeled by refinement assertions. In the synthesis predicate $hnr\ \Gamma\ m_\dagger\ \Gamma'\ A\ m$, the pre-heap Γ contains the refinements for the variables before the execution, Γ' contains the refinements after the execution, and A is the refinement assertion for the result of m. For example, a `bind` is processed by the following synthesis rule:

```
1   ⟦ hnr Γ m↑ Γ' A_x m;
2     (∀x x↑. hnr (A_x x x↑ ⋆ Γ') (f↑ x↑) (A'_x x x↑ ⋆ Γ'') R_y (f x)) ⟧
3   ⟹ hnr Γ (do {x↑ ← m↑; f↑ x↑}) Γ'' A_y (do {x ← m; f x})
```

To refine $x \leftarrow m;\ f\ x$, we first execute m, synthesizing the concrete program $m_\dagger$ (line 1). The state after m is $R_x\ x\ x_\dagger \star \Gamma'$, where x is the result created by m. From this state, we execute $f\ x$ (line 2). The new state is $R'_x\ x\ x_\dagger \star \Gamma'' \star R_y\ y\ y_\dagger$, where y is the result of $f\ x$. Note that $x_\dagger$ goes out of scope but is still on the heap. It is contained in the *true* portion of the postheap in the synthesis rule. You can think of it being taken care of by the garbage collector.

While executing the abstract program, not only a concrete program is created, but also the set of refinement assertions Γ evolves: it contains all the data structures (pure or on the heap) that the concrete program maintains.

All the other combinators (`rec`, `while`, `if`, `case`, ...) have similar rules that are used to decompose an abstract program into parts, to synthesize corresponding concrete parts recursively and to combine them afterwards.

At the leaves of this decomposition one has to find "atomic" abstract operations, with a suitable synthesis rule. We have seen examples earlier.

Example 9.1.5. To illustrate this process let us just synthesize a program for the operation that pushes an element to a list and then returns the length of the resulting list. For this example let us assume we have a synthesis rule for a program determining the length of a list.

$$(len\_array_{impl},\ len_{spec}\ (\lambda.\ 1)) \in {dyna_{assn}}^k \to Id$$

We define the following monadic program and want to synthesize an Imperative-HOL-Time program from it:

$$\begin{array}{l} len\_push\ (as,\ a) = \texttt{do}\ \{ \\ \quad as' \leftarrow push\_list_{spec}\ (\lambda.\ 19)\ as\ a; \\ \quad length_{spec}\ (\lambda.\ 1)\ as' \\ \} \end{array}$$

To start the synthesis, we assume that we have an implementation of the input list xs in the form of a dynamic array and the elements to push are refined by themselves. The synthesis goal to solve is:

$$hnr\ (dyna_{assn}\ as\ p \star id_{assn}\ a_\dagger\ a)\ ?c\ ?\Gamma'\ ?A\ (len\_push\ (as,\ a))$$

Here the program c, the post-heap Γ' and the refinement assertion A for the result are unknown in advance. This is indicated with the leading $?$. Those variables will be generated in the synthesis process. Let us denote the pre-heap by Γ. We now apply the synthesis rule for `bind` and have to solve the synthesis goal for the first part:

$$hnr\ (dyna_{assn}\ as\ p \star id_{assn}\ a\ a_\dagger)\ ?c_x\ ?\Gamma'_x\ ?A_x\ (push\_list_{spec}\ (\lambda.\ 19)\ as\ a)$$

We use the synthesis rule for $push\_list_{spec}$ to obtain $c_x = push\_array_{impl}\ p\ a_\dagger$, the post-heap ${\Gamma_x}' = inv\ dyna_{assn}\ as\ p \star inv\ id_{assn}\ a\ a_\dagger$, and the refinement assertion for the result $A_x = dyna_{assn}$. Note that before applying the synthesis rule of $push\_list_{spec}\ as\ a$, frame inference is used on the pre-heap Γ to find the portions of the heap that contain the implementations of the parameters as and a. Now we obtain the pair of abstract and concrete intermediate results (as' and $as'_\dagger$), and we have to synthesize the program for the second push operation starting from pre-heap $A_x\ as'\ as'_\dagger \star \Gamma'_x$. Thus the next synthesis goal is:

$$\begin{array}{l} hnr\ (dyna_{assn}\ as'\ as'_\dagger \star inv\ dyna_{assn}\ as\ p \star inv\ id_{assn}\ a_\dagger\ a) \\ \qquad ?c_f\ ?\Gamma'_f\ ?A_f\ (len_{spec}\ (\lambda.\ 1)\ as') \end{array}$$

Again, after finding the heap portion that contains the abstract parameter as', we now apply the synthesis rule for $length_{spec}$. This yields the program $c_f = len\_array_{impl}\ as'_\dagger$, the final post-heap $\Gamma'_f = dyna_{assn}\ as'\ as'_\dagger \star inv\ dyna_{assn}\ as\ p \star inv\ id_{assn}\ a\ a_\dagger$, and the refinement assertion for the result $A_f = id_{assn}$.

Now, the intermediate result as' goes out of scope and we need to get rid of its portion on the heap $dyna_{assn}\ as'\ as'_\dagger$. We can push it into the *true* portion of the post-heap of the synthesis rule. That way, we ensure that only portions of the heap that correspond to the parameters in scope are valid in the symbolic heap.

In the next section we will not be able to discard arbitrary portions of the heap and Sepref will have to explicitly deallocate the intermediate result. Here, the garbage collector takes care of it. Both input parameters as and a are invalidated on the post-heap, and the result is contained in the result assertion A_f.

The final synthesis predicate thus is:

$$(len\_push_{impl},\ len\_push) \in dyna_{assn}{}^d \to id_{assn}{}^d \ \to id_{assn}$$

with $len\_push_{impl}\ (p,\ a_\dagger) = \texttt{do}\ \{\ as'_\dagger \leftarrow push\_array_{impl}\ p\ a_\dagger;\ len\_array_{impl}\ as'_\dagger\ \}$.

The Sepref tool registers a set of synthesis rules for all combinators and keeps a set of synthesis rules for the "atomic" abstract operations. Those sets of rules can be extended later: if we use Sepref to synthesize an implementation from an abstract algorithm that comes with a refinement proof w. r. t. to some specification, we can combine those two refinement lemmas into a synthesis rule and register it with Sepref. That way, one can grow those sets of rules and reuse implemented abstract algorithms when synthesizing larger programs. The synthesis can be modularized and need not be run on the whole algorithm once, but can be run on parts. The resulting synthesis rules can be saved for later use.

In particular, it is notable that during the synthesis process the Sepref tool has to check, whether the timing precondition is fulfilled, i. e. whether the reserved time for any abstract operation suffices to cover the costs of the implementation. In general proving those inequalities can be quite hard. The tool tries to discharge them automatically and, if this fails, leaves them as a proof obligation for the user. In the following we give an example of a synthesis procedure where a nontrivial timing side condition occurs.

9.1.5 Case Study: Removing Duplicates

In earlier examples we have seen how to specify $push\_list_{spec}$ (Example 8.3.2), how to implement it with a concrete program (Example 5.5.1) and how to prove a synthesis rule for it (cf. Example 9.1.1). In the following I want to illustrate how this can be used in a more complex algorithm. Note that we are working here in the NREST-enat, so no resource currencies are used and we have to use locales in order to structure the refinement of resource consumption.

The abstract operation $push\_list_{spec}$ can now be used when specifying an abstract algorithm. Then, a concrete time function T can be specified, which is used to determine the overall cost of the algorithm.

We now want to implement the removal of duplicates from a list. First we specify that operation:

$$remdups_{spec}\ T\ as = \texttt{spec}\ (\lambda ys.\ set\ ys = set\ as \wedge distinct\ ys)\ T$$

```
1   remdups as = do {
2     ys ← empty_list_spec t_el;
3     S ← set_empty_spec t_es;
4     (zs, ys, S) ← while (λ(xs, ys, S). |xs| > 0) (λ(xs, ys, S). do {
5       assert (|xs| > 0 ∧ |xs| + |ys| ≤ |as| ∧ |S| ≤ |ys|);
6       (x, xs) ← return (hd xs, tl xs);
7       b ← set_member_spec (λ_. t_sm |as|) x S;
8       if b then
9         return (xs, ys, S)
10      else do {
11        S ← set_insert_spec (λ_. t_si |as|) x S;
12        ys ← push_list_spec (λ_. t_pl) x ys;
13        return (xs, ys, S)
14      }
15    }) (as, ys, S);
16    return ys
17  }
```

Figure 9.1: An algorithm in NREST-enat for removing duplicates from a list.

Consider the following program to remove duplicates from a list (Figure 9.1). It lives in an environment where we fix the cost for initializing an empty list t_{el}, initializing an empty set t_{es}, testing for set membership ($\lambda n.\ t_{sm}\ n$), inserting into a set ($\lambda n.\ t_{si}\ n$) and the cost for pushing an element to the back of a list t_{pl}.

The idea of the algorithm in Figure 9.1 is to iterate through the list, while maintaining the set of already encountered elements. Only new elements will be pushed to an accumulator list. Observe that the costs for the set operations in line 7 and 11 are bounded not dependent on the current size of the set ($|S|$) but on the length of the input list *as*. This is an upper bounding that we make explicitly at this level of abstraction. That way, this allows to give a closed form for the time bound of *remdups* that only depends on the input parameter. Note that at this point the upper bounding does not require proving any property. The running time bounds are just assumed and thus can be combined to the total cost for *remdups*. We can prove the correctness lemma for *remdups* by specification refinement:

$$remdups_{time}\ n = n * (t_{sm}\ n + t_{si}\ n + t_{pl}) + t_{el} + t_{es}$$
$$(remdups,\ remdups_{spec}\ (\lambda\_.\ remdups_{time}\ |as|)) \in Id \to Id$$

The program uses $push\_list_{spec}$ from above as well as other abstract operations with corresponding reserved running time bounds. For example insertion into a set:

$$set\_insert_{spec}\ t\ x\ S = \ \mathtt{res}\ [S \cup \{x\} \mapsto t\ S]$$

For each operation in the program some time is reserved. The overall running time of the program is a function of these reserved quantities. Before synthesizing

an Imperative-HOL-Time program we have to provide values for those reserved quantities. On the one hand, those quantities have to be as small as possible in order to allow for a tight time bound for the overall algorithm. On the other hand, those quantities have to be at least the time bound the implementation needs. We now set the following quantities in order to implement the set by a red-black tree and the list by a dynamic array:

$t_{el} = 12 \quad t_{pl} = 19$
$t_{es} = 1 \quad t_{sm}\ n = rbt\_search_{time}\ n \quad t_{si}\ n = rbt\_insert_{time}\ n$

Here, $rbt\_search_{time}\ n$ and $rbt\_insert_{time}\ n$ are the running time bounds for searching and inserting in a red-black tree for a tree of with n elements. Both are monotone and in $O(\log n)$. On the one hand we can now use the automation from Chapter 5 to automatically prove that the overall running time of *remdups* is in $O(n * \log n)$.

$remdups_{time} \in \Theta(n * \log n)$

On the other hand we can synthesize an Imperative-HOL-Time program using the adapted Sepref. During that process, the synthesis rules will be applied and their preconditions must be discharged. For the $push\_list_{spec}$ this boils down to the trivial check $19 \leq 19$. The synthesis rule for $push\_list_{impl}$ (cf. Example 9.1.1) only displays the amortized cost, while the amortization is hidden at this level. Amortized data structures seamlessly can be modeled using time credits, and this comfort extends to also be available for the abstract algorithm. At the abstract level, an amortized data structure behaves just as a normal data structure does.

Let us consider a more interesting operation. The synthesis rule of the red-black tree implementation of $insert\_set_{spec}$ is the following:

$(rbt\_set\_insert,\ set\_insert_{spec}\ t) \in$
$\quad [\lambda S.\ rbt\_insert_time\ (|S|) \leq t\ S]\ {id_{assn}}^d \to {rbt\_set_{assn}}^d \to rbt\_set_{assn}$

Here, the refinement assertion $rbt\_set_{assn}\ S\ p$ relates a set S with a red-black tree at address p. During synthesis the Sepref tool has to check whether there is enough reserved time for the set insertion. I have added the relevant information that is known from the assertions (line 5) at that point.

$|S| \leq |ys| \wedge |xs| + |ys| \leq |as|$
$\quad \Longrightarrow\ rbt\_insert_{time}\ (|S|) \leq (\lambda\_.\ rbt\_insert_{time}\ (|as|))\ S$

The goal can be discharged with the knowledge from the premises that stem from the assertions in the program and the monotonicity of rbt_insert_{time}. This goal may not be solved automatically by the Sepref tool initially. Either the user provides some hints (e. g. the monotonicity lemma) or proves the timing side condition after the synthesis.

From Sepref we will obtain an implementation for *remdups* with a synthesis predicate. We can combine that with the refinement lemma for *remdups* and obtain the following synthesis predicate:

$(remdups_{impl},\ remdups_{spec}\ (\lambda\_.\ remdups_{time}\ |as|)) \in {id_{assn}}^d \to\ dyna_{assn}$

Finally, we obtain a Hoare triple and massage it into the following form:

$$<\$\,(remdups_{time}\ |as|)>\ remdups_{impl}\ as\ <\lambda ys.\ set\ ys = set\ as \wedge distinct\ ys>$$

That Hoare triple can be easily interpreted. Actually one need not know anything about the NREST monad or the synthesis process. To believe in the final judgment, one only has to inspect the definition of Hoare triples and the program logic of Imperative-HOL-Time.

9.1.6 Recap

Let me comment on some specifics of Sepref, how much manual interaction generally is required in this process, and what some limitations are.

The Sepref tool maintains a set of synthesis rules for implementations of operations of abstract data structures. For example, we have seen the synthesis rule connecting $push\_array_{impl}$ and $push\_list_{spec}$. There might be a different synthesis rule that implements $push\_list_{spec}$ by a different data structure (e. g. a linked list). While it is possible to let Sepref choose which implementation to use for each abstract data structure, it is more efficient to avoid the backtracking and manually annotate the choice. This is done by rewriting the initialization of a data structure (e. g. $empty\_list_{spec}$ in line 2) with a synonym (e. g. $empty\_array_{spec}$) and only register the synthesis rule for the implementation relative to the synonym. Once the data structure is initialized on the symbolic heap, frame inference will find it during synthesis and only one synthesis rule will be applicable.

With that restriction Sepref is deterministic and synthesizes an implementation by traversing the abstract program once. In that process side conditions are generated that are automatically discharged or left to the user as final proof obligations. In general, it is favorable to add the preconditions as assertions in the abstract program. This takes the work off the Sepref tool and moves the proof burden to the refinement proofs that typically are more interactive. Only for the timing side conditions this is not possible, as they will be inserted with the synthesis rules and are not known to the abstract program. Registering special rules with Sepref that solve those side conditions and adding their premises as assertions in the abstract program usually works well.

When inspecting the Hoare triple and the program *remdups* it occurs that the splitting of the first element of the list in line 6 does not incur time costs. As commented in Chapter 5, the cost semantics of Imperative-HOL-Time only counts the operations that involve arrays and references. One has to interpret Hoare triples in Imperative-HOL-Time always relative to that premise. The cost semantics of LLVM-Time, however, does not allow for functional operations and leads to a more faithful running time analysis. Furthermore, it is not clear when and how exactly the memory of the input list of *remdups* is cleared by the garbage collector. For LLVM-Time we have to explicitly free the memory. This also has to happen during synthesis when an intermediate result goes out of scope. In the next section, I will show how amortization again allows for an elegant solution.

9.2 NREST-ecost with LLVM-Time

Portions of this section appear in the paper "For a Few Dollars More – Verified Fine-Grained Algorithm Analysis Down to LLVM" (Haslbeck and Lammich [44]).

Now let us turn to synthesizing programs in LLVM-Time from abstract algorithms in the NREST-monad with resource currencies. First, I will note some peculiarities of LLVM-Time. Then, I will present how the synthesis predicate is defined in this instance, comment on the *attains-sup* effect, and show how the ***bind*** rule has to be altered in order to cope with the fact that LLVM does not allow arbitrary garbage-collection of heap portions.

As in the last section, the synthesis predicate joining NREST-ecost with LLVM-Time can not be defined on Hoare triples of LLVM-Time. Instead we had to go one semantic level deeper and I repeat their definition from Chapter 6 here.

Hoare triples of the LLVM semantics are defined in the general *wp* framework (Section 6.2.3) relative to the definition of a weakest precondition predicate *wp* and a garbage collection assertion $\mathbb{T}$ together with some properties about them. Based on these, a *Hoare triple* $\{P\}\ c\ \{Q\}$ hold iff:

$$\forall F\ s.\ (P \star F)\ (abs\ s) \implies wp\ c\ (\lambda r\ s'.\ (Q\ r \star \mathbb{T} \star F)\ (abs\ s'))\ s$$

To instantiate this generic framework to our LLVM cost model, states are pairs of a heap and a resource cost function $s = (m,\ c)$, where c is of type $ecost = string \to enat$. The type of the cost credits has two consequences: first this allows for resource currencies, and second this allows to have infinite resource credits. The latter allows us to write down Hoare triples that only focus on functional correctness and on certain currencies, by adding infinitely many time credits of the currencies to ignore in the precondition.

The type of the time credits in the LLVM semantics thus is the same as in the NREST-monad. When refining NREST to Imperative-HOL-Time we did not have this situation, as Imperative-HOL-Time only allowed finite time credits but NREST-enat did allow infinite costs.

The generic framework we instantiated for the LLVM cost model can of course be instantiated for other program semantics. In the following we present how the synthesis works for LLVM, but it actually serves as a blueprint how to treat other instances in a similar way.

Before delving into the definition of the synthesis rule, there is another peculiarity of LLVM to consider.

9.2.1 Extra Combinators

In contrast to Imperative-HOL-Time we chose to design control flow to actually have costs in LLVM-Time. In Imperative-HOL-Time branching and recursive calls do not incur any costs. This is different here.

We defined the combinators for branching, recursion and while loops that do not incur time. As the respective combinators of the target LLVM *do* incur cost, we define resource-aware variants. Furthermore, we also derive a while combinator:

$$\texttt{if}_c\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 = \texttt{elapse}\ (r \leftarrow b;\ \texttt{if}\ r\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ \$_{if}$$
$$\texttt{rec}_c\ F\ x = \texttt{elapse}\ (\texttt{rec}\ (\lambda D\ x.\ F\ (\lambda x.\ \texttt{elapse}\ (D\ x)\ \$_{call})\ x)\ x)\ \$_{call}$$
$$\texttt{while}_c\ b\ f\ s = \texttt{rec}_c\ (\lambda D\ s.\ \texttt{if}_c\ b\ s\ \texttt{then}\ s \leftarrow f\ s;\ D\ s\ \texttt{else}\ \texttt{return}\ s)\ s$$

Here, the guard of $\texttt{if}_c$ is a computation itself, and it consumes an additional *if* coin to account for the conditional branching in the target model. Similarly, every recursive call consumes an additional *call* coin.

Those definitions are actually used when presenting abstract algorithms in NREST-ecost. Only programs that contain only the resource-aware variants and none of the original ones can be synthesized to implementations. Similarly, we will only be able to show synthesis rules for the resource-aware variants and not for the original ones.

9.2.2 Synthesis Predicate

As in the last section, the synthesis predicate $hnr\ \Gamma\ m_\dagger\ \Gamma'\ A\ m$ intuitively expresses that the concrete program $m_\dagger$ computes a concrete result that relates, via the *refinement assertion* A, to a result in the abstract program m, using at most the resources specified by m for that result. A refinement assertion describes how an abstract variable is refined by a concrete value on the heap. It can also contain time credits.

The assertions Γ and Γ' constitute the heaps before and after the computation and typically are a separating conjunction of refinement assertions for the respective parameters of $m_\dagger$ and m. Formally, we now define:

$$\begin{aligned}
&hnr :: assn \to \alpha\ M \to assn \to (\beta \to \alpha \to assn) \to (\beta,\ ecost)\ NREST \to bool\\
&hnr\ \Gamma\ m_\dagger\ \Gamma'\ A\ m = m \neq \texttt{fail} \implies\\
&\quad (\forall F\ s\ c.\ (\Gamma \star F)\ (abs_m\ s,c) \implies\\
&\qquad (\exists r_a\ c_a.\ \texttt{elapse}\ (\texttt{return}\ r_a)\ c_a \leq m\\
&\qquad\quad \wedge\ wp\ m_\dagger\ (\lambda r\ (s',c').\ (\Gamma' \star A\ r_a\ r \star F \star \mathbb{T})\ (abs_m\ s',c'))\ (s,\ c{+}c_a)))
\end{aligned}$$

The predicate holds if either the abstract program fails or for all heaps and resources $(s,\ c)$ that satisfy the pre-assertion Γ with some frame F, there exists an abstract result and cost $(r_a,\ c_a)$ that refine m, and $m_\dagger$ terminates with concrete result r in a state s' which is described by Γ' and the frame F, and r relates to the abstract result via assertion A. The execution costs of $m_\dagger$ and the time credits c' required by the post-assertion Γ' are paid for by the specified cost c_a and the time credits c described by the pre-assertion Γ. Thus, the real costs are paid by a combination of the advertised costs in the abstract program and the potential difference of Γ' and Γ. This is the same technique as we seen in the last section (Example 9.1.1).

Using the affine top $\mathbb{T}$, it is possible for the program to throw away portions of the heap. Note that $\mathbb{T}$ for LLVM can only discard time credits. Memory must be explicitly freed by the concrete program $m_\dagger$.

When presenting the synthesis rule for `bind` we will see that the free operation can be added automatically and is transparent to the abstract algorithm.

In contrast to the definition for Imperative-HOL-Time we here do not talk about the semantics of LLVM explicitly, but use the abstraction of the weakest precondition

to express the effect of the implementation $m_\dagger$. In that way, the definition of the synthesis predicate may serve as a blueprint for connecting NREST with other program semantics: it is only defined relative to *wp* and a time credit Separation Logic. If it provides a weakest precondition predicate, and a Separation Logic with an affine top $\mathbb{T}$ and a resource credit assertion, the above synthesis rule can be defined. I already commented that it is only mere technical problems to prevent Imperative-HOL-Time with its original Separation Logic to be pressed into the generic *wp* in Section 5.7.

Note that in the NREST-monad time increases during execution, in the Hoare triple time credits get consumed during execution and thus decreases, and again time increases in the LLVM-Time as well as the Imperative-HOL-Time monad. It is noteworthy that the direction of lapse of time changes twice during the synthesis from NREST to LLVM. On the other hand, time credits do not measure time passed but the ability that time will pass safely in the future.
Time credits constitute *potential* while in the forwards direction of NREST or the program semantics it is rather *cost*.

9.2.3 The Synthesis Mechanism

In the previous section we have already seen how Sepref symbolically executes the abstract program while synthesizing a concrete implementation. For that process we need *synthesis rules* for all abstract combinators. The combinator `bind` is processed by the following rule:

```
1   ⟦ hnr Γ m_↑ Γ' A_x m;
2     (∀x x_↑. hnr (A_x x x_↑ ⋆ Γ') (f_↑ x_↑) (A'_x x x_↑ ⋆ Γ'') A_y (f x));
3     destructor A'_x free ⟧
4       ⟹ hnr Γ (x_↑ ← m_↑; r_↑ ← f_↑ x_↑; free x_↑; return r_↑) Γ'' A_y (x ← m; f x)
```

The rule works similarly as the rule for Imperative-HOL-Time (cf. Example 9.1.5), with one modification. As our LLVM semantics only allows to discard time credits, but no data structures from the heap, we need to explicitly deallocate the intermediate variable x when it goes out of scope. The predicate *destructor* A'_x *free* (line 3) states that *free* is a deallocator for data structures implemented by refinement assertion A'_x (cf. Section 6.2.5). The program *free* is now inserted into the implementation after calculating $f_\dagger$ and before returning the result. Note that *free* can only use time credits that are stored in A'_x. Typically, these are paid for during creation of the data structure. We require each data structure to contain time credits for freeing it (cf. Section 6.2.5), such that those can simply be consumed at this point. This way amortization can be used effectively to hide the necessary *free* operation and its costs in the abstract program.

All other combinators (`rec`$_c$, `if`$_c$, `while`$_c$, etc.) have similar rules that are used to decompose an abstract program into parts, synthesize corresponding concrete parts recursively and combine them afterwards with the respective combinators from LLVM.

9.2.4 Attain Supremum

Let us comment on a problem that arises when composing *hnr* predicates and data refinement in the NREST monad. Consider the following programs and relations:

$m' = \texttt{res}\ [x \mapsto \$_a,\ y \mapsto \$_b]$ $\qquad$ $\mathrm{R} = \{(z,\ a),\ (z,\ b)\}$
$m = \texttt{res}\ [z \mapsto \$_a + \$_b]$ $\qquad$ $\mathrm{A} = \mathrm{id}_{assn}$
$m_\dagger = \texttt{consume}\ (\$_a + \$_b);\ \texttt{return}\ z$

Data refinement defines the resource bound for a concrete result (here z) as the supremum over all bounds of related results (here x, y). Thus, we have $m \leq \Downarrow_C R\ m'$. Moreover, we trivially have $hnr\ \square\ m_\dagger\ \square\ A\ m$. Intuitively, we want to compose these two refinements, to obtain $hnr\ \square\ m_\dagger\ \square\ (A \circ_{AR} R)\ m'$. However, as our definition of *hnr* does not form a supremum, this would require $\$_a + \$_b \leq \$_a$ or $\$_a + \$_b \leq \$_b$, which obviously does not hold.

We have not yet found a way to define *hnr* or $\Downarrow_D$ in a form that does not exhibit this effect. Instead, we explicitly require that the supremum of the data refinement has a witness. The predicate *attains_sup* $m\ m'\ R$ characterizes that situation: it holds, if for all results r of m the supremum of the set of all abstractions $(r,\ r') \in R$ applied to m' is in that set. This trivially holds if R is *single-valued*, i.e. any concrete value is related with at most one abstract value, or if m' is *one-time*, i. e. assigns the same resource bound to all its results.

In our example, both do not holds. The refinement relation R is not single-valued — z is abstracted by both a and b — and the abstract program m' is not *one−time* — the results x and y are assigned different costs.

In practice we *do* encounter non-single-valued relations[2], but they only occur as intermediate results where the composition with an *hnr* predicate is not necessary. Also, collapsing synthesis predicates and refinements in the NREST-monad typically is performed for the final algorithm whose running time does not depend on the result, thus is *one-time*, and ultimately *attains_sup*.

Why did that not occur already in the synthesis predicate for Imperative-HOL-Time? We did not have the same problem earlier, because the resource usage of Imperative-HOL-Time is always a finite number. In particular, it cannot be infinite nor the supremum of two incomparable elements (there are no currencies!). Thus, if the presumed *hnr* rule holds, for the result r there exists a witness result w of the abstract operation m that has enough advertised cost to cover the expenses of the implementation. Now from the refinement theorem $m \leq \Downarrow_D R\ m'$ we know that the supremum over the costs of all results w' that are refined by w is larger than the assigned costs for w and in turn larger than the costs of the implementation. Even if the supremum in m' is infinite while all costs incurred by the witnesses w' are finite, we always find a

[2]The relation *oarr*, described in earlier work [80, p. 4.2] by one of the authors, is used to model ownership of parts of a list on an abstract level and is an example for a relation that is not single-valued.

witness w' in m' that is larger than the real costs.

It is remarkable that naively choosing the time credits of Imperative-HOL-Time to be natural numbers instead of *enat* lead to this simplification. Both modifications — changing the type of time credits from *nat* to *enat* and adding resource currencies — make the *attains−sup* problem apparent.

9.2.5 Extracting Hoare Triples

Similar to the synthesis predicate in the last section we can again extract Hoare triples for synthesis rules for abstract algorithms whose resource consumption does not depend on the result but only on the input parameters:

$$\begin{aligned} &\mathit{hnr}\ \Gamma\ m_\dagger\ \Gamma'\ R\ (\texttt{spec}\ \Phi\ (\lambda\_.\ T)) \\ &\quad = \{\$T \star \Gamma\}\ m_\dagger\ \{\lambda r.\ \Gamma' \star (\exists_A r_a.\ R\ r_a\ r \star \uparrow(\Phi\ r_a))\} \end{aligned}$$

Again note that the above rule is an equivalence. Thus, it can also be used to obtain synthesis rules from Hoare triples that are provided by the basic VCG infrastructure.

9.2.6 Basic Data Structures

At the leaves of decomposition performed by the synthesis process, atomic operations need to be provided with suitable synthesis predicates.

An example is a list lookup that is implemented by an array:

$$\begin{aligned} &(\mathit{array\_nth},\ \mathit{list\_get}_{spec}\ (\lambda\_.\ \mathit{array\_get}_{cost})) \\ &\quad \in \mathit{array}^k_{assn} \times_a \mathit{snat}^k_{assn} \to_a \mathit{id}_{assn} \end{aligned}$$

Here, the refinement assertions array_{assn}, snat_{assn} and id_{assn} relate a list with an array, an unbounded natural number with a bounded signed word and identical elements respectively.

With an array at address p holding the list xs and an index $i_\dagger$ that is a bounded signed word representing an unbounded natural number i, *array_nth* p $i_\dagger$ leaves the parameters unchanged and extracts the element specified by $\mathit{list\_get}_{spec}$ incurring costs $\mathit{array\_get}_{cost} = \$_{ofs\_ptr} + \$_{load}$.

Ideally, each operation has its own currency (e.g. *list_get*). However, as our definition of *hnr* does not include currency refinement, the basic operations must use the currencies of the LLVM cost model.

To still obtain modular *hnr* rules, we encapsulate specifications for data structures with their cost, e.g. by defining $\mathit{array\_get}_{spec} = \mathit{list\_get}_{spec}\ (\lambda\_.\ \mathit{array\_get}_{cost})$. These can easily be introduced in an additional refinement step. Automating this process, and possibly integrating currency refinement into *hnr* is left to future work.

It is left to comment that for LLVM-Time we only provided very basic data structures: arrays and option arrays. We focused on realizing a case study with a complicated algorithm that involves some complicated running time arguments but only uses simple data structures. For future work, we plan to port many data structures from Imperative-HOL-Time and the Imperative Collections Framework [84, Table 1].

The verification of some of them depends on reasoning about pointer manipulations (e. g. circular linked lists), while other involved data structures are based on basic data structures (e. g. dynamic arrays depend on arrays). For the latter it is possible to model them in an abstract NREST setting, and synthesize implementations using the more basic data structures (following the approach in [82]). For dynamic arrays we describe how amortization can be modeled on the abstract level in Section 10.2. After proving the data structures correct we can wrap up their operations into synthesis rules, to be used as basic operations in more complicated algorithms.

9.3 Summary

- In this chapter we have seen how adaptations of the Sepref tool can be used to synthesize implementations from abstract algorithms in the NREST monad.
- Time credits can be used to seamlessly integrate amortized concrete data structures, where only the advertised costs are visible to the abstract algorithm. Prepaying the costs of deallocation when generating a data structure allows hiding the freeing from the abstract algorithm, keeping those algorithms clear of too many implementation details.
- On the LLVM level control flow is modeled to incur costs. Unfortunately, this has to be modeled also on the NREST level to allow synthesis.
- When combining a synthesis refinement and an abstract refinement, we have to make sure that a witness in the abstract program exists. I have presented an important effect, called it “attains-sup”, described its effects and a solution to cope with it.

10 Case Studies

We have seen that our framework allows to conveniently model abstract algorithms and reason about their functional correctness and resource consumption. Ultimately the goal is to verify the correctness and running time analysis of competitive and executable programs. In this chapter I describe the verification of two larger case studies that I believe would not be feasible to verify using only the verification frameworks described in Part II. Furthermore, I show that amortized analysis can be conducted at the NREST level.

First, I present the verification of Kruskal's minimum spanning tree algorithm. I conducted that formalization in NREST-enat. This yields a verified implementation in Imperative-HOL-Time with time complexity $O(E \log E + M + E\alpha(M))$ using the union-find data structure presented in Chapter 5.

Second, I present the verification of the amortized data structure dynamic array. While I already verified its analysis in Imperative-HOL-Time (Chapter 5) using basic reasoning infrastructure, I now show how amortized data structures can also be modeled in the NREST monad and based on the already provided array formalization can be automatically refined to a concrete implementation in LLVM-Time.

Lastly, I present the verification of the competitive introsort algorithm, a combination of quicksort, insertion sort and heap sort, which has — in contrast to quicksort alone — time complexity $O(n \log n)$. I present the verification of introsort and its components, as well as the refinement to an executable LLVM program that is competitive with real world implementations from standard libraries.

In the end of this chapter I will review related work to the case studies and close this part by examining the framework w. r. t. the features identified in Section 7.1.

10.1 Kruskal

Portions of this section appear in the paper "Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL" (Haslbeck and Lammich [45]).

Functional correctness of Kruskal's minimum spanning tree algorithm was verified in the standard Isabelle Refinement Framework and can be found in the Archive of Formal Proofs [52]. In an earlier publication [45], we additionally have added the verification of the running time analysis. In this section I elaborate more on the structuring of the algorithm verification and focus on the novel running time analysis. I want to illustrate in detail what mechanisms can be used to decompose the verification into parts and thus make it manageable.

10.1.1 Overview

A minimum spanning tree is a concept from graph theory that is a special instance of a general concept in matroids called *minimum weight basis*. Our proof development is structured as follows: first we define the abstract algorithm for minimum weight basis in matroids (c.f. Figure 8.3b) and verify it. Then we instantiate it with the cycle matroid for forests in undirected graphs and refine the algorithm with the usage of equivalence classes. Figure 8.3a shows the last-but-one stage in the stepwise refinement process. In a last step we fix the vertices to be natural numbers and the domain of the disjoint-set data structure to be the set $\{0,\ldots,M\}$, with M being the maximal vertex in the graph. After that, we use the implementation of the union-find data structure from the TIICF to synthesize a concrete algorithm with the Sepref tool.

Provided with a procedure that obtains a list of edges of a graph in linear time, a $O(n \log n)$ sorting algorithm, and an efficient union-find data structure we obtain a concrete algorithm that calculates the minimum weight spanning forest for the graph in time $O(E \log E + M + E\alpha(M))$, with E being the number of edges and M being the maximal vertex in the graph.

In a first iteration, only the logarithmic bounds for the union-find data structure were proved for this case-study. Charguéraud and Pottier [21] verified a union-find data structure with amortized running time $O(\alpha(M))$ (where M is the size of the domain of the disjoint-set data structure and α is the inverse Ackermann function) in Coq. Löwenberg [98] ported that formalization to Isabelle/HOL using the framework from Chapter 5, and it replaced the less efficient implementation.

I consider it worthwhile to illustrate here in more detail how the verification is structured. We start from the abstract theory (matroids), form the greedy algorithm, and prove it correct abstractly. Then, we instantiate it with an instance of the *Cycle Matroid* structure and refine an algorithm to an executable Imperative-HOL-Time program.

This case study illustrates that NREST can serve the two desired purposes. First, it can be used to model abstract algorithms (i. e. the minimum weight basis algorithm) and reason about their correctness and running time complexity. Second, it allows specializing the abstract algorithm and refining it to an executable and efficient implementation that preserves the guarantees for correctness and running time bounds.

Before we delve into the algorithm verification, let us recall some basic theory about matroids.

10.1.2 Weighted Matroids and Minimum Weight Basis

A *matroid* $(E, \mathcal{I})$ consists of a finite set E (called the *carrier set*) and a family $\mathcal{I}$ of subsets of E (called the *independent sets*) and fulfills the following three properties: first, the empty set is independent; second, every subset of an independent set is independent; and finally, the *augmentation property*. It states that if A and B are two independent sets and A has more elements than B, then there exists some element $x \in A - B$ such that $B \cup \{x\}$ is independent.

A subset A of E that is not independent (i. e. $A \notin \mathcal{I}$) is called *dependent*. If a subset A of E is independent and adding any element $x \in E - A$ would make it dependent, it is called a *basis*. That is, a basis is an inclusion-maximal independent set.

If we extend the matroid with a weight function w that maps any element of the carrier set to an element of a linearly ordered commutative monoid (e. g. integers) we obtain a *weighted matroid*. The weight of a subset A of E is defined to be the sum of the weights of the elements in the subset. A *minimum weight basis* is a basis with minimum weight.

Example 10.1.1. The *cycle matroid* is a well-known instance of the abstract concept, and also the one we use in this case study. For a graph $G = (V, E)$, the carrier set is the set of edges and all sets of edges $A \subseteq E$ that do not contain a cycle are deemed independent. Consequently, dependent sets of edges are those that *do* contain a cycle, and a basis in the cycle matroid is a *spanning forest*.

Suppose we have an additional function w that assigns a weight to each edge in the graph. Then, a minimum weight basis corresponds to a *minimum spanning forest* in the graph G, i. e. a minimum-weight inclusion-maximal subset of the edges that does not contain a cycle. If the graph G is connected, that forest is actually a tree.

We use the formalization of matroids by Keinholz [68].

10.1.3 Greedy Algorithm

The task of finding a minimum weight basis in a weighted matroid can be solved by a greedy algorithm: starting from the empty set the algorithm grows the basis by repeatedly adding one element at a time, at each step selecting the minimum-weight element which does not make the set dependent if it was added. The only component of the algorithm that has to do with the matroid is an *independence oracle*: i. e. a subroutine that checks whether adding an element to a set keeps the set independent.

The algorithm is depicted in Figure 8.3b. It lives in the locale *minimum-weight-basis*, which fixes the carrier set c, its subset of possible elements E, the weight function w and the time bounds I will mention in the following. In line 2 the algorithms calculates a list that contains each element of the carrier set once and whose elements are sorted in ascending order by their weight. The running time t_{sc} is reserved for that first operation. In line 5 the accumulator set is initialized as the empty set at cost t_{eb}. Then the algorithm iterates over each element e of the sorted list, checks whether adding e would keep the set independent (line 9), and adds it in case it stays independent (line 12). Those operations have cost t_{it} and t_i respectively. After iterating over all elements, the accumulator is returned as the resulting minimum weight basis.

The running time of the algorithm consists of the time to obtain the sorted list and initialize the accumulator with the empty set, $|E|$ times the cost for executing the independence oracle, and the cost for adding the element to the accumulator.

The correctness lemma for the algorithm *mwb_greedy* was already mentioned in Example 8.3.3. We repeat it here:

$$mwb\_greedy \leq \texttt{spec}\ min\_weight\_basis\ (\lambda\_.\ t_{sc} + t_{eb} + |E| * (t_{it} + t_i))$$

It can be routinely proved with the automation *refine_vcg* for specification refinement. The predicate *min_weight_basis* characterizes a subset $F \subseteq E$ that is a minimum weight basis of for the set of elements E fixed in the locale *minimum-weight-basis*.

While this algorithm works for any matroid, in our case study we instantiate it with the cycle matroid and refine it down to executable code.

10.1.4 Refinement to Graphs

For instantiating the matroid structure an abstract undirected graph G is assumed. It is represented by a set of edges E, a weight function w, and the reachability relation on vertices (*connected F*). That relation is is an equivalence relation for each subset $F \subseteq E$. The predicate *forest* characterizes sets of edges $F \subseteq E$ that do not contain a cycle. Together with some properties on the graph and predicates, we assume one crucial property to hold:

$$[\![\ \textit{forest } F;\ (u,\ v) \in E - F\ ]\!]\ \textit{forest } (F \cup \{(u,\ v)\}) \Longleftrightarrow (u,\ v) \notin \textit{connected } F$$

It expresses that the query to the independence oracle for adding the edge $(u,\ v)$ can be decided by checking whether u and v are connected in the graph induced by the set of edges F. As connectivity is an equivalence relation, it will later be represented by a disjoint-set forest and implemented with the union-find data structure.

Using those concepts, the locale *minimum-weight-basis* can be instantiated. The derived predicate *min_weight_basis* characterizes minimum weight spanning forests in the graph G.

The context that assumes the graph is called *minimum-spanning-tree*. It additionally fixes certain time bounds for the basic operations needed for the algorithm that refines the minimum weight basis algorithm. With those time bounds the locale *minimum-weight-basis* can be instantiated by proving the matroid properties for the cycle matroid and by replacing its parameters with combinations of time bounds fixed in the locale *minimum-spanning-tree*. The expression t_{sc} for obtaining the sorted list of the carrier set is replaced by the sum of $\textit{get\_edges}_{time}$ and $\textit{sort}_{time}$, which is the time allotted for getting a distinct list of the edges of the graph, and sorting it. The expression t_{eb} for obtaining an empty basis is replaced by the sum of $\textit{empty\_forest}_{time}$ and $\textit{empty\_djs}_{time}$, which stand for the time used to generate the data structure of an empty forest and a disjoint-set forest where each element is in its own equivalence class. The running time reserved for the call of the independence oracle ($\textit{indep\_test}_{time}$) replaces t_{it}. For adding an element to the accumulator the abstract algorithm reserves t_i time, this is replaced by the sum of $\textit{insert\_forest}_{time}$ and $\textit{insert\_djs}_{time}$, the time to insert a new edge into a forest and for linking two disjoint sets in the disjoint-set forest.

The total running time of *kruskal* then reads the following:

$$\begin{aligned}\textit{kruskal}_{time} = {} & \textit{get\_edges}_{time} + \textit{sort}_{time} \\ & + \textit{empty\_djs}_{time} + \textit{empty\_forest}_{time} \\ & + |E| * (\textit{indep\_test}_{time} + \textit{insert\_forest}_{time} + \textit{insert\_djs}_{time})\end{aligned}$$

Remember, that this case study is conducted in NREST-enat and Imperative-HOL-Time. Thus the running time bound is just a number.

In the context *minimum-spanning-tree* the algorithm *mwb_greedy* is refined to *kruskal* (cf. the two algorithms in Figure 8.3). There are three main ingredients for that refinement: first, the accumulator set is refined with a list. Second, a disjoint-set forest is maintained that represents the connectivity in the graph induced by the accumulator through the loop. Finally, the independence oracle is implemented by a lookup in the disjoint-set forest. In Example 8.3.5 we described technical details which infrastructure is used to prove the refinement. We obtain the following refinement relation:

$$kruskal \leq \Downarrow_D \ R_{list}^{graph} \ mwb\_greedy$$

Here, R_{list}^{graph} relates a set of abstract edges in the undirected graph with a list of edge tuples representing them. Example 8.3.5 describes how this refinement can be automatically proven with the tactic *refine_rcg*.

10.1.5 Refinement to Executable

At this point the weighted graph is represented by a weight function and a set of edges, which are represented by abstract unordered pairs of vertices. Furthermore, some subroutines are only specified but we did not yet provide implementations for them.

We now fix the representation of an edge to a tuple $(x,\ w_{xy},\ y)$ of two natural numbers x and y representing its end nodes and an integer w_{xy} representing its weight. A weighted graph is then represented by a distinct list of such tuples. We assume that for each edge there is only one element in that list, and the weight is identical to the weight assigned by the weight function w.

The implementation of the disjoint-set forest will later use maps from the vertices of the graph to information needed in the data structure. Those maps will be implemented by arrays with indices from 0 to the maximum vertex number M occurring in the graph. Thus, we refine the algorithm *kruskal* one step further to $kruskal_2$, where the sorting algorithm also calculates the maximum node number and the initialization of the disjoint set data structure uses that number. The algorithm $kruskal_2$ (Figure 10.1) again lives in a locale ($minimum\text{-}spanning\text{-}tree_2$) that fixes time bounds, and instantiates the locale *minimum-spanning-tree* that contains the abstract algorithm *kruskal*. Note that at this point we have to add implementation details to the abstract algorithm in order to allow the implementation later. For the monadic programs further up the refinement chain, we could leave out those details effectively.

Consider the algorithm $kruskal_2$ in Figure 10.1. Here, the domain of the partial equivalence relation *per* is denoted by *Dom per*. Observe that in this refinement step, the **if**-branches are swapped. That is because $per\_compare_{spec}$ *per a b* returns *True* if the elements a and b are in the same equivalence class, i. e. when the current edge should *not* be added to the forest. Note that this breaks lockstep refinement. While it is straightforward to conduct the proof interactively it is not so clear how to automate steps like this. Furthermore, we added some more or less obvious assertions that express facts about the size of data structures. By making them visible as assertions, the Sepref

```
1   kruskal_2 = do {
2     l ← get_edges;
3     assert (|l| = |E|);
4     (sl, mn) ← sort_edges_spec (λ_. sort_edges_time) l;
5     assert (mn = Max V);
6     per_0 ← per_init_spec (λ_. empty_djs_time) (mn + 1);
7     assert ( |Dom per_0| = mn + 1);
8     F_0 ← empty_list_spec empty_forest_time;
9     s ← return (per_0, F_0);
10    (per, spanning_forest) ← nfold sl
11      (λ(a, w, b) (per, F). do {
12        assert (a ∈ Dom per ∧ b ∈ Dom per
13                  ∧ |Dom per| = Max V + 1);
14        i ← per_compare_spec (λ_. indep_test_time) per a b;
15        if i then
16          return (per, F)
17        else do {
18          assert ((a, w, b) ∉ set F);
19          per' ← per_union_spec (λ_. insert_djs_time) per a b;
20          F' ← push_list_spec (λ_. insert_time) (a, w, b) F;
21          return (per', F')
22        }
23      }) s;
24    return spanning_forest
25  }
```

Figure 10.1: The second refinement of Kruskal's algorithm using a disjoint-set forest.

tool can plainly use them to prove the timing side conditions of the synthesis rules for basic operations. We will see an example soon.

Implementation Locales As we might later plug in different implementations for obtaining the distinct list of edges, sorting and disjoint-set forest we use Isabelle's locale mechanism to form components that assume those implementations and can be combined to synthesize an algorithm for Kruskal.

First we come up with a locale for the sorting algorithm. We assume to have an Imperative-HOL-Time program $sort\_edges_{impl}$ and a running time function $sort_{time}$ that maps the length of a list to an upper bound on the running time of that algorithm. Then we assume the following synthesis rule:

$$\begin{aligned}&(sort\_edges_{impl},\ sort\_edges_{spec}\ t)\\ &\quad\in [\lambda xs.\ sort_{time}\ |xs| \le t\ |xs|\ ]\\ &\qquad (\langle edge_{assn}\rangle list_{assn})^k \to (\langle edge_{assn}\rangle list_{assn} \times nat_{assn})\end{aligned}$$

Here, the assertion $\langle A\rangle list_{assn}$ relates a list of elements to another list, where the elements are refined with assertion A. The assertion $edge_{assn}$ relates an edge with a triple representing its end nodes and weight. It essentially is a lifting of the refinement relation R^{edge}_{tuple} to assertions. Note that $\langle id_{assn}\rangle list_{assn} = id_{assn}$. The monadic program $sort\_edges_{spec}\ t\ xs$ specifies a computations that sorts the list xs consisting of edge tuples $(x,\ w_{xy},\ y)$ in ascending order by the second component, calculates the maximum value of the first and third components, and returns both while incurring cost at most t. The implementation $sort\_edges_{impl}$ works on lists, and the rule is able to refine the specification if it reserves enough time: the allotted time $t\ |xs|$ needs to be at least the time needed by the implementation $sort_{time}\ |xs|$.

Second, we form a locale for the disjoint-set forest, or *partial equivalence relation.* We come up with specifications for the following operations: $per\_init_{spec}\ n$ to initialize a partial equivalence relation with the numbers 0 to $n - 1$ each in one equivalence class, $per\_compare_{spec}$ for checking whether two elements are in the same equivalence class, and per_union_{spec} for unifying the equivalence classes of two elements. Then, the locale fixes a refinement assertion that relates some a partial equivalence relation on natural numbers with a type *per* representing the partial equivalence relation on Imperative-HOL-Time's heap. For each of the above mentioned operations the locale fixes a program and a running time function, and assumes a synthesis rule. For instance, initialization has the program $per\_init :: nat \to per\ Heap$, the running time bound $per\_init_{time} :: nat \to nat$ and the following synthesis rule:

$$(per\_init_{impl},\ per\_init_{spec}\ t) \in [\lambda l.\ per\_init_{time}\ l \le t\ l]\ (nat\_assn)^k \to is\_per_{assn}$$

This rule ensures that if the specification reserved enough time, $per\_init_{impl}\ n$ refines $per\_init_{spec}\ t\ n$. That is, given a natural number it returns a data structure that refines (via the refinement assertion is_per_{assn}) the partial equivalence relation that contains singleton equivalence classes for the numbers 0 to $n - 1$. It is not an equivalence relation because it is not idempotent for elements larger than $n - 1$.

For merging of two equivalence classes, we assume the program per_union_{impl}, the function per_union_{time} and the following synthesis rule:

$$\begin{aligned}&(per\_union_{impl},\ per\_union_{spec})\\ &\quad\in [\lambda(R',\ a',\ b').\ a' \in Dom\ R' \wedge b' \in Dom\ R'\\ &\qquad\qquad\qquad\wedge\ per\_union_{time}\ |Dom\ R'| \leq t\ R']\\ &\qquad {is\_per_{assn}}^d \to {nat_{assn}}^k \to {nat_{assn}}^k \to is\_per_{assn}\end{aligned}$$

In the precondition we have to ensure that the elements to be joined actually are in the domain of the data structure and that enough time was reserved by the abstract operation.

For the third locale we extend the locale that assumes a graph to additionally assume to have an implementation that computes a distinct list of the edges of the graph in the tuple representation we described in this subsection. In essence, we fix an Imperative-HOL-Time program *get_edges* and the following synthesis rule:

$$\begin{aligned}&(get\_edges\_impl,\ get\_edges\ t)\\ &\quad[\in \lambda\_.\ get\_edges_{time} \leq t\ ()]\\ &\qquad unit\_assn^k \to list\_assn\ (nat\_assn \times int\_assn \times nat\_assn)\end{aligned}$$

Forming those locales allows us to postpone the decision of how to implement certain subprocesses and data structures, but still makes it possible to synthesize an Imperative-HOL-Time program using Sepref. As a last step, when instantiating (*interpreting* in Isabelle-speak) the locales we need to provide Imperative-HOL-Time programs with their respective running time bounds and prove the respective synthesis rules.

Combining Locales Now for synthesizing a program for $kruskal_2$ we combine the three locales from above. We have to additionally add conditions that ensure that the abstract algorithm reserved enough time for the respective operations. Recall, that $empty\_per_{time}$ was the time reserved for initializing the disjoint sets data structure. As this has to be at least as much as the assumed implementation uses, we add the following assumption:

$$per\_init_{time}\ (Max\ V + 1) \leq empty\_djs_{time}$$

where $Max\ V$ describes the maximum vertex number in the graph. As the elements of the union-find data structure is 0-indexed we need to cover $Max\ V + 1$ elements. We need to add such constraints for getting the list of edges, sorting and the other operations of the disjoint sets data structure. This shows that the reserved amounts of time now have to respect lower bounds in order to allow implementations to be generated. Until now those numbers have only been meaningless place holders. When getting nearer to the implementation they have to respect more properties, in particular they have to respect lower bounds.

Synthesis In the context of this locale we can use Sepref to synthesize an algorithm for $kruskal_2$. As the number of edges in the final forest is not known in advance and

elements need to be pushed to the resulting list in line 20 (in Figure 10.1), the list is implemented efficiently by dynamic arrays. Sepref has to check timing side conditions in the synthesis process: e. g. for the initialization of the union-find data structure in line 6 the property $per\_init_{time}\ (mn + 1) \leq (\lambda\_.\ empty\_djs_{time})\ (mn + 1)$ has to be checked. It can be discharged with the assumptions from the locale and the fact from the assertion in line 5. For the operation in line 19 that unifies the equivalence classes of two elements the side condition $per\_union_{time}\ |Dom\ per| \leq (\lambda\_.\ insert\_djs_{time})\ per$ has to be checked. It can be discharged with the respective locale assumption and the fact from the assertion in line 12 and 13. We then obtain the synthesis rule for $kruskal_2$, which is combined with its refinement lemma and the correctness lemma for *kruskal*.

Already at the current stage we can extract a Hoare triple for the synthesized program, which is dependent on the assumed Hoare triples for the subprograms and their running times:

$$<\$\, kruskal_{time}>$$
$$kruskal_{impl}$$
$$<\lambda r.\ (\exists_A ra.\ \langle edge_{assn}\rangle da_{assn}\ ra\ r \star \uparrow(min\_weight\_basis\ ra))>$$

Here, the refinement assertion $\langle A\rangle da_{assn}\ ra\ r$ refines an abstract set *ra* with a dynamic array, where the elements are related by the refinement assertion A.[1] The predicate *min_weight_basis* characterizes subsets $F \subseteq E$ of the edges of the graph G that form a minimum weight spanning forest. Note that the program does not have explicit arguments, as they are captured in the implementation locale. Also $kruskal_{time}$ actually depends on the number of edges $|E|$ in the graph, and the largest vertex M.

Implementing the Locales But remember, that we still assume to have implementations for *get_edges*, sorting and the partial equivalence class. As a final step we have to provide implementations for that. This is achieved by interpreting the locales with concrete implementations.

I illustrate this for with the union-find data structure: in Section 5.5.2 we described how we obtained Hoare triples for implementing the union-find data structure. For the initialization, for example, we get the program uf_init_{impl} and the time function uf_init_{time}. We feed them into the locale and obtain the goal to prove the synthesis rule for uf_init_{impl} and per_init_{spec}. Let the representation predicate for union-find be *is_uf*. We can reduce proving the synthesis rule to proving a Hoare triple using the equivalence mentioned in Section 9.1, which gives us the following proof obligation:

$$<\$\, uf\_init_{time}\ n>\ uf\_init_{impl}\ n\ <\lambda r.\ is\_uf_{assn}\ r\ \{(v,\ v)|\ v \leq n\}\}>$$

This lemma can be proved with what we already established in Section 5.5.2.

Similarly we provide implementations for the other union-find operations and the sorting locale. In the end we have to get an input graph. We assume to get a list of distinct edges L, which represent an abstract graph. Then, the implementation of *get_edges* just emits that list. We can plug that operation into the locale and finally

[1] This assertion is essentially $dyna_{assn}$ from Example 5.5.1 with an extra refinement for the elements.

can use $kruskal_{impl}$ together with the implementations of the subprograms. As the running time bound for the synthesized Kruskal algorithm ($kruskal_{time}$ ($|E|$, M)) only depends on the size of the edge set and the maximum vertex number and otherwise is free from parameters, we can also examine its running time complexity. Using the union-find data structure mentioned in Section 5.5.2 we get:

$$kruskal_{time} \in \Theta_2(\lambda(E,\ M).\ E \log\ E + M + E\ \alpha(M))$$

Here the first summand covers the sorting of the edge list, and also the linear time for obtaining the edge list, the constant time initialization of the empty forest list and the linear time for pushing of elements to the forest list. The second summand pays the initialization of the union-find data structure, and the last summand covers the union and compare operations that are executed for each edge in the worst case.

Recap Let me recapitulate what we have just seen. Provided with theory about matroids, we verified also the running time analysis of the greedy algorithm on that abstract level. After that we instantiated it with the specific form for the cycle matroid and refined the independence oracle to be implemented via a connectivity check in the graph. That check can be realized by maintaining a disjoints set data structure and executing checks on that. Implementing that data structure by union-find and refining other subprograms accordingly yields an executable implementation. At any stage of that stepwise refinement one could intervene and take another path, e. g. instantiating the greedy algorithm for a different matroid structure, or using another implementation of union-find (as we did) or changing the representation of the independent set that gets grown to a basis.

We use locales and their free variables as place holders for running times that are yet to be determined. When refining some operation we also refine those place holders by substituting them with sums of new and more basic place holders that come from the locale in which we interpret the original locale. In that way, a locale serves as a *currency system environment* that reserves time functions as implicit *currencies*. When interpreting those locales from another foreign currency environment, one has to determine what value those foreign currencies have in terms of the current domestic currencies. In fact it evolves a stacking of several locales and "currency exchanges".

10.1.6 Lessons Learned

Separation of Concerns What we have seen in this case study is an example where we separated the abstract analysis of an algorithm from the implementation details. The combinatorial concept that generalizes minimum spanning trees is a concept in matroids. I think this use case shows that interesting algorithmic challenges of software verification can be separated cleanly and tackled individually, while living in the same environment and being combined to obtain a final correctness lemma (the Hoare Triple) of a synthesized algorithm.

Reuse of formalizations without running time analysis This case study shows that the correctness arguments can be plainly reused from earlier verifications that focus on functional correctness only. Adding the proofs of the running time claims does not interfere with the proofs for functional correctness. They only evoke additional verification conditions and leave the ones concerned with functional correctness unchanged. However, it is necessary to add more assertions in the algorithms that express facts about the sizes of the data structures used. This reasoning is mostly done on the abstract level, but the information has to be passed to the concrete algorithm via assertions. In the Sepref translation phase, this information is needed to discharge the preconditions of the *hnr* predicates, which demand that enough time has been reserved to execute the step.

Improved Union-Find Data Structure For the first iteration we wanted to showcase the mechanism, how to use NREST and the Sepref tool for the running time analysis of larger algorithms. We only proved the logarithmic running time bound and came up with the (ICF-style [89]) locale to later facilitate plugging in other implementations of union-find. Later, Löwenberg ported the Coq formalization of Charguéraud and Pottier [21] that proved the improved $O(\alpha(n))$ bound to Isabelle/HOL. It was easy to use that one instead of the one with the worse running time complexity proof. As mentioned before, he uncovered a running time bug in the original Imperative-HOL-Time implementation that prohibited proving the better bound: it did not compress on union but only on find. The tighter running time bound was proven for the repaired program. This shows that even verifying the functional correctness does not necessarily prevent running time bugs.

10.2 Dynamic Arrays in the Abstract

In this section I will present a case study that shows that amortized data structures can also be proven correct on the abstract NREST level.

I have already presented the verification of dynamic arrays for Imperative-HOL-Time in Example 5.5.1. There the bottom-up approach was used: first the amortized data structure dynamic array was directly implemented in the target language and its correctness and amortized running time was verified with the automation of Imperative-HOL-Time. This includes a component that reasons about time credits, which I have described in Section 5. That development is specific to Imperative-HOL-Time and cannot be reused for other program logics. Also, it does not take advantage of the top-down approach of the refinement framework.

For LLVM-Time we did not adapt the special tactics for reasoning about time credits, but only added the simple running time consumption of some basic operations. As described in Chapter 6, a thin layer is added on top of the shallow LLVM semantic and only rules for basic operations are proved. As much as possible of the verification of more involved data structures and algorithms should be moved to the abstract NREST-level and the proof automation on that level should be used. That way, those algorithms

are largely independent from the target language and can be reused for several back ends. In the overview of the IRF (Section 7.2) I have mentioned that while the back ends evolve, the monadic language is more stable. Furthermore, it is beneficial to separate concerns, and prove the correctness of the algorithmic idea separately from the implementation details.

This certainly has limitations: pushing reasoning to the abstract seems not to be possible for more involved data structures that use pointer reasoning, as NREST does not support that. But we still can lift amortized complexity reasoning to the abstract. This case study shows how that works and what its limitations are.

For presentation purposes I omit some size side conditions that are vital for the implementation in LLVM. I will comment on that at the end of this section.

Note that because in this case study we target LLVM-Time we work with NREST-ecost and resource currencies. In fact, the mechanism is general enough that it also works for the NREST-enat monad. For presentation purposes I will present the theory for the instance NREST-ecost.

10.2.1 Specifying the Operation on Lists

The operation to focus on in this section is the amortized constant time *push* operation on dynamic arrays. The corresponding abstract operation is appending an element to the back of a list. We routinely define the specification in NREST:

$$push\_list_{spec}\ t = \texttt{spec}\ [xs \cdot [x] \mapsto t]$$

The resource bound is left as a parameter to be specified later. In the end, we want to obtain a synthesis rule that can directly refine the abstract list operation to a concrete LLVM-Time implementation with a constant running time. In Example 5.5.1 and Example 9.1.1 we have seen the verification in Imperative-HOL-Time and the synthesis predicate that can be obtained from it. This case study takes a different route by verifying the abstract data structure on the NREST-level and then refining it down to LLVM-Time. Regardless, we want to obtain a similar result: a synthesis rule that refines $push\_list_{spec}$ with constant time. Formally the result should look like this:

$$(da\_push_{impl},\ push\_list_{spec}\ arraylist\_push_{cost}) \\ \in (\langle A\rangle da_{assn})^d \to A^k \to \langle A\rangle da_{assn}$$

Here, the refinement assertion $\langle A\rangle da_{assn}$ relates an abstract list to a concrete LLVM representation of a *dynamic array (da)* with the elements of that list being refined with refinement assertion A. Note that this refinement assertion will contain time credits that represent the potential of the data structure. The LLVM program da_push_{impl} refines the push operation on a dynamic array. To the user the internals of the data structure, i. e. the length, the capacity and the implementation of the list, are irrelevant.

10.2.2 Modeling Dynamic Lists

We model dynamic arrays (da) first abstractly by *dynamic lists* (dl). They consist of a carrier list cs and two numbers l and c representing the *length* and the *capacity* of

the dynamic list. A list as is refined by a dynamic list $(cs,\ l,\ c)$, if the first l elements of cs form the list as. Furthermore, in a valid dynamic list the length is at most the capacity, the capacity is the length of the carrier list and the carrier list is non-empty. Formally:

$$((cs,\ l,\ c),\ as) \in R^{list}_{dynlist} \longleftrightarrow take\ l\ cs = as \wedge l \leq c \wedge c = |cs| \wedge |cs| > 0$$

Using this representation, we can now specify a push operation on dynamic lists. A push of an element x to a dynamic list $(cs,\ l,\ c)$ will result in a valid dynamic list that contains the same elements as before and adds the element x at the end. As the dynamic list may have reached its capacity, it may be necessary to increase the capacity. We can state the intuition in the following NREST-program:

$$\begin{aligned} dl\_push_{spec}\ t\ (cs,\ l,\ c)\ x = \mathtt{spec}\ (\lambda(cs',\ l',\ c').\ & take\ l\ cs' = take\ l\ cs \wedge cs'\ !\ l = x \\ & \wedge\ l' \leq c' \wedge c' = |cs'| \\ & \wedge\ l' = l + 1 \wedge c' \geq c)\ (\lambda r.\ t) \end{aligned}$$

Here, we first only specify the functional correctness, and leave the cost t as a parameter. We already fix that the program has constant cost, independent from the result and the input. The specification requires that the resulting dynamic list contains all the elements as before and adds x at the end. It is not specified whether or how much the carrier list has to increase.

We now can show that the push operation on dynamic lists refines the *push_list* operation on lists:

$$(dl\_push_{spec}\ t,\ push\_list_{spec}\ t) \in R^{list}_{dynlist} \rightarrow Id\ \rightarrow R^{list}_{dynlist}$$

Pushing element x to a dynamic list yields a valid dynamic list that represents the abstract list with element x at the end. Note that, the time specified with t represents the amortized costs of that operation. It is agnostic to the exact implementation of the operation.

Refining the push operation Now let us refine the push operation with the abstract algorithmic idea. If we run out of capacity, we double the size of the carrier list and push the element afterwards. First we specify the doubling and the basic push operation.

The first operation doubles the carrier list while keeping the first l elements of the carrier list the same. The cost is specified to be linear in the capacity c of the original dynamic list. We will have to allocate $2 * c$ space at cost linear in c and have to copy at most c elements from the old to the new carrier list. We use a new currency dl_double_c to capture the constants that will arise in implementations. Formally we specify:

$$\begin{aligned} & dl\_double_{spec}\ (cs,\ l,\ c) = \mathtt{do}\ \{ \\ & \quad \mathtt{assert}\ (l \leq c \wedge c = |cs|); \\ & \quad \mathtt{spec}\ (\lambda(cs',\ l',\ c').\ take\ l\ cs' = take\ l\ cs \wedge |cs'| = 2 * |cs| \\ & \qquad\qquad \wedge\ l' = l \wedge l \leq c' \wedge c' = |cs'|) \\ & \qquad (\lambda\_.\ \$_{dl\_double_c}\ c) \\ & \} \end{aligned}$$

Note that, while we will use the *double* operation only if the capacity is reached ($l = c$), we do not assert this in the specification. The specification works also in the case when there is still space left. Then the cost for copying the elements will not be tight, but over-approximated.

The second operation is to push an element to a dynamic list, given we already know that there is enough space. We write the element x to position l of the carrier list and increase the length l. This operation takes cost for the addition and the write into the list.

$$
\begin{array}{l}
dl\_push\_basic_{spec}\ (cs,l,c)\ x = \mathbf{do}\ \{ \\
\quad \mathbf{assert}\ (l < |cs|); \\
\quad cs' \leftarrow list\_set_{spec}\ (\lambda\_.\ \$_{list\_set})\ cs\ l\ x; \\
\quad l' \leftarrow \mathbf{return}\ (l +_{\$_{add}}\ 1); \\
\quad \mathbf{return}\ (cs',\ l',\ c) \\
\}
\end{array}
$$

Here, the program $\mathbf{return}\ (a +_T b)$ returns the result $a + b$ incurring cost T.

At this point we still leave open how we will implement the carrier list. As long as the *set* operation will have constant time, we will be able to prove the dynamic list correct. Later we will use LLVM arrays to implement those lists.

Now let us formulate the algorithmic idea of the push operation for dynamic lists: if there is space left, we simply use $dl\_push\_basic_{spec}$ to push the element into the dynamic list. If the capacity is reached, we first double the carrier list size with dl_double_{spec} and then push the element.

$$
\begin{array}{l}
dl\_push\ (cs,\ l,\ c)\ x = \mathbf{do}\ \{ \\
\quad \mathbf{assert}\ (l \leq c \wedge c = |cs| \wedge 0 < |cs|); \\
\quad \mathbf{if}_c\ l <_{\$_{less}} c\ \mathbf{then\ do}\ \{ \\
\quad\quad dl\_push\_basic_{spec}\ (cs,\ l,\ c)\ x \\
\quad \mathbf{else\ do}\ \{ \\
\quad\quad (cs',\ l',\ c') \leftarrow dl\_double_{spec}\ (cs,l,c); \\
\quad\quad \mathbf{assert}\ (l' = l \wedge l < c' \wedge c' = |cs'| \wedge take\ l\ cs = take\ l\ cs'); \\
\quad\quad dl\_push\_basic_{spec}\ (cs',\ l',\ c')\ x \\
\quad \} \\
\}
\end{array}
$$

While we later call *dl_push* only from valid dynamic lists, we add seemingly trivial assertions to facilitate further refinement. When refining *dl_push* we can simply use the facts from those assertions, and do not need to restrict in what context the operation is used.

There is a trade-off between making abstract algorithms verbose by adding too many assertions and additional proof burden by using too few assertions. During development of a refinement chain one often tries to use as few as possible but as many as necessary to allow to automatically solve the side conditions emitted by the VCG during a refinement proof. Usually, to facilitate the automatic proof of a refinement on one level, one adds assertions that express facts about the abstract concepts to the program one level above.

But those assertions then have to be proved in the refinement proof of the abstract program. Often that are simple facts on that level and they are proved automatically. Using assertions allows to transport knowledge down the refinement chain.

Note that the cost of the push operation incurs the *raw*, i. e. non-amortized, costs of the operation. Let us examine what costs the two cases incur. If there is capacity left, we have to pay for the *if*-branch and its guard as well as the addition and the list update for the push. This can be summarized in the constant costs *push* incurs: $push\_overhead_{cost} = \$_{less} + \$_{if} + \$_{add} + \$_{list\_set}$. In the other case, we have to additionally pay for the doubling: $push\_overhead_{cost} + \$_{dl\_double_c}\ c$. Thus, the worst-case cost of the operation is not constant, but rather linear in c because of the *double* operations.

As a next step we will see how we can model the potential method on the NREST level and prove that the abstract push operation has amortized constant time.

10.2.3 Amortized Analysis

The potential method for amortized complexity has the following well-known inequality that relates the raw cost of an operation with its advertised cost and the potential of the data structure before and after an operation.

$$raw\_cost_i \leq (\Phi_i + advertised\_cost_i) - \Phi_{i+1}$$

Before executing an operation we can get the *cost credits* from the potential of the data structure and add it to the cost that is advertised to the caller of the operation. Then we execute the operation incurring the raw costs, and afterwards we need to give back the *cost credits* for the potential of the resulting data structure. Then we can execute several operations on the data structure one after the other and use telescoping to obtain the following inequality

$$\begin{aligned}\sum_{0\leq i<n} raw\_cost_i &\leq \sum_{0\leq i<n} (\Phi_i + advertised\_cost_i) - \Phi_{i+1}\\ &= (\Phi_0 + \sum_{0\leq i<n} advertised\_cost_i) - \Phi_n\\ &\leq \sum_{0\leq i<n} advertised\_cost_i\end{aligned}$$

Where we assume that each raw_cost_i and Φ_i is non-negative. For the last inequality we assume that the potential is *0* in the beginning, i. e. $\Phi_0 = 0$. The inequality expresses that the real costs can be upper bounded by the sum of the advertised costs.

To model the subtraction in the amortization inequality we cannot simply use `elapse` as we do not necessarily allow for negative costs. Instead, we introduce a new combinator `reclaim` and formulate the amortization inequality in the NREST-monad in the following way:

$$m_{raw}\ ds \leq \texttt{reclaim}\ (\texttt{elapse}\ (m_{adv}\ ds)\ (\Phi\ ds))\ (\lambda ds'.\ \Phi\ ds')$$

Here the raw monadic program m_{raw} executed on some data structure has to refine the program that first consumes the potential of the data structure, then executes the monadic program with advertised costs, and in the end reclaims as much costs as the resulting data structure needs for its potential.

The combinator *reclaim* subtracts cost from a monadic program, and fails if it would get negative. Note that this approach only works if the resource type provides a minus operator, as *ecost* does in our case.[2] Here is the formal definition:

$$\begin{aligned}
&\mathit{reclaim} :: (\alpha,\ \mathit{ecost})\ \mathit{NREST} \to (\alpha \Rightarrow \mathit{ecost}) \to (\alpha,\ \mathit{ecost})\ \mathit{NREST}\\
&\mathit{reclaim}\ \mathit{fail}\ t = \mathit{fail}\\
&\mathit{reclaim}\ (\mathit{res}\ M)\ t = \mathit{Sup}\ \{\ \mathit{if}\ t\ x \le t'\ \mathit{then}\ \mathit{res}\ [x \mapsto t' - t\ x]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{fail} \mid t'\ x.\ M\ x = \mathit{Some}\ t'\ \}
\end{aligned}$$

For each possible result x of M the combinator checks whether the consumed time t' is at least the reclaimed time $t\ x$ for that result. This ensures not falling into the negative when subtracting. If one of the inequalities does not hold, the whole program $\mathit{reclaim}\ m\ t$ fails.

Using *reclaim* we can state the theorem that *dl_push* refines $\mathit{dl\_push}_{spec}$ using the potential method. Here is the *amortization refinement lemma*:

$$\begin{aligned}
&\mathit{dl_push}\ (cs,\ l,\ c)\ x\\
&\quad \le \mathit{reclaim}\ (\mathit{elapse}\ (\mathit{dl_push}_{spec}\ \mathit{push_adv}_{cost}\ (cs,\ l,\ c)\ x)\ (\Phi_{dl}\ (cs,\ l,\ c)))\ \Phi_{dl}
\end{aligned}$$

We did not yet specify what $\mathit{push\_adv}_{cost}$ and Φ_{dl} are, and we can leave them as free variables to explore what properties they need to fulfill. The generated verification conditions regarding functional correctness can be discharged automatically and we are left with those that are concerned with the cost analysis:

$$\begin{aligned}
&\mathit{push_overhead}_{cost} \le \mathit{push_adv}_{cost} + \Phi_{dl}\ (cs,\ l,\ c) - \Phi_{dl}\ (cs',\ l+1,\ |cs|)\\
&\mathit{push_overhead}_{cost} + \$_{\mathit{dyn_list_double}_c}\ (|cs|)\\
&\quad \le \mathit{push_adv}_{cost} + \Phi_{dl}\ (cs,\ l,\ |cs|) - \Phi_{dl}\ (cs',\ l+1,\ 2 * |cs|)
\end{aligned}$$

For readability I have left out the preconditions of the goals. The first one originates from the first branch where there is enough capacity: the length increases and the capacity stays the same. The second one stems from the second branch where we need to double the carrier list: here the length increases and the capacity doubles.

First consider the latter condition: the raw costs contain the overhead to push the element plus the cost to double the carrier list, which is linear in the capacity ($c = |cs|$). We can pay for that with the advertised cost and the difference in potential. One way of achieving that is that the potential Φ_{dl} is *0* after the doubling, i. e. when the length is at most half the capacity, and contains the necessary resource coins when the capacity is reached ($l = c$). The potential function we already have used in Example 5.5.1 works here again: $\Phi_{dl}\ (cs,\ l,\ c) = \$_{\mathit{dl\_double}_c}\ (2 * l - c)$. For $l = c$ we obtain c coins, and for $2 * l \le c$ we obtain *0* coins as we cut off negative values. Note that we use the currency $\$_{\mathit{dl\_double}_c}$ to mean per-element costs instead of per-operation costs. That is, one coin stands for how much a double operation costs per element.

Consider now the first inequality with the potential that we have just chosen. If we push an element, the length increases and thus the potential increases by two $\mathit{dl\_double}_c$

[2] Alternatively one could pull the reclaim to the left of the inequality and model it with a consume. But we would like to further refine raw programs and use transitivity of the refinement relation to easily compose refinements. That is why we want to have only the raw program on the left of the refinement statement.

coins. So additionally to the overhead costs of *push* we need to spend two additional coins to account for the potential difference and thus obtain a feasible value for the advertised cost: $push\_adv_{cost} = push\_overhead_{cost} + \$_{dl\_double_c}\ 2$. In particular, we can see that the advertised cost is constant.

Choosing the potential and the advertised cost in that way lets us prove the correctness lemma between *dl_push* and dl_push_{spec} involving the potentials. To prove the lemma, one can first pull the `consume` and `elapse` into the specification. This involves proving that the sum of pre potential ($\Phi\ ds$) and the advertised cost (t_a) is at least the post potential ($\Phi\ ds'$). Then, the goal has the standard form for specification refinement and can be routinely solved by *refine_vcg*. It essentially proves that program *dl_push* has to fulfill the functional specification and take less resources than the remaining difference ($(\Phi\ ds + t_a) - \Phi\ ds'$).

This concludes the verification on the NREST-level. We have shown that we can use the potential Φ_{dl} to prove *dl_push* having amortized constant time. We can go on proving correct other operations on the data structure with amortization, e. g. lookup, write within bounds, and also initialization. That includes to show that they respect the change of potential. We can also apply telescoping on this level and sequentially compose several `reclaim`–`elapse` pairs on the same data structure following the intuition above.

It is left to show that we can actually implement the operation with a concrete program and obtain the desired synthesis rule mentioned at the beginning of this section.

10.2.4 Moving Potential to Time Credits

Now we have obtained a refinement in the `reclaim`–`elapse` pattern. In order to obtain the desired synthesis rule, we need to move the potential from the abstract NREST-program into the pre- and post-heap in the synthesis rule. This only leaves the advertised cost in the abstract program.

We now present two rules that allow to move credits in a synthesis rule between the heap and the NREST program. First, we give a rule that moves the post-potential that is captured in a `reclaim` combinator to the post-heap. More specifically we will integrate the potential using time credits into the refinement assertion relating the abstract and concrete result of the computation. Second, we prove a rule that moves the pre-potential that is captured in an `elapse` combinator to the pre-heap of the synthesis rule.

Intuitively, the resource usage of the abstract program m in a synthesis rule is the specified advertised time for the concrete implementation. If we look closer into the definition of the synthesis predicate (cf. Section 9.1.1 or rather Section 9.2.2) we see that this abstract cost c_a pays together with the resource credits c in the pre-heap for the cost (say t) of the program $m_\dagger$ and the credits c' that need to be stored on the heap to make the postcondition true. Additionally, we can discard surplus resource credits with the garbage collection assertion $\mathbb{T}$. Thus, morally we have $c + c_a \geq t + c'$.

I now present the two rules for moving credits from `reclaim` and `elapse` into the synthesis predicate.

On the one hand, the meaning of a non-failing program $m = \texttt{reclaim}\ m'\ T$ is that each result of m' incurs more than T cost and that this cost can just be reclaimed and the program m then reserves less time. More formally, it expresses that each result x of m' incurs more time than $T\ x$ and m has the same results as m' with $T\ x$ subtracted from each of the costs. Let c'_a be the cost of m' for the abstract result r_a, with $c'_a \geq T\ r_a$. Consequently, we have $c_a = c'_a - T\ r_a$, which we can plug into the inequality from above and obtain: $c + (c'_a - T\ r_a) \geq t + c'$. We can now move that reclaimed time to the right hand side and put it on the post-heap: $c + c'_a \geq t + (c' + T\ r_a)$. We can formulate this intuition as a rule on *hnr*:

$$\begin{array}{l}[\![\ \mathit{nofail}\ m' \implies \mathit{nofail}\ (\texttt{reclaim}\ m'\ \Phi);\\ \quad \mathit{hnr}\ \Gamma\ c\ \Gamma'\ A\ (\texttt{reclaim}\ m'\ \Phi\ ]\!]\\ \implies \mathit{hnr}\ \Gamma\ c\ \Gamma'\ ([\Phi]A)\ m'\end{array}$$

Here, the operator $[\Phi]A\ r\ r_a = \$\Phi\ r_a \star A\ r\ r_a$ adds the potential as time credits depending on the abstract result to an assertion. The first premise of the rule essentially states that Φ can be safely subtracted from m'. Then, if we can prove the synthesis rule for m we can move the reclaimed resources to the refinement assertion for the result and obtain a synthesis rule for m'.

On the other hand, the meaning of $m = \texttt{elapse}\ m'\ T$ is that we spend T more resources than the computation m'. That is for the abstract result r_a we have $c_a = {c_a}' + T$, which we can plug into the equality from above, and get: $c + ({c_a}' + T) \geq t + c'$. We can shift that to the pre-heap and obtain: $(c + T) + {c_a}' \geq t + c'$. We can formulate that as a rule on *hnr*.

$$\mathit{finite\_cost}\ t \implies \mathit{hnr}\ \Gamma\ m_\dagger\ \Gamma'\ R\ (\texttt{elapse}\ m'\ t) \implies \mathit{hnr}\ (\$t \star \Gamma)\ m_\dagger\ \Gamma'\ R\ m'$$

where *finite_cost t* expresses that the resource function t is finite everywhere, i. e. all of the currencies in t have a finite amount. From a synthesis predicate with a consume, we can obtain a synthesis predicate for m' that moved the surplus cost into the pre-heap. Note that we could only prove this rule for finite cost t.

Combining those two rules, we can convert a synthesis rule with a *reclaim–elapse* abstract program into a synthesis rule that moves the potentials into the refinement assertion of a data structure in the pre-heap and the refinement assertion for the result:

$$\begin{array}{l}(m_\dagger,\ \lambda(x,r).\ \texttt{reclaim}\ (\texttt{consume}\ (m\ x\ r)\ (\Phi\ x))\ \Phi) \in G^d \to R^k \to G\\ \implies\ \ (m_\dagger,\ m) \in ([\Phi]A)^d \to R^k \to [\Phi]A\end{array}$$

I call this rule an *amortization synthesis rule*. Note that I have dropped the two side conditions for simplicity.

10.2.5 Obtaining a Synthesis Rule

In order to obtain the desired synthesis rule, we need to provide an implementation and connect it to the program *dl_push*. Observe that *dl_push* lives in the currency system of dynamic lists and not of LLVM currencies. We need to refine it to some abstract program *da_push* that fixes the way we implement the carrier list to arrays and refines

all operations to operations we have synthesis rules for. This involves exchanging the currencies from dynamic lists to LLVM currencies via some exchange rate E_{da}. For presentation purposes we skip the details of that part and assume we come up with a program *da_push* and a suitable refinement $da\_push\ dl\ x \leq \Downarrow_C E_{da}\ (dl\_push\ dl\ x)$.

Furthermore, let $\langle A\rangle da\_raw_{assn}$ be the refinement assertion that relates a concrete representation of a dynamic array (e. g. a triple of an array and two words for the length and capacity) — parameterized with the refinement assertion A for the elements of the array — with a dynamic list. Further, we assume that we have synthesized an LLVM program da_push_{impl} that refines *da_push*, with the following synthesis rule:

$$(da\_push_{impl},\ da\_push) \in (\langle A\rangle da\_raw_{assn})^d \to A^k \to \langle A\rangle da\_raw_{assn}$$

Now we can combine the currency refinement rule for *da_push* and the amortization refinement rule for *dl_push* and obtain to the following refinement:

$$\begin{aligned} &da\_push\ dl\ x\\ &\quad \leq \texttt{reclaim}\ (\texttt{elapse}\ (dl\_push_{spec}\ push\_concrete\_adv_{cost}\ dl\ x)\ (\Phi_{da}\ dl))\ \Phi_{da}\end{aligned}$$

Where the currency refinement was already distributed over **reclaim** and **elapse**, which yields two following two cost functions: $push\_adv'_{cost} = \downarrow_C E_{da}\ push\_adv_{cost}$ and $\Phi_{da}\ dl = \downarrow_C E_{da}\ (\Phi_{dl}\ dl)$.

We can now combine that refinement rule with the synthesis rule from above. Note that the refinement does not involve data refinement, and thus does not have any attains-sup side conditions (cf. Section 9.2.4). We obtain the following synthesis rule:

$$\begin{aligned} &(\lambda(da', x').\ da\_push_{impl}\ da'\ x',\\ &\qquad \lambda(dl, x).\ \texttt{reclaim}\ (\texttt{elapse}\ (dl\_push_{spec}\ push\_adv'_{cost}\ dl\ x)\ (\Phi_{da}\ dl))\ \Phi_{da})\\ &\quad \in (\langle A\rangle da\_raw_{assn})^d \to A^k \to \langle A\rangle da\_raw_{assn}\end{aligned}$$

This form fits the precondition of the amortization synthesis rule, and we can apply it to move the elapsed and reclaimed resources to the pre-heap and the refinement assertion for the result respectively.

$$\begin{aligned} &(\lambda(da', x').\ da\_push_{impl}\ da'\ x',\ \lambda(dl,x)\ dl\_push_{spec}\ push\_adv'_{cost}\ dl\ x)\\ &\quad \in ([\Phi_{da}]\langle A\rangle da\_raw_{assn})^d \to A^k \to [\Phi_{da}]\langle A\rangle da\_raw_{assn}\end{aligned}$$

At this point we already have established a refinement between the push operation on dynamic lists dl_push_{spec} and the implementation on dynamic arrays da_push_{impl}. We could extract a Hoare triple from the synthesis rule that shows the correctness of the implementation and the amortized constant running time.

As a last step, we hide the intermediate concept of dynamic lists and obtain a refinement between the list operation and the implementation on dynamic arrays. First, consider the data refinement between *dl_push* and $push\_list_{spec}$. We repeat it here:

$$(dl\_push_{spec}\ T,\ push\_list_{spec}\ T) \in R^{list}_{dynlist} \to Id\ \to R^{list}_{dynlist}$$

We can apply this data refinement to the synthesis rule above, and use the fact that dl_list_{rel} is single-valued[3] to solve the sup-attains side condition. Then, we obtain the final synthesis rule:

[3]That is, every dynamic list has at most one corresponding abstract list.

$$(da\_push_{impl},\ push\_list_{spec}\ push\_adv'_{cost}) \in (\langle A\rangle da_{assn})^d \to A^k \to \langle A\rangle da_{assn}$$

Where $\langle A\rangle da_{assn}$ relates a list with a dynamic array, where assertion A relates the elements. This refinement assertion combines the refinement relation dl_list_{rel}, the raw refinement assertion $\langle A\rangle da\_raw_{assn}$ and the augmentation with the time credits containing the potential. Formally we define:

$$\langle A\rangle da_{assn}\ as\ al = \exists dl.\ [\Phi_{da}](\langle A\rangle da\_raw_{assn})\ dl\ al \star \uparrow((dl,as) \in dl\_list_{rel})$$

Once we have the last synthesis rule, we can cut out the whole reasoning with the combinators `reclaim` and `elapse` and inspect the rule on its own. For a user of the rule, only the constant advertised cost is visible in $push\_adv'_{cost}$ and the whole amortization is hidden and happens under the hood. The refinement assertion da_{assn} serves as a black box for the user. It is not transparent whether it is an amortized data structure, and can be treated like a normal one.

10.2.6 Recap

Let me sum up how the development process of a data structure with amortized running time was structured. I indicate the terms in this case study that correspond to the general concepts by adding them in brackets.

First, the operation ($push\_list_{spec}\ T$) on the abstract data type (list) is defined with a parameter T as time bound. Next, the abstract representation of the data structure (dynamic list) and a refinement relation with the abstract data type ($R^{list}_{dynlist}$) is defined. Then, the operation on that representation ($dl\_push\_{spec}\ T$) is defined and the refinement of the abstract operation is proved. The programs involved until now have flexible cost functions and represent amortized costs.

Second, the algorithmic idea is formulated by implementing the operation (dl_push) on the data structure. To implement the data structure, typically simpler data structures and operations on them are used (in our example lists). At this point it is still unspecified how those data structures are implemented, but they already have assigned costs in currencies specific to the data structure (e. g. $list\_set$ and dl_double_c). The cost of that program is a raw cost that is non-amortized, and typically differs for different cases, which will be leveled out by amortization. Now, an amortization refinement lemma between the specification (dl_push_{spec}) and the operation (dl_push) is set up. Here, the amortized costs and the potential function have to be given. Interactive proof can help by extracting the necessary conditions, but concrete timing functions need to be found manually.[4]

Next, the operation needs to be refined into a program (da_push) using LLVM currencies and only subprograms that come with suitable synthesis rules. Also, a concrete implementation (da_push_{impl}) with a synthesis rule need to be established. That program operates on a raw LLVM data structure that refines the abstract data structure via a refinement assertion (da_raw_{assn}).

[4] One could imagine using a semi-automatic procedure with templates to find candidates, though. See Carbonneaux's work on automated amortized resource analysis [17].

Finally, the synthesis rule can be composed with the currency refinement, the amortization refinement, and the data refinement to obtain the synthesis rule that connects the LLVM implementation(da_push_{impl}) with the abstract operation ($push\_list_{spec}\ T$). The data and amortization can be collapsed into a single refinement assertion (da_{assn}).

10.2.7 Reflections

While this methodology seems to be overkill for proving a data structure as simple as dynamic arrays, it actually shows which parts of the development there are and separate them to show which concerns can best be tackled on which level. The general mantra is to always prove properties of algorithms as much in the abstract as possible. For example the derivation of Φ and the advertised costs of the push operations is quite abstract and not covered under implementation details. Adding those can be postponed.

I believe that other amortized analyzes can follow the same pattern, and more involved case studies will show whether this structuring makes sense. A direct application would be to lift the Imperative-HOL-Time verification of union-find by Löwenberg [98] to the NREST level, reestablish the result for Imperative-HOL-Time and get a similar result for LLVM-Time. That is already possible, as LLVM-Time provides the only necessary basic data structure: arrays.

The limits of that approach might be reached when amortization happens at the level of pointer structures. This cannot be modeled at the NREST level easily. However, it might be possible to model the pointer structure as a general graph or a relation, and then reason about that in the abstract realm.

Moving the verification from a specific program logic to the abstract NREST-level allows to reuse parts of the verification with several back ends. At least the the proofs for establishing the *reclaim–elapse* pattern can be reused. They are independent from the back end. Also the theory about how to push resource usage from the *reclaim–elapse* program into the heaps can be adapted for other program logics.

As mentioned at the beginning of this section, for presentation purposes I have left out size constraints that are necessary to avoid overflows in the LLVM implementation. When doubling the list we have to make sure that the multiplication of the capacity with *2* does not lead to an overflow. We can restrict this by adding a size constraint to the synthesis rule demanding the length of the list may at most be half of *MAX_INT* before pushing an element to it. In a program that uses that operation, one then has to add assertions before those invocations that help the Sepref tool to discharge the respective size constraints. Those size constraints then can be propagated to the precondition of the program. For example, a depth-first search that uses a dynamic array to represent its waiting list might have an additional size constraint restricting the number of edges in the graph to *MAX_INT / 2*.

In the future work section of Chapter 5 I have mentioned that amortization via time credits can be used to model the running time analysis of dynamic programming (both bottom-up and top-down). I believe that using the methodology of this section one can also push the analysis of dynamic programming algorithms to the abstract level.

Maybe one could even combine the memoization monad of Wimmer et al. [134] with the NREST monad and automatically get memoized NREST-programs that then can be synthesized to imperative code.

10.3 Sorting Algorithms

Portions of this section appear in the paper "For a Few Dollars More – Verified Fine-Grained Algorithm Analysis Down to LLVM" (Haslbeck and Lammich [44]).

In this section, I present the application of our framework to the introsort algorithm [104]. The verification of its functional correctness [80] is used as a basis to verify its running time analysis and synthesize competitive efficient LLVM code for it. In contrast to an earlier publication I focus more on the structuring of the verification. Following the "top-down" mantra, several intermediate steps are used to refine a specification down to an implementation.

10.3.1 Specification of Sorting

We start with the specification of sorting a slice of a list.

$$
\begin{array}{l}
\mathit{slice\_sort\_aux}\ xs_0\ l\ h\ xs = \\
\quad (|xs| = |xs_0| \wedge xs[0:l] = xs_0[0:l] \wedge xs[h:|xs|] = xs_0[h:|xs|] \\
\qquad \wedge\ \mathit{mset}\ (xs_0[l:h]) = \mathit{mset}\ (xs[l:h]) \wedge \mathit{sorted}\ (xs[l:h]))
\end{array}
$$

$$
\begin{array}{l}
\mathit{slice\_sort}_{spec}\ T\ xs_0\ l\ h = \texttt{do}\ \{ \\
\quad \texttt{assert}\ (l \le h \wedge h \le |xs_0|); \\
\quad \texttt{spec}\ (\lambda xs.\ \mathit{slice\_sort\_aux}\ xs_0\ l\ h\ xs)\ (\lambda\_.\ T) \\
\}
\end{array}
$$

Here, the term $as[a:b]$ denotes the slice of as between a and b, $\mathit{mset}\ as$ is the multiset of a list as and $\mathit{sorted}\ xs$ characterizes lists xs that are sorted. The result xs of the specification is a permutation of xs_0, xs is sorted between l and h and equal to xs_0 anywhere else.

This specification can be implemented by several sorting algorithms. In the following I present the verification of introsort.

10.3.2 Introsort's Idea

The introsort algorithm is based on quicksort. Like quicksort, it finds a pivot element, partitions the list around the pivot, and recursively sorts the two partitions. Unlike quicksort, however, it keeps track of the recursion depth, and if it exceeds a certain value (typically $2\lfloor \log n \rfloor$), it falls back to heapsort to sort the current partition. Intuitively, quicksort's worst-case behavior can only occur when unbalanced partitioning causes a high recursion depth, and the introsort algorithm limits the recursion depth, falling back to the $O(n \log n)$ heapsort algorithm. This combines the good practical performance of quicksort with the good worst-case complexity of heapsort.

Our implementation of introsort follows the implementation of *libstdc++*, which includes a second optimization: a first phase executes quicksort (with fallback to heapsort), but stops the recursion when the partition size falls below a certain threshold τ. Then, a second phase sorts the whole list with one final pass of insertion sort. This exploits the fact that insertion sort is actually faster than quicksort for *almost-sorted* lists, i.e., lists where any element is less than τ positions away from its final position in the sorted list. While good a good value for the threshold τ needs to be determined empirically, it does not influence the worst-case complexity of the final insertion sort, which is $O(\tau n) = O(n)$ for constant τ. The threshold τ will be an implicit parameter from now on.

While the two-phase approach seems like a quite concrete optimization, the two phases are already visible in the abstract algorithmic structure, which is defined as follows in NREST-ecost:

```
introsort xs l h = do {
    assert(l ≤ h);
    n ← return (h −$sub l);
    if_c 1 <$lt n then do {
        xs ← almost_sort_spec $almost_sort xs l h;
        xs ← final_sort_spec $final_sort xs l h;
        return xs
      }
    else return xs
  }
```

Here, $almost\_sort_{spec}\ T$ specifies an algorithm that almost-sorts a list, consuming at most T resources and $final\_sort_{spec}\ T$ specifies an algorithm that sorts an almost-sorted list, consuming at most T resources.

The program *introsort* leaves trivial lists unchanged and otherwise executes the first and second phase. Its resource usage is bounded by the sum of the first and second phase and some overhead for the subtraction, comparison, and *if-then-else*. Using the verification condition generator we prove that *introsort* is correct, i.e., refines the specification of sorting a slice:

$$introsort\ xs\ l\ h \leq \Downarrow_C E_{is}\ (slice\_sort_{spec}\ \$_{sort}\ xs\ l\ h)$$

where $E_{is} = \uparrow\downarrow[sort{:=}introsort_{cost}]$ is the exchange rate used at this step and the total allotted cost for introsort is $introsort_{cost} = \$_{sub} + \$_{if} + \$_{lt} + \$_{almost\_sort} + \$_{final\_sort}$.

10.3.3 Quicksort Scheme

The first phase can be implemented in the following way:

```
1   introsort_aux μ xs l h = do {
2       d ← depth_spec $depth l h;
3       rec_c (λintrosort_rec (xs, l, h, d). do {
4           assert (l ≤ h);
```

```
5           n ← return (h −$sub l);
6           if_c τ <$lt n then
7             if_c d =$eq 0 then
8               slice_sort_spec ($sort_c (μ (h − l))) xs l h
9             else do {
10              (xs, m) ← partition_spec ($partition_c (h − l)) xs l h;
11              d' ← d −$sub 1;
12              xs ← introsort_rec (xs, l, m, d');
13              xs ← introsort_rec (xs, m, h, d');
14              return xs
15            }
16          }
17          else
18            return xs
19        ) (xs,l,h,d)
20    }
```

where $partition_{spec}$ partitions a slice into two non-empty partitions, returning the start index m of the second partition, and $depth_{spec}$ specifies the computation of $2\lfloor\log(h-l)\rfloor$.

Let us first analyze the recursive part: if the slice is shorter than the threshold τ, it is simply returned (line 15). Unless the recursion depth limit is reached, the slice is partitioned using $h - l$ $partition_c$ coins, and the procedure is called recursively for both partitions (lines 10-14). Otherwise, the slice is sorted at a price of $\mu\ (h-l)$ $sort_c$ coins (line 8). The function μ here represents the leading term in the asymptotic costs of the used sorting algorithm, and the $sort_c$ coin can be seen as the constant factor. This currency will later be exchanged into the respective currencies that are used by the sorting algorithm. Note that we use currency $sort_c$ to describe costs per comparison of a sorting algorithm, while currency *sort* describes the cost for a whole sorting algorithm.

Showing that the procedure results in an almost-sorted list is straightforward. The running time analysis, however, is a bit more involved. We presume a function μ that maps the length of a slice to an upper bound on the abstract steps required for sorting the slice. We will later use heapsort with $\mu_{nlogn}\ n = n \log n$.

Consider the recursion tree of a call of *introsort_rec*: we pessimistically assume that for every leaf in the recursion tree we need to call the fallback sorting algorithm. Furthermore, we have to partition at every inner node. This has cost linear in the length of the current slice. For each following inner level the lengths of the slices add up to the current one's, and so do the incurred costs. Finally we have some overhead at every level including the final one. The cost of the recursive part of *introsort_aux* is:

$$\begin{aligned} &introsort\_rec_{cost}\ \mu\ (n,\ d) = \$_{sort_c}\ (\mu\ n) + \$_{partition_c}\ d * n \\ &\quad + ((d + 1) * n) * (\$_{if}\ 2 + \$_{call}\ 2 + \$_{eq} + \$_{lt} + \$_{sub}\ 2) \end{aligned}$$

The correctness of the running time bound is proved by induction over the recursion of *introsort_rec*. If the recursion limit is reached ($d = 0$), the first summand pays for

the fallback sorting algorithm. If $d > 0$, part of the second summand pays for the partitioning of the current slice, then the list is split into two and the recursive costs are paid for by parts of all three summands. To bound the costs for the fallback sorting algorithm, μ needs to be *superadditive*: $\mu\ a + \mu\ b \leq \mu\ (a{+}b)$. In both cases, the third summand pays for the overhead in the current call.

If we set $d = \lfloor 2 \log n \rfloor$ and use an $O(n \log n)$ fallback sorting algorithm ($\mu = \mu_{nlogn}$), we have that $introsort\_rec_{cost}\ \mu_{nlogn}$ is in $O(n \log n)$.[5] In fact, any $d \in O(\log n)$ would do.

Before executing the recursive method, *introsort_aux* calculates the depth limit d. The correctness theorem then reads:

$$introsort\_aux\ \mu_{nlogn}\ xs\ l\ h \leq \Downarrow_C(E_{isa}(h - l))(almost\_sort_{spec}\ \$_{almost\_sort}\ xs\ l\ h)$$

with $E_{isa}\ n = \Updownarrow[almost\_sort{:=}\ \$_{depth} + introsort\_rec_{cost}\ \mu_{nlogn}\ (n, \lfloor 2 \log n \rfloor)]$.

Note that specifications typically use a single coin of a specific currency for their abstract operation, which is then exchanged for the actual costs, usually depending on the parameters.

This concludes the interesting part of the running time analysis of the first phase. It is now left to plug in an $O(n \log n)$ fallback sorting algorithm, and a linear partitioning algorithm.

Heapsort Independently of introsort, we have proved correctness and worst-case complexity of heapsort, yielding the following refinement lemma:

$$heapsort\ xs\ l\ h \leq \Downarrow_C(E_{hs}\ (h - l))\ (slice\_sort_{spec}\ \$_{sort}\ xs\ l\ h)$$

where $E_{hs}\ n = \Updownarrow[sort{:=}\ c_1 + log\ n * c_2 + n * c_3 + (n \log n) * c_4]$ for some constants $c_i :: ecost$.

Assuming[6] that $n \geq 2$, we can estimate $E_{hs}\ n\ sort \leq \mu_{nlogn}\ n * c$, for $c = c_1 + c_2 + c_3 + c_4$, and thus get, for $E_{hs'} = \Updownarrow[sort_c := c]$:

$$\begin{aligned}&\Downarrow_C(E_{hs}\ (h{-}l))\ (slice\_sort_{spec}\ \$_{sort}\ xs\ l\ h)\\ &\leq \Downarrow_C E_{hs'}\ (slice\_sort_{spec}\ (\$_{sort_c}\ (\mu_{nlogn}\ (h{-}l)))\ xs\ l\ h)\end{aligned}$$

By transitivity we then get

$$heapsort\ xs\ l\ h \leq \Downarrow_C E_{hs'}\ (slice\_sort_{spec}\ (\$_{sort_c}\ (\mu_{nlogn}\ (h - l)))\ xs\ l\ h)$$

Note that our framework allowed us to easily convert the abstract currency from a single operation-specific *sort* coin to a $sort_c$ coin for each comparison operation.

Partition and Depth Computation We implement partitioning with the Hoare partitioning scheme using the median-of-3 as the pivot element. Moreover, we implement the computation of the depth limit ($2\lfloor \log(h - l) \rfloor$) by a loop that counts how often we can divide by two until zero is reached. This yields the following refinement lemmas:

[5] More precisely, the sum over all (finitely many) currencies is in $O(n \log n)$.

[6] Note that this is a valid assumption, as heapsort will never be called for trivial slices.

$$pivot\_partition\ xs\ l\ h \leq \Downarrow_C E_{pp}\ (partition_{spec}\ (\$_{partition_c}\ (h - l))\ xs\ l\ h)$$
$$calc\_depth\ l\ h \leq \Downarrow_C (E_{cd}\ (h - l))\ (depth_{spec}\ \$_{depth}\ l\ h)$$

Combining the Refinements We replace $slice\_sort_{spec}$, $partition_{spec}$ and $depth_{spec}$ by their implementations *heapsort*, *pivot_partition* and *calc_depth*. We call the resulting implementation $introsort\_aux_2$, and prove

$$introsort\_aux_2\ xs\ l\ h \leq \Downarrow_C (E_{aux}\ (h - l))\ (introsort\_aux\ \mu_{nlogn}\ xs\ l\ h)$$

Here, the exchange rate E_{aux} combines the exchange rates $E_{hs'}$, E_{pp} and E_{cd} for the component refinements.

Transitive combination with the correctness lemma for *introsort_aux* then yields the correctness lemma for $introsort\_aux_2$:

$$introsort\_aux_2\ xs\ l\ h \leq \Downarrow_C (E_{isa2}\ (h - l))\ (almost\_sort_{spec}\ \$_{almost\_sort}\ xs\ l\ h\ )$$

With $E_{isa2}\ n = \Uparrow\Downarrow[almost\_sort{:=}\downarrow_C(E_{aux}\ n)\ (introsort\_aux_{cost}\ n)]$ where the operation $\downarrow_C E\ t$ applies an exchange rate to a resource function.

Refining Resources The stepwise refinement approach allows to structure an algorithm verification in a way that correctness arguments can be conducted on a high level and implementation details can be added later. Resource currencies permit the same for the resource analysis of algorithms: they summarize compound costs, allow reasoning on a higher level of abstraction and can later be refined into fine-grained costs. For example, in the resource analysis of *introsort_aux* the currencies $sort_c$ and $partition_c$ abstract the cost of the respective subroutines. The abstract resource argument is independent from their implementation details, which are only added in a subsequent refinement step, via the exchange rate E_{aux}.

10.3.4 Final Insertion Sort

The second phase is implemented by insertion sort, repeatedly calling the subroutine *insert*. The specification of *insert* for an index i captures the intuition that it goes from a slice that is sorted up to index $i - 1$ to one that is sorted up to index i. Insertion is implemented by moving the last element to the left, as long as the element left of it is greater (or the start of the list has been reached). Moving an element to its correct position takes at most τ steps, as after the first phase the list is almost-sorted, i.e., any element is less than τ positions away from its final position in the sorted list. Moreover, elements originally at positions greater τ will never reach the beginning of the list, which allows for the *unguarded* optimization. It omits the bounds check for those elements, saving one index comparison in the innermost loop. Formalizing these arguments yields the implementation *final_insertion_sort* that satisfies

$$final\_insertion\_sort\ xs\ l\ h \leq \Downarrow_C (E_{fis}(h - l))\ (final\_sort_{spec}\ \$_{final\_sort}\ xs\ l\ h)$$

where $E_{fis}\ n = \Uparrow\Downarrow[final\_sort{:=}final\_insertion_{cost}\ n]$, and $final\_insertion_{cost}\ n$ is linear in n.

Note that *final_insertion_sort* and $introsort\_aux_2$ use the same currency system. Plugging both refinements into *introsort* yields $introsort_2$ and the lemma

$$introsort_2\ xs\ l\ h \leq \Downarrow_C(E_{is2}(h - l))\ (introsort\ xs\ l\ h)$$

where the exchange rate E_{is2} combines the rates E_{isa2} and E_{fis}.

10.3.5 Separating Correctness and Complexity Proofs

Our formalization of heapsort maintains a binary heap and uses the crucial function *sift_down*. It restores the heap property by moving the top element down in the heap. To implement this function, we first prove correct a version $sift\_down_1$, which uses swap operations to move the element. Using swaps captures the natural algorithmic idea and simplifies the correctness proof. In a next step, we refine this to $sift\_down_2$, which saves the top element, then executes upward moves instead of swaps, and, after the last step, moves the saved top element to its final position. This optimization spares half of the memory accesses, exploiting the fact that the next swap operation will overwrite an element just written by the previous swap operation.

However, this refinement is not structural: it replaces swap operations by move operations, and adds an additional move operation at the end. At this point, we chose to separate the functional correctness and resource aspect, to avoid the complexity of a combined non-structural functional and currency refinement. It turns out that proving the complexity of the optimized version $sift\_down_2$ directly is straightforward.

Thus, as sketched in Section 8.3.5, we first focus on proving[7] functional correctness only $sift\_down_2 \leq sift\_down_1 \leq sift\_down_{spec}\ (\infty)$, ignoring the resource aspect. Separately, we prove $sift\_down_2 \leq_n \texttt{spec}\ (\lambda\_.\ True)\ sift\_down_{cost}$, and combine the two statements to get $sift\_down_2 \leq sift\_down_{spec}\ sift\_down_{cost}$.

10.3.6 Refining to LLVM

The above abstract programs implicitly come with a fixed type and comparison operator for the elements of the list to be sorted. Those programs use abstract operations and currencies for arithmetic operations on indexes, control flow, comparisons and read/write of a random-access iterator (abstracted by lists with update and lookup operations).

When we further assume an LLVM program that refines the comparison operator in LLVM, and specify how the random-access data structure should be implemented — we choose arrays — we can automatically synthesize an LLVM program *introsort_impl* that refines $introsort_2$, i.e., satisfies the theorem:

$$(introsort_{impl},\ introsort_2) \in array^d_{assn} \to snat^k_{assn} \to snat^k_{assn} \to array_{assn}$$

Combination with the refinement lemmas for $introsort_2$ and *introsort*, followed by conversion to a Hoare triple, yields our final correctness statement:

[7] Note that I have omitted the function parameters for better readability.

$$
\begin{array}{l}
l \le h \wedge h < |xs_0| \implies \\
\{\$(\mathit{introsort\_impl}_{cost}\ (h-l)) \star \mathit{array}_{assn}\ xs_0\ p \star \mathit{snat}_{assn}\ l\ l_\dagger \star \mathit{snat}_{assn}\ h\ h_\dagger\} \\
\quad \mathit{introsort\_impl}\ p\ l_\dagger\ h_\dagger \\
\{\lambda r.\ \exists_A xs.\ \mathit{array}_{assn}\ xs\ r \star \uparrow(\mathit{slice\_sort\_aux}\ xs_0\ l\ h\ xs) \\
\qquad\qquad \star \mathit{snat}_{assn}\ l\ l_\dagger \star \mathit{snat}_{assn}\ h\ h_\dagger\}
\end{array}
$$

Where $\mathit{introsort\_impl}_{cost} :: nat \to ecost$ is the cost bound obtained from applying the exchange rates E_{is} and then E_{is2} to $\$_{sort}$.

Note that this statement is independent of the Refinement Framework. Thus, to believe in its meaningfulness, one has to only check the formalization of Hoare triples, Separation Logic, and the LLVM semantics.

To formally prove the statement "*introsort_impl has complexity* $O(n \log n)$", we observe that $\mathit{introsort\_impl}_{cost}$ uses only finitely many currencies, and only finitely many coins of each currency. We define the overall number of coins as

$$\mathit{introsort\_impl}_{allcost}\ n = \Sigma c.\ \mathit{introsort\_impl}_{cost}\ n\ c,$$

which expands to

$$\mathit{introsort\_impl}_{allcost}\ n = 4693 + 5 * \ \mathit{log}\ n + 231 * n + 455 * (n \log n),$$

which, in turn, is routinely proved to be in $O(n \log n)$.

As a last step, we instantiate the element type to 64-bit unsigned integers and the comparison operation to LLVM's *icmp_ult* instruction, to obtain a program that sorts integers in ascending order. Our code generator can export this to actual LLVM text and a corresponding header file for interfacing our sorting algorithm from C or C++.

The extracted program in LLVM performs on par with the real world implementation of introsort from the GNUC++ Library (*libstdc++*). For detailed results I refer to Section 5.7 in the original paper [44].

As LLVM does not support generics, we cannot implement a replacement for C++'s generic *std::sort<T>*. However, by repeating the last step for different types and compare operators, we can implement a replacement for any fixed T.

10.3.7 Recap

We have defined abstract algorithms and proved their complexity. Using currencies helps structuring the refinement proofs. In particular, there is no need for verbose place holders as well as forming and instantiating locales. It feels more natural to reason that way. Note that currencies only make sense together with refinement. However, we still use locales in order to assume synthesis rules for some basic operation that we reuse.

10.4 Discussion and Related Work

In this section I will revisit the reflection from Section 7.1. I will discuss whether the design goals are met by the framework described in this part of my thesis. Ultimately, I review related work to the case studies of this chapter.

10.4.1 Discussion

For expressing algorithm sketches I have presented the NREST monad. Its do-notation allows one to write down pseudocode (e. g. Figure 8.3b) that — in my opinion — already comes quite near to what one finds in a textbook (e. g. Figure 7.1). Instead of writing English prose, subprograms are often written as specifications. While NREST programs are certainly readable by a machine, I claim that they are also readable by a human.

Stepwise refinement allows to fill in implementation details and allows to establish the refinement of algorithms of different levels of abstraction. The nondeterministic result monad allows to express qualitative properties about results of algorithms and the extension to NREST additionally allows to express quantitative properties about their execution.

We did not yet consider illustrating the execution of abstract algorithms, but I will comment on that at the end of this section. Furthermore, the algorithm sketches in NREST lack a precise operational meaning. However, the resource usage specified by a program in NREST is an abstraction of one aspect of the operational meaning of an implementation that can be synthesized from it.

Proofs in our framework often consist of annotating the code with invariants, applying a verification condition generator and then proving the verification conditions by automatic or interactive proof. Those proofs are readable by the proof assistant and most of the time readable by a human — especially when structured *Isar* [133] proofs are used. One problem that arises is that the correspondence between code and verification conditions is lost during the automatic proof. Using a labeled VCG [117, §6] might improve the situation.

Resource currencies are used to further structure the stepwise refinement process. They furthermore allow for a natural reasoning about the costs of an algorithm in terms of certain subprograms instead of in terms of the underlying cost model of the program semantics the algorithm will be implemented in. This mechanism allows to hide constants and make specifications more robust. It allows for a fine-grained analysis of the usage of specific operations. The obtained bounds can finally be analyzed for their asymptotic behavior using the formalization of Landau symbols using filter.

In addition to the desired features mentioned in Section 7.1, it is possible to extract competitive executable implementations from the algorithm sketches. Both functional correctness as well as the running time bounds are preserved by the synthesis.

I believe that we have the right tools to comfortably formalize the quantitative analysis of imperative algorithms. In that process humans still have to provide the right ideas by a sort of pseudocode and set up the refinement proofs. But they are assisted by automation that automatically extracts the hard parts of a refinement proof and automates the rest of the straightforward reasoning.

Limits of Automatic Methods In the introduction of this thesis I stated that the analysis of many algorithms and data structures is beyond the limit of current automatic tools. Already in Chapter 5 I have presented several involved case studies,

e. g. the median of medians selection algorithm or the union-find data structure, whose running time analysis involves the Akra–Bazzi theorem and laborious reasoning about invariants and potentials. I can imagine that with sufficient annotations by the user those challenges could be automated, but I guess automatic tools will not be able to find non-trivial solutions on their own. Finding tight running time bounds for complex algorithms involves reasoning about invariants and that is already complicated when only functional correctness is involved.

In this chapter I have presented two bigger case studies. Here, the same applies. However, when using a modular approach automation could support the verification of abstract algorithms where the running time of the components are already known. Often such abstract algorithms combine data structures or subprograms in simple patterns. It would be desirable to apply automatic tools on that level, integrate their solutions into the interactive proof and certify their correctness. I believe that automatic methods could be applied to algorithms on the right level of abstraction. If that fails, one still can switch to interactive proof.

Artifacts of the Verification Once an algorithm is verified in our framework, what can we do with it? The obvious first answer is: we can synthesize an executable program and use the implementation in some software component. But there is more.

Another idea is to use the algorithm sketches for teaching and education. Besides studying the formalized algorithm and the proofs of its analysis statically, also dynamic concepts are imaginable. In Section 7.1 we have seen that algorithm sketches also should be able to be illustrated with an example execution. For instance, Kruskal's algorithm — albeit being presented as an algorithm sketch which actually is nondeterministic — is illustrated with a sample execution in CLRS [24]. At least when all the refinements of the algorithm are structural (i. e. follow the lockstep refinement pattern) and the data refinement relation used is single-valued, one could execute one implementation instead of the abstract program and determine the value of the abstract program at any time during the execution. This actually is already true for programs in *nres*. For programs in NREST we additionally have the property to know how much time (in terms of abstract currencies) has passed after each subprogram. It would be interesting to investigate how one can effectively extract sample executions and illustrations with that idea. That might help students to interactively explore algorithms operationally and support their learning experience.

10.4.2 Related Work

I already discussed related work to the IRF and NREST in Section 7.3. Here I want to mention work that is related to the case studies of this section.

Amortized Analysis Rajani et al. [123] present a unifying type-theory λ^{amor} for higher-order (amortized) cost analysis. The introduction of the `elapse` combinator is straightforward, but the `reclaim` operator in NREST seems to be related to their type constructor $[p]\tau$. That constructor is central to their paper. It would be interesting to see

whether their cost monad can be extended to nondeterminism. Rajani [122] studies the type-theory λ^{cg} using a different domain for Information Flow Control. He then unifies the type-theories and their cost monads for a general domain being a commutative monoid. I suspect that the underlying mechanism is quite similar to ours.

Nipkow and Brinkop [112] verify several amortized data structures. Löwenberg [98] verifies the amortized union-find data structure. Integrating those verifications amortized data structures in NREST would be a next step.

Textbook Algorithms Nipkow *et al.* [113] collect verification efforts concerning textbook algorithms.

Apart from our formalization, there are several verifications of minimum spanning tree algorithms (cf. [113, §8.2]). But none of them verifies the running time. Besides the one in Isabelle by Keinholz [68], there are formalizations of matroids at least in Coq [99] and Mizar [5].

We add a few instances verifying sorting algorithms with their running time to [113, §4.1]: Wang *et al.* use TiML [131] to verify correctness and asymptotic time complexity of mergesort automatically. Zhan and Haslbeck [138] verify functional correctness and asymptotic running time analysis of imperative versions of insertion sort and mergesort. The verification in Section 10.3 builds on earlier work by Lammich [80] and provide the first verification of functional correctness and asymptotic running time analysis of heapsort and introsort.

10.5 Summary

In this chapter we have seen the following:

- NREST allows to reason abstractly about algorithms and their running time at a pseudocode level.
- Furthermore it still allows to extract efficient verified code.
- Amortized data structures can be analyzed at the NREST-level and refined down to executable code. That is an example for simple algorithms on involved data structures.
- Resource currencies can be effectively used to structure refinement of running times with abstract notions. It can be used for involved algorithms on simple data structures.

Part IV

Conclusion

11 Conclusion

To conclude this thesis, I will summarize my contributions, comment on possible directions for future work and end with general reflections on my work.

11.1 Results

The main contributions of my thesis are the following.

- A formal study of three Hoare logics for time bounds from the literature, formalized in a common framework in Isabelle/HOL, with soundness and completeness results of the logics and their respective VCGs.
- The verification of an extension of the quantitative Hoare logic for reasoning about randomized algorithms, the verification of the Quantitative Separation Logic and its generalization to arbitrary quantales.
- The implementation of a practical framework for reasoning about asymptotic time complexity of imperative programs in Isabelle/HOL together with an array of case studies.
- The combination of stepwise refinement with resource analysis for modular development of algorithm verification. The novel concept of resource currencies that allows for a natural formal treatment of algorithm analysis.
- The extension of the IRF and the Sepref tool for the time-bound preserving synthesis of efficient imperative implementations.
- The first verification of the asymptotic running time $O(n \log n)$ of an implementation of introsort.

11.2 Future Directions

I have mentioned several ideas for future work in my thesis, but I would like to collect some of them again and present them together with further general ideas.

Theory On the theory side I see several very interesting paths to take. The identification of quantales as the underlying structure needed for quantitative separating connectives gives rise to many questions. Clarifying the role of quantales is a major open question for me. Can the algebraic approach towards software verification [37] be

lifted along the lines of the potential method, in order to obtain a quantitative analysis? In general, it would be interesting to revisit the work from Chapter 2 and generalize the resource type, like we did for NREST in Chapter 8. This also includes exploring lower bounds. Studying other resource types for NREST, or lifting it to probability distributions to model randomized algorithms would be interesting future work.

Randomized Algorithms With the verification projects described in Chapter 3 and 4 we only touched the surface of verification of randomized algorithm. Based on the verification of the *ert* calculus and the quantitative separating connectives, there are many possibilities for further projects and opportunities to apply the theory to case studies. In particular, the integration of classical verification frameworks for deterministic programs with the *ert* calculus for probabilistic programs seems promising.

Verified Complexity Theory The NREST monad can serve as a formalism to reason about computations and reason about their quantitative properties. This is necessary for formalizing complexity theory and notions like P, NP and polynomial reductions. While those concepts are defined w. r. t. a concrete computational model with cost semantics, algorithms in complexity theory are mostly formulated very abstractly and their refinement is left implicit. To bridge the gap, stepwise refinement seems like a promising technique. Hooking up NREST with Turing machines or a simple imperative computational model using a component similar to Sepref could decouple low-level reasoning about programs and high level arguments typical in complexity theory. A goal in this direction is the verification of the Cook-Levin theorem.

Tooling While my focus was on developing the necessary theory and sufficient automation to make the verification of specific case studies feasible, turning the techniques into a framework that is effectively usable needs more tool support. Synthesizing cost expressions from a system of constraints (e. g. when finding the potentials in Section 10.2), or assistance in synthesizing exchange rates in the NREST-ecost framework are tools that are needed. Furthermore, it would be interesting to design a generalized Sepref tool, for the synthesis of implementations in a target monad with a heap from a different source monad. Properly integrating the classical Isabelle Refinement Framework and its data structures with their verified correctness proofs into the quantitative version would be desirable.

Libraries & Case Studies Building up libraries of verified data structures that come with verified resource bounds is an obvious next step. The verification of Fibonacci heaps needs to be finished and the flow framework formalization (cf. Section 5.5.3) might serve as a technique to verify overlaid data structures that are hard to handle with classical Separation Logic.

11.3 General Reflections

I think in essence the research question of how to verify quantitative analysis of imperative programs in a proof assistant is answered satisfactorily. Functional correctness and the running time analysis of concrete imperative programs is effectively described with time credits Separation Logic. Asymptotic analysis, formally defined with filters, is used to hide implementation details for robust specifications. Stepwise refinement is extended to reason about resource consumption, the concept of resource currencies further structures the algorithm development and the IRF "top-down" tool-chain is adapted to synthesize efficient imperative code that comes with verified running time bounds. Our case studies have shown that the verified quantitative analysis of imperative algorithms is feasible.

The most central idea in this thesis is the potential method, i. e. the step from qualitative predicates to quantitative potential. I first encountered it in the papers by Quentin Carbonneaux et al. [18, 17, 15], and it unfolded its potential in many parts of this thesis: be it the probabilistic quantitative Hoare logic, the quantitative separating connectives, or the generalization of the *nres* monad. But the story is not over yet. The line of research concerning the type-theoretic approach for amortized cost analysis has recently been advanced by Jan Hoffmann et al. [65, 119, 123]. Furthermore, the connection to quantales and its application to the algebraic approach to software verification is to be explored.

I conducted all the formalizations of this thesis in Isabelle/HOL and I think I would not have completed my contributions in another proof assistant. This is partly because of the system itself, which provides many mechanisms and tools that facilitate the work on formal proofs: type classes, locales, strong simplification and first-order automation, *sledgehammer*, the query panel and the Isabelle search tool, and many more. Furthermore, I have built my thesis upon the shoulders of many people whose work was conducted in Isabelle/HOL. The abstract analysis of Hoare logics for time bounds takes inspiration from Tobias Nipkow's work [110, 114], and we integrated his formal verification of amortized data structures [109, 112] into an imperative setting. I build upon Johannes Hölzl's formalization of probability theory and the *ert* calculus [62] for further work on probabilistic programs. We use Manuel Eberl's formalization of Landau symbols [31] and the impressive Akra–Bazzi method [32] to analyze the running time of imperative programs. Bohua Zhan's *auto2* tactic and his formalization of a Separation Logic for Imperative-HOL [137] formed the basis for Imperative-HOL-Time. There are many other smaller pieces of work in the Isabelle eco-system that I was able to conveniently use. Above all, the work by Peter Lammich on the Isabelle Refinement Framework gave me the blueprint for the theoretical and practical implementation of quantitative algorithm analysis following the "top-down" approach. At times it feels like I was just adding "time" to everything Peter was working on: first extending his Separation Logic of Imperative-HOL with time credits, and adapting his *sep_auto* tactic; then, extending the *nres* monad and its refinement calculus with time, adapting the *Sepref* tool and adding the time analysis to some of Peter's case studies; finally, extending LLVM with time and adding the verified $O(n \log n)$ time bound to

his introsort case study. On the theory side there are some innovations which are independent from Isabelle. On the tool side of my work, however, it is mainly incremental extensions, which would not have been possible for me without Peter's work.

While it is certainly possible to implement the ideas of this thesis in another proof assistant, I expect it to be considerably much more work given that many of the used formalizations would need to be ported first. However, it would be desirable to connect the Isabelle Refinement Framework with components of other proof assistants, foremost the CompCert compiler to obtain executable byte code from synthesized LLVM programs.

For me, the *fistful of dollars* paper [41] by Armaël Guéneau, Arthur Charguéraud and François Pottier was very influential. It came exactly at the right point in time when I was just beginning to work with Bohua Zhan on extending Imperative-HOL. Because it was implemented in Coq, we could not directly use it, but it provided us with the necessary ideas and techniques to come up with a practical framework in Isabelle/HOL and connect it with other components that are available there. I hope that our extension of stepwise refinement to resource consumption and the concept of *resource currencies* [44] will be equally stimulating to others and worth *a few dollars more*. I am curious what *good, bad and ugly* concepts will be presented in the final paper of the Dollars Trilogy at ESOP.

Bibliography

[1] Hannah Arndt. "Randomized Meldable Heaps: A more formal proof of a less simple probabilistic data structure." MA thesis. RWTH Aachen University, 2019.

[2] Robert Atkey. "Amortised Resource Analysis with Separation Logic." In: *ESOP*. Vol. 6012. Springer. 2010, pp. 85–103.

[3] Ralph Johan Back. "On the correctness of refinement steps in program development." PhD thesis. University of Helsinki Helsinki, Finland, 1978.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. ISBN: 978-0-387-98417-9. DOI: 10.1007/978-1-4612-1674-2.

[5] Grzegorz Bancerek and Yasunari Shidama. "Introduction to Matroids." In: *Formalized Mathematics* 16.4 (2008), pp. 325–332. DOI: 10.2478/v10037-008-0040-0.

[6] Callum Bannister, Peter Höfner, and Gerwin Klein. "Backwards and Forwards with Separation Logic." In: *Interactive Theorem Proving*. Ed. by Jeremy Avigad and Assia Mahboubi. Cham: Springer International Publishing, 2018, pp. 68–87. ISBN: 978-3-319-94821-8.

[7] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "How long, O Bayesian network, will I sample thee? - A program analysis perspective on expected sampling times." In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 186–213. DOI: 10.1007/978-3-319-89884-1_7.

[8] Kevin Batz et al. "Quantitative separation logic: a logic for reasoning about probabilistic pointer programs." In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[9] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality." In: *J. Autom. Reason.* 61.1-4 (2018), pp. 333–365. DOI: 10.1007/s10817-018-9455-7.

[10] Julian Brunner. "Formal Verification of Executable Complementation and Equivalence Checking for Büchi Automata." In: *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*. Ed. by Brijesh Dongol and Elena Troubitsyna. Vol. 12546. Lecture Notes in Computer Science. Springer, 2020, pp. 239–256. DOI: 10.1007/978-3-030-63461-2_13.

[11] Julian Brunner and Peter Lammich. "Formal Verification of an Executable LTL Model Checker with Partial Order Reduction." In: *J. Autom. Reason.* 60.1 (2018), pp. 3–21. DOI: 10.1007/s10817-017-9418-4.

[12] Julian Brunner, Benedikt Seidl, and Salomon Sickert. "A Verified and Compositional Translation of LTL to Deterministic Rabin Automata." In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 11:1–11:19. DOI: 10.4230/LIPIcs.ITP.2019.11.

[13] Lukas Bulwahn et al. "Imperative functional programming with Isabelle/HOL." In: *Lecture Notes in Computer Science* 5170 (2008), pp. 134–149.

[14] Cristiano Calcagno, Peter W O'Hearn, and Hongseok Yang. "Local action and abstract separation logic." In: *Logic in Computer Science, LICS 2007*. IEEE. 2007, pp. 366–378.

[15] Quentin Carbonneaux. "Modular and Certified Resource-Bound Analyses." http://cs.yale.edu/homes/qcar/diss/. PhD dissertation. Yale University, 2017.

[16] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. "End-to-end verification of stack-space bounds for C programs." In: *PLDI 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. http://www.cs.yale.edu/homes/qcar/data/veristack-paper.pdf. ACM. 2014, pp. 270–281.

[17] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. "Automated Resource Analysis with Coq Proof Objects." In: *International Conference on Computer Aided Verification, CAV, 2017*. Springer. 2017, pp. 64–85.

[18] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. "Compositional certified resource bounds." In: *Conference on Programming Language Design and Implementation, PLDI, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 467–478.

[19] Michael E. Caspersen and Michael Kölling. "A novice's process of object-oriented programming." In: *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. Ed. by Peri L. Tarr and William R. Cook. ACM, 2006, pp. 892–900. DOI: 10.1145/1176617.1176741.

[20] Arthur Charguéraud. "Separation logic for sequential programs (functional pearl)." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 116:1–116:34. DOI: 10.1145/3408998.

[21] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits." In: *Journal of Automated Reasoning* (2017), pp. 1–35.

[22] Arthur Charguéraud and François Pottier. "Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits." In: *J. Autom. Reason.* 62.3 (2019), pp. 331–365. DOI: 10.1007/s10817-017-9431-7.

[23] David Cock, Gerwin Klein, and Thomas Sewell. "Secure Microkernels, State Monads and Scalable Refinement." In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Mohamed, Muñoz, and Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 167–182. DOI: 10.1007/978-3-540-71067-7_16.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. "Introduction to algorithms third edition." In: (2009).

[25] Han-Hing Dang, Peter Höfner, and Bernhard Möller. "Algebraic separation logic." In: *J. Log. Algebraic Methods Program.* 80.6 (2011), pp. 221–247. DOI: 10.1016/j.jlap.2011.04.003.

[26] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. *Data refinement: model-oriented proof methods and their comparison*. 47. Cambridge University Press, 1998.

[27] Jose Divasón, Sebastiaan Joosten, René Thiemann, and Akihisa Yamada. "The Factorization Algorithm of Berlekamp and Zassenhaus." In: *Archive of Formal Proofs* (Oct. 2016). https://isa-afp.org/entries/Berlekamp_Zassenhaus.html, Formal proof development. ISSN: 2150-914x.

[28] Brijesh Dongol, Victor B. F. Gomes, and Georg Struth. "A Program Construction and Verification Tool for Separation Logic." In: *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer, 2015, pp. 137–158. DOI: 10.1007/978-3-319-19797-5_7.

[29] Brijesh Dongol, Ian J. Hayes, and Georg Struth. "Convolution as a Unifying Concept: Applications in Separation Logic, Interval Calculi, and Concurrency." In: *ACM Trans. Comput. Log.* 17.3 (2016), 15:1–15:25. DOI: 10.1145/2874773.

[30] Freeman Dyson. "Birds and frogs." In: *Notices of the AMS* 56.2 (2009), pp. 212–223.

[31] Manuel Eberl. "Landau Symbols." In: *Archive of Formal Proofs* (July 2015). http://isa-afp.org/entries/Landau_Symbols.html, Formal proof development. ISSN: 2150-914x.

[32] Manuel Eberl. "Proving Divide and Conquer Complexities in Isabelle/HOL." In: *Journal of Automated Reasoning* 58.4 (Apr. 2017), pp. 483–508. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9378-0.

[33] Manuel Eberl. "The Median-of-Medians Selection Algorithm." In: *Archive of Formal Proofs* (Dec. 2017). http://isa-afp.org/entries/Median_Of_Medians_Selection.html, Formal proof development. ISSN: 2150-914x.

[34] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. "Verified Analysis of Random Binary Tree Structures." In: *J. Autom. Reason.* 64.5 (2020), pp. 879–910. DOI: 10.1007/s10817-020-09545-0.

[35] Javier Esparza et al. "A Fully Verified Executable LTL Model Checker." In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 463–478. DOI: 10.1007/978-3-642-39799-8_31.

[36] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. "A verified SAT solver with watched literals using imperative HOL." In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. Ed. by June Andronick and Amy P. Felty. ACM, 2018, pp. 158–171. DOI: 10.1145/3167080.

[37] Victor Borges Ferreira Gomes. "Algebraic Principles for Program Correctness Tools in Isabelle/HOL." https://www.cl.cam.ac.uk/~vb358/articles/thesis.pdf. PhD thesis. University of Sheffield, UK, 2015. URL: http://etheses.whiterose.ac.uk/12457/.

[38] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem." In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[39] Simon Griebel. "Verification of the decrease-key operation in Fibonacci Heaps in Imperative HOL." MA thesis. Technical University of Munich, 2020.

[40] Armaël Guéneau. "Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)." PhD thesis. Inria, Paris, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02437532.

[41] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification." In: *European Symposium on Programming (ESOP)*. 2018.

[42] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. "Formal Proof and Analysis of an Incremental Cycle Detection Algorithm." In: *International Conference on Interactive Theorem Proving*. Springer. 2019. URL: http://gallium.inria.fr/~agueneau/publis/gueneau-jourdan-charguer aud-pottier-2019.pdf.

[43] Thomas C. Hales et al. "A formal proof of the Kepler conjecture." In: *CoRR* abs/1501.02155 (2015). arXiv: 1501.02155. URL: http://arxiv.org/abs/1501.02155.

[44] Maximilian P. L. Haslbeck and Peter Lammich. "For a Few Dollars More – Verified Fine-Grained Algorithm Analysis Down to LLVM." In: *ESOP 2021*.

[45] Maximilian P. L. Haslbeck and Peter Lammich. "Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL." In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 20:1–20:18. DOI: 10.4230/LIPIcs.ITP.2019.20.

[46] Maximilian P. L. Haslbeck and Tobias Nipkow. "Hoare Logics for Time Bounds." In: *Archive of Formal Proofs* (Feb. 2018). https://www.isa-afp.org/entries/Hoare_Time.html, Formal proof development.

[47] Maximilian P.L. Haslbeck. *Nondeterministic Result Monad with Time*. https://github.com/maxhaslbeck/NREST. 2020.

[48] Maximilian P.L. Haslbeck. *Quantitative Separating Connectives*. https://github.com/maxhaslbeck/QuantSepCon. 2020.

[49] Maximilian P.L. Haslbeck. *Verification of expected running time analysis of probabilistic programs in Isabelle/HOL*. https://github.com/maxhaslbeck/verERT. 2020.

[50] Maximilian P.L. Haslbeck and Peter Lammich. *Isabelle-LLVM with Time*. https://www21.in.tum.de/~haslbema/llvm-time/. 2020.

[51] Maximilian P.L. Haslbeck and Peter Lammich. *Sepref-Time*. https://www21.in.tum.de/~haslbema/Sepreftime/. 2019.

[52] Maximilian P.L. Haslbeck, Peter Lammich, and Julian Biendarra. "Kruskal's Algorithm for Minimum Spanning Forest." In: *Archive of Formal Proofs* (Feb. 2019). http://isa-afp.org/entries/Kruskal.html, Formal proof development. ISSN: 2150-914x.

[53] Maximilian Paul Louis Haslbeck and Tobias Nipkow. "Hoare Logics for Time Bounds." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018)*. Ed. by M. Huisman and D. Beyer. Vol. 10805. LNCS. Springer, 2018, pp. 155–171.

[54] Fabian Hellauer and Peter Lammich. "The string search algorithm by Knuth, Morris and Pratt." In: *Arch. Formal Proofs* 2017 (2017). URL: https://www.isa-afp.org/entries/Knuth%5C_Morris%5C_Pratt.html.

[55] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[56] C. A. R. Hoare. "Proof of Correctness of Data Representations." In: *Acta Informatica* 1 (1972), pp. 271–281. DOI: 10.1007/BF00289507.

[57] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Multivariate amortized resource analysis." In: *ACM Trans. Program. Lang. Syst.* 34.3 (2012), 14:1–14:62. DOI: 10.1145/2362389.2362393.

[58] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. "Towards automatic resource bound analysis for OCaml." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 359–373. URL: http://dl.acm.org/citation.cfm?id=3009842.

[59] Jan Hoffmann, Michael Marmar, and Zhong Shao. "Quantitative reasoning for proving lock-freedom." In: *Logic in Computer Science, LICS, 2013*. IEEE. 2013, pp. 124–133.

[60] Martin Hofmann and Steffen Jost. "Type-Based Amortised Heap-Space Analysis." In: *Programming Languages and Systems, ESOP 2006*. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 22–37.

[61] Martin Hofmann and Dulma Rodriguez. "Automatic Type Inference for Amortised Heap-Space Analysis." In: *Programming Languages and Systems, ESOP 2013*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 593–613.

[62] Johannes Hölzl. "Formalising Semantics for Expected Running Time of Probabilistic Programs." In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 475–482. DOI: 10.1007/978-3-319-43144-4_30.

[63] Johannes Hölzl. "Markov Chains and Markov Decision Processes in Isabelle/HOL." In: *J. Autom. Reason.* 59.3 (2017), pp. 345–387. DOI: 10.1007/s10817-016-9401-5.

[64] Fabian Immler. "Verified Reachability Analysis of Continuous Systems." In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 37–51. DOI: 10.1007/978-3-662-46681-0_3.

[65] David M. Kahn and Jan Hoffmann. "Exponential Automatic Amortized Resource Analysis." In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 359–380. DOI: 10.1007/978-3-030-45231-5_19.

[66] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest precondition reasoning for expected run–times of probabilistic programs." In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 364–389.

[67] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. "Recurrence extraction for functional programs through call-by-push-value." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 15:1–15:31. DOI: 10.1145/3371083.

[68] Jonas Keinholz. "Matroids." In: *Archive of Formal Proofs* (Nov. 2018). https://isa-afp.org/entries/Matroids.html, Formal proof development. ISSN: 2150-914x.

[69] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. "Separation Algebra." In: *Archive of Formal Proofs* (May 2012). http://isa-afp.org/entries/Separation_Algebra.html, Formal proof development. ISSN: 2150-914x.

[70] Gerwin Klein, Thomas Sewell, and Simon Winwood. "Refinement in the Formal Verification of the seL4 Microkernel." In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Springer, 2010, pp. 323–339. DOI: 10.1007/978-1-4419-1539-9_11.

[71] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[72] Thomas Kleymann. "Hoare logic and auxiliary variables." In: *Formal Aspects of Computing* 11.5 (1999), pp. 541–566.

[73] Dexter Kozen. "A Probabilistic PDL." In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. Ed. by David S. Johnson et al. ACM, 1983, pp. 291–297. DOI: 10.1145/800061.808758.

[74] Alexander Krauss. "Recursive Definitions of Monadic Functions." In: *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010*. Ed. by Ana Bove, Ekaterina Komendantskaya, and Milad Niqui. Vol. 43. EPTCS. 2010, pp. 1–13. DOI: 10.4204/EPTCS.43.1.

[75] Alexander Krauss et al. "Termination of Isabelle Functions via Termination of Rewriting." In: *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*. Ed. by Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk. Vol. 6898. Lecture Notes in Computer Science. Springer, 2011, pp. 152–167. DOI: 10.1007/978-3-642-22863-6_13.

[76] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. "Local Reasoning for Global Graph Properties." In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 308–335. DOI: 10.1007/978-3-030-44914-8_12.

[77] Peter Lammich. "Automatic Data Refinement." In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 84–99. DOI: 10.1007/978-3-642-39634-2_9.

[78] Peter Lammich. "Efficient Verified (UN)SAT Certificate Checking." In: *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. Springer, 2017, pp. 237–254. DOI: 10.1007/978-3-319-63046-5_15.

[79] Peter Lammich. "Efficient Verified (UN)SAT Certificate Checking." In: *J. Autom. Reason.* 64.3 (2020), pp. 513–532. DOI: 10.1007/s10817-019-09525-z.

[80] Peter Lammich. "Efficient Verified Implementation of Introsort and Pdqsort." In: *IJCAR 2020*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Lecture Notes in Computer Science. Springer, 2020, pp. 307–323. DOI: 10.1007/978-3-030-51054-1_18.

[81] Peter Lammich. "Generating Verified LLVM from Isabelle/HOL." In: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 22:1–22:19. DOI: 10.4230/LIPIcs.ITP.2019.22.

[82] Peter Lammich. "Refinement based verification of imperative data structures." In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. Ed. by Jeremy Avigad and Adam Chlipala. ACM, 2016, pp. 27–36. DOI: 10.1145/2854065.2854067.

[83] Peter Lammich. "Refinement for Monadic Programs." In: *Archive of Formal Proofs* (Jan. 2012). https://isa-afp.org/entries/Refine_Monadic.html, Formal proof development. ISSN: 2150-914x.

[84] Peter Lammich. “Refinement to Imperative HOL.” In: *J. Autom. Reason.* 62.4 (2019), pp. 481–503. DOI: 10.1007/s10817-017-9437-1.

[85] Peter Lammich. “Refinement to Imperative/HOL.” In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 253–269. DOI: 10.1007/978-3-319-22102-1_17.

[86] Peter Lammich. “The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees.” In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by Serge Gaspers and Toby Walsh. Vol. 10491. Lecture Notes in Computer Science. Springer, 2017, pp. 457–463. DOI: 10.1007/978-3-319-66263-3_29.

[87] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Component Algorithm.” In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 325–340. DOI: 10.1007/978-3-319-08970-6_21.

[88] Peter Lammich and Andreas Lochbihler. “Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches.” In: *J. Autom. Reason.* 63.1 (2019), pp. 53–94. DOI: 10.1007/s10817-018-9461-9.

[89] Peter Lammich and Andreas Lochbihler. “The Isabelle Collections Framework.” In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 339–354. DOI: 10.1007/978-3-642-14052-5_24.

[90] Peter Lammich and Rene Meis. “A Separation Logic Framework for Imperative HOL.” In: *Archive of Formal Proofs* (Nov. 2012). https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development. ISSN: 2150-914x.

[91] Peter Lammich and René Neumann. “A Framework for Verifying Depth-First Search Algorithms.” In: *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. Ed. by Xavier Leroy and Alwen Tiu. ACM, 2015, pp. 137–146. DOI: 10.1145/2676724.2693165.

[92] Peter Lammich and S. Reza Sefidgar. “Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL.” In: *J. Autom. Reason.* 62.2 (2019), pp. 261–280. DOI: 10.1007/s10817-017-9442-4.

[93] Peter Lammich and S. Reza Sefidgar. "Formalizing the Edmonds-Karp Algorithm." In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 219–234. DOI: 10.1007/978-3-319-43144-4_14.

[94] Peter Lammich and Thomas Tuerk. "Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm." In: *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*. Ed. by Lennart Beringer and Amy P. Felty. Vol. 7406. Lecture Notes in Computer Science. Springer, 2012, pp. 166–182. DOI: 10.1007/978-3-642-32347-8_12.

[95] Francis William Lawvere. "Metric spaces, generalized logic, and closed categories." In: *Rendiconti del seminario matématico e fisico di Milano* 43.1 (1973), pp. 135–166.

[96] Xavier Leroy. "A Formally Verified Compiler Back-end." In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[97] Junyi Liu et al. "Quantum Hoare Logic." In: *Archive of Formal Proofs* (Mar. 2019). https://isa-afp.org/entries/QHLProver.html, Formal proof development. ISSN: 2150-914x.

[98] Adrián Löwenberg Casas. "Proof af the Amortized time complexity of an efficient Union-Find data structure in Isabelle/HOL." BS Thesis. Technical University of Munich, 2019.

[99] Nicolas Magaud, Julien Narboux, and Pascal Schreck. "A case study in formalizing projective geometry in Coq: Desargues theorem." In: *Comput. Geom.* 45.8 (2012), pp. 406–424. DOI: 10.1016/j.comgeo.2010.06.004.

[100] Christoph Matheja. "Automated reasoning and randomization in separation logic." PhD thesis. RWTH Aachen University, Germany, 2020. URL: https://publications.rwth-aachen.de/record/780877.

[101] Jay A. McCarthy et al. "A Coq library for internal verification of running-times." In: *Sci. Comput. Program.* 164 (2018), pp. 49–65. DOI: 10.1016/j.scico.2017.05.001.

[102] Annabelle McIver, Carroll Morgan, and Charles Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

[103] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Time Credits and Time Receipts in Iris." In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 3–29. DOI: 10.1007/978-3-030-17184-1_1.

[104] David R. Musser. "Introspective Sorting and Selection Algorithms." In: *Software: Practice and Experience* 27.8 (1997), pp. 983–993.

[105] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. "Bounded Expectations: Resource Analysis for Probabilistic Programs." In: *Conference on Programming Language Design and Implementation, PLDI, 2018*. 2018.

[106] Hanne Riis Nielson. "A Hoare-like proof system for analysing the computation time of programs." In: *Science of Computer Programming* 9.2 (1987), pp. 107–136.

[107] Hanne Riis Nielson. "Hoare logic's for run-time analysis of programs." PhD thesis. University of Edinburgh, 1984.

[108] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer, 2007.

[109] Tobias Nipkow. "Amortized Complexity Verified." In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 310–324. DOI: 10.1007/978-3-319-22102-1_21.

[110] Tobias Nipkow. "Hoare Logics in Isabelle/HOL." In: *Proof and System-Reliability*. Ed. by H. Schwichtenberg and R. Steinbrüggen. Kluwer, 2002, pp. 341–367.

[111] Tobias Nipkow. "Verified Root-Balanced Trees." In: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. Ed. by Bor-Yuh Evan Chang. Vol. 10695. Lecture Notes in Computer Science. Springer, 2017, pp. 255–272. DOI: 10.1007/978-3-319-71237-6_13.

[112] Tobias Nipkow and Hauke Brinkop. "Amortized Complexity Verified." In: *J. Autom. Reason.* 62.3 (2019), pp. 367–391. DOI: 10.1007/s10817-018-9459-3.

[113] Tobias Nipkow, Manuel Eberl, and Maximilian P. L. Haslbeck. "Verified Textbook Algorithms. A Biased Survey." In: *ATVA 2020, Automated Technology for Verification and Analysis*. Ed. by Dang Van Hung and Oleg Sokolsky. Vol. 12302. LNCS. Invited paper. Springer, 2020, pp. 25–53.

[114] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[115] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[116] Yue Niu and Jan Hoffmann. "Automatic Space Bound Analysis for Functional Programs with Garbage Collection." In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. Ed. by Gilles Barthe, Geoff Sutcliffe, and Margus Veanes. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 543–563. URL: https://easychair.org/publications/paper/dcnD.

[117] Lars Noschinski. "Formalizing graph theory and planarity certificates." PhD thesis. Technische Universität München, 2016.

[118] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. "Reasoning About Recursive Probabilistic Programs." In: *Logic in Computer Science*. LICS '16. ACM, 2016, pp. 672–681.

[119] Long Pham and Jan Hoffmann. "Typable Fragments of Polynomial Automatic Amortized Resource Analysis." In: *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*. Ed. by Christel Baier and Jean Goubault-Larrecq. Vol. 183. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 34:1–34:19. DOI: 10.4230/LIPIcs.CSL.2021.34.

[120] André Platzer. "Differential Game Logic." In: *Archive of Formal Proofs* (June 2019). https://isa-afp.org/entries/Differential_Game_Logic.html, Formal proof development. ISSN: 2150-914x.

[121] Bernhard Pöttinger. "Verification of the Flow Framework from "Local Reasoning for Global Graph Properties"." MA thesis. LMU München, 2020.

[122] Vineet Rajani. "A type-theory for higher-order amortized analysis." PhD thesis. Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29104.

[123] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. "A unifying type-theory for higher-order (amortized) cost analysis." In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434308.

[124] Martin Schwenke and Brendan Mahony. "The essence of expression refinement." In: *Proc. of International Refinement Workshop and Formal Methods*. 1998, pp. 324–333.

[125] Georg Struth. "Quantales." In: *Archive of Formal Proofs* (Dec. 2018). https://isa-afp.org/entries/Quantales.html, Formal proof development. ISSN: 2150-914x.

[126] Daniel Stüwe. "Verification of Fibonacci Heaps in Imperative HOL." MA thesis. Technical University of Munich, 2019.

[127] Philip Wadler. "Monads for Functional Programming." In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 24–52. DOI: 10.1007/3-540-59451-5_2.

[128] Philip Wadler. "Theorems for Free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. Ed. by Joseph E. Stoy. ACM, 1989, pp. 347–359. DOI: 10.1145/99370.99404.

[129] Di Wang, David M. Kahn, and Jan Hoffmann. "Raising expectations: automating expected cost analysis with types." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 110:1–110:31. DOI: 10.1145/3408992.

[130] Peng Wang, Di Wang, and Adam Chlipala. “TiML: a functional language for practical complexity analysis with invariants.” In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 79:1–79:26. DOI: 10.1145/3133903.

[131] Peng Wang, Di Wang, and Adam Chlipala. “TiML: a functional language for practical complexity analysis with invariants.” In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 79:1–79:26. DOI: 10.1145/3133903.

[132] Eelis van der Weegen and James McKinna. “A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq.” In: *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*. Ed. by Stefano Berardi, Ferruccio Damiani, and Ugo de’Liguoro. Vol. 5497. Lecture Notes in Computer Science. Springer, 2008, pp. 256–271. DOI: 10.1007/978-3-642-02444-3_16.

[133] Markus Wenzel. “Isabelle, Isar - a versatile environment for human readable formal proof documents.” PhD thesis. Technical University Munich, Germany, 2002. URL: http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf.

[134] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. “Verified Memoization and Dynamic Programming.” In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 579–596. DOI: 10.1007/978-3-319-94821-8_34.

[135] Simon Wimmer and Peter Lammich. “The Floyd-Warshall Algorithm for Shortest Paths.” In: *Arch. Formal Proofs* 2017 (2017). URL: https://www.isa-afp.org/entries/Floyd%5C_Warshall.shtml.

[136] Simon Wimmer and Peter Lammich. “Verified Model Checking of Timed Automata.” In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 61–78. DOI: 10.1007/978-3-319-89960-2_4.

[137] Bohua Zhan. “Efficient verification of imperative programs using auto2.” In: *TACAS 2018*. Ed. by Dirk Beyer and Marieke Huisman. 2018, pp. 23–40.

[138] Bohua Zhan and Maximilian P. L. Haslbeck. “Verifying asymptotic time complexity of imperative programs in Isabelle.” In: *International Joint Conference on Automated Reasoning*. Springer. 2018, pp. 532–548.

[139] Bohua Zhan and Maximilian P.L. Haslbeck. *Verifying asymptotic time complexity of imperative programs in Isabelle*. https://github.com/bzhan/Imperative_HOL_Time. 2018.