# Algorithmic Resource Verification

THÈSE N$^{O}$ 7885 (2017)

PRÉSENTÉE LE 9 NOVEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ravichandhran KANDHADAI MADHAVAN

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. R. Majumdar, rapporteur
Dr G. Ramalingam, rapporteur
Prof. J. Larus, rapporteur


ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

This thesis is dedicated to my maternal grandmother. Observing her I learnt what it means to work hard and be devoted.

# Acknowledgements

As I reflect upon my journey through my PhD I feel a sense of elation but at the same time an overwhelming sense of gratitude to many people who made me realize this dream. I take this opportunity to thank them all. First and foremost, I thank my advisor Professor Viktor Kuncak for the very many ways in which he has helped me during the course of my PhD. It is impossible for me to imagine a person who would have more faith and confidence in me than my advisor. His presence felt like having a companion with immense knowledge and experience who is working side-by-side with me towards my PhD. Besides his help with the technical aspects of the research, he was a great source of moral support. He shared my elation and pride when I was going through a high phase, my disappointment and dejection during paper rejections, and my effort, nervousness and excitement while working on numerous submission deadlines. I also thank him for all the big responsibilities that he had trusted me with: mentoring interns, managing several aspects of teaching like setting up questions, leading teaching assistants and giving substitute lectures. These were great experiences that I, retrospectively, think I am very fortunate to have had. I am very grateful to him that he had overlooked many of my blunders and mistakes in discharging my responsibilities and have always been very kind and supportive. Since it is impossible to elucidate all ways in which he has helped me, I would conclude by saying that I would always be proud to be called his advisee and I hope that I can live up to the trust and confidence he has in me in the future as well.

I thank Dr. G. Ramalingam for being a constant source of inspiration and guidance all through my research career. He has helped me in my career in numerous ways: first as a great teacher, later as a great mentor and always as a great, supportive friend. I developed and honed much of my research skills by observing him. What I find amazing and contagious is his rational and deep reasoning not just about research problems but also about issues in life. I hope to have absorbed a bit of that skill having been in close acquaintance with him for these many years. I thank him for motivating me to pursue a PhD, to join EPFL and to work with my advisor. This thesis would not have been possible without his advise and motivation. I am very happy and proud that this dissertation is reviewed by him.

I thank Professor Martin Odersky for being an excellent teacher and a source of inspiration. I greatly enjoyed the conversations we had and enjoyed working with him as a teaching assistant. I am very grateful to him for being very supportive about my research, and for presiding over the defense jury. I thank other members of the defense committee: Professor Jim Larus and Professor Rupak Majumdar for reviewing my dissertation and providing me very valuable

comments and feedback on the research, despite their very busy schedules. I greatly enjoyed discussing my research with them.
I thank Pratik Fegade and Sumith Kulal who in the course of their internships helped with improving the system and the approach described in this dissertation. Particularly, I thank Pratik Fegade for contributions to the development of the techniques detailed in section 3.8 of this dissertation, and Sumith Kulal for helping carry out many experiments detailed in section 5. I thank my current and past lab mates Mikaël Mayer, Jad Hamza, Manos Koukoutos, Nicolas Voirol, Etienne Kneuss, Tihomir Gvero, Regis Blanc, Eva Darulova, Manohar Jonalagedda, Ivan Kuraj, Hossein Hojjat, Phillipe Suter, Andrew Reynolds, Mukund Raghothaman, Georg Schimd, Marco Antognini, Romain Edelmann, and also many other friends and colleagues. I enjoyed the numerous discussions I had with them in the lab, over meals, during outing and in every other place we were caught up in. In their company I felt like a truly important person. In particular, I thank Mikaël Mayer for being an amazing friend and collaborator all through my PhD life – with him I could freely share my successes as well as my frustrations. I thank Fabien Salvi for providing great computing infrastructure and for accommodating each and every request of mine no matter how urgent and time constrained they were. I thank Sylvie Jankow for her kindness and help with administrative tasks.
Furthermore, I thank Professor K.V. Raghavan for being an excellent advisor of my master's thesis and for being a constant source of encouragement. I thank him for persuading me to continue with a research career and eventually pursue PhD. I am also thankful to him and Professor Deepak D'Souza for introducing me to the area of program analysis. I fell in love with this amazing field at that very instant, and it feels like the rest of my research path was a foregone conclusion. I thank Dr. Sriram Rajamani for being a great source of inspiration and support in my research career.
Last but not the least, I thank my amazing family for their never ending love, support and sacrifices. I thank my parents for, well, everything I am today. I thank my sister, my grandmother and my brother-in-law for their limitless love and support. I thank my nieces Abi and Apoorva for showing me that life outside research can also be awesome. It is funny that they were born, grew and went to school while I was still pursuing my PhD. And, of course, I thank my wife Nikhila for her love and encouragement and, more importantly, for tolerating all my quirks. I thank numerous other friends, colleagues and relatives who I, unfortunately, could not mention but have made my PhD years the best phase in my life so far.

*Lausanne, August 2017* Ravichandhran Kandhadai Madhavan

# Preface

Is the program I have written efficient? This is a question we face from the very moment we discover what programming is all about. The improvements in absolute performance of hardware systems have made this question more important than ever, because analytical behavior becomes more pronounced with large sizes of the data that today's applications manipulate.

Despite the importance of this question, there are surprisingly few techniques and tools that can help the developer answer such questions. Program verification is an active area of research, yet most approaches for software focus on safety verification of the values that the program computes. For performance, one often relies on testing, which is not only very incomplete, but provides little help in capturing the reasons for the behavior of the program.

This thesis presents a system that can automatically verify efficiency of programs. The verified bounds are guaranteed to hold for all program inputs. The prototypical examples of resources are time (for example, the number of operations performed), or notions related to memory usage (for example, the number of allocation steps performed). These bounds are properties of the algorithms, not of the unerlying hardware implementation.

The presented system is remarkable in that it suffices for the developers to provide only sketches of the bounds that are expected to hold (corresponding to expected asymptotic complexity). The system can infer the concrete coefficients automatically and confirm the resource bounds.

The paradigm supported by the approach in this thesis is purely functional, making it feasible to consider verification of very sophisticated data structures. Indeed, a "by product" of this thesis is that the author verified a version of rather non-trivial Conc Trees data structure that was introduced by Dr. Aleksandar Prokopec, and that play an important role in parallel collections framework of Scala.

The thesis deals with subtleties of verifying performance of programs in the presence of higher-order functions. What is more, the thesis supports efficient functional programming in practice through treatment of the construct inspired by imperative behavior: memoization (caching). Memoization, and it special case, lazy evaluation, are known to improve efficiency of functional programs both in practice and in asymptotic terms, so they make functional programming languages more practical. Yet, reasoning about the performance in the presence of these constructs introduces substantial aspects of state into the model. The thesis shows how to make specification and verification feasible even under this challanging scenario.

## Preface

Verification of program correctness has been a long road, and we are starting to see practical solutions that are cost-effective, especially for functional programs. With each step along this road, the gap in reasoning ability between developers and tools is narrowing. The work in this thesis makes a big step along an under-explored dimension of program meaning—reasoning about performance bounds. Given the extent to which program development is driven by performance considerations, closing this gap is likely to have not only great benefits for verifying programs, but will also open up new applications that leverage reliable performance information to improve and adapt software systems.
The tools are waking up to the notion of verified performance as program metric, and this will make them even more profoundly important for software development.

*Lausanne, Summer 2017* Viktor Kunčak

# Abstract

Static estimation of resource utilization of programs is a challenging and important problem with numerous applications. In this thesis, I present new algorithms that enable users to specify and verify their desired bounds on resource usage of functional programs. The resources considered are algorithmic resources such as the number of steps needed to evaluate a program (steps) and the number of objects allocated in the memory (alloc). These resources are agnostic to the runtimes and platforms on which the programs are executed yet provide a concrete estimate of the resource usage of an implementation. Our system is designed to handle sophisticated functional programs that use recursive functions, datatypes, closures, memoization and lazy evaluation.

In our approach, users can specify in the contracts of functions an upper bound they expect to hold on the resource usages of the functions. The upper bounds can be expressed as templates with numerical holes. For example, a bound steps ≤ ?*size(inp) + ? denotes that the number of evaluation steps is linear in the size of the input inp. The templates can be seamlessly combined with correctness invariants or preconditions necessary for establishing the bounds. Furthermore, the resource templates and invariants are allowed to use recursive and first-class functions as well as other features supported by the language. Our approach for verifying such resource templates operates in two phases. It first reduces the problem of resource inference to invariant inference by synthesizing an instrumented first-order program that accurately models the resource usage of the program components, the higher-order control flow and the effects of memoization, using algebraic datatypes, sets and mutual recursion. The approach solves the synthesized first-order program by producing verification conditions of the form ∃∀ using a modular assume/guarantee reasoning. The ∃∀ formulas are solved using a novel counterexample-driven algorithm capable of discovering strongest resource bounds belonging to the given template.

I present the results of using our system to verify upper bounds on the usage of algorithmic resources that correspond to sequential and parallel execution times, as well as heap and stack memory usage. The system was evaluated on several benchmarks that include advanced functional data structures and algorithms such as balanced trees, meldable heaps, Okasaki's lazy data structures, streams, sorting algorithms, dynamic programming algorithms, and also compiler phases like optimizers and parsers. The evaluations show that the system is able to infer hard, nonlinear resource bounds that are beyond the capability of the existing approaches. Furthermore, the evaluations presented in this dissertation show that, when averaged over many benchmarks, the resource consumption measured at runtime is 80% of

the value inferred by the system statically when estimating the number of evaluation steps and is 88% when estimating the number of heap allocations.

Key words: verification, static analysis, complexity, resource usage, decision procedures

# Résumé

L'analyse statique de la consommation en resources des programmes est un problème important avec de nombreuses applications possibles. Dans cette thèse, je présente de nouveaux algorithmes pour vérifier la consommation en ressources des programmes fonctionnels, algorithmes qui permettent aux utilisateurs de spécifier à leur guise des limites en resources, et de les vérifier. Les ressources considérées sont des ressources dites algorithmiques, comme par exemple le nombre d'étapes nécessaires pour évaluer un programme (steps) ou le nombre total d'objets qu'il crée en mémoire (alloc). Ces resources sont indépendantes de la plate-forme, bien qu'elles fournissent une mesure concrète de la consommation des implémentations d'algorithmes. Notre système peut analyser des programmes fonctionnels sophistiqués qui comportent des fonctions récursives, des données typées, des fonctions ayant capturé des variables (fermetures), des mises en cache (mémoïsations) ou des évaluations paresseuses.

Grâce à notre approche, les utilisateurs peuvent, dans les contrats de fonctions, spécifier une limite supérieure à l'utilisation de ressources sous la forme de modèles à trous numériques, par exemple steps ≤ ?*size(l) + ?. Les limites en ressources peuvent être combinées avec des invariants ou des spécifications nécessaires à l'établissement de ces limites. Les limites en ressources et les invariants peuvent également utiliser des fonctions récursives et les fonctions elles-mêmes comme des valeurs, ainsi que d'autres fonctionnalités prises en charge par le langage que nous avons développé. L'approche visant à vérifier les modèles à trous comporte deux phases. La première phase réduit d'abord le problème de l'inférence des ressources en inférence d'invariant, en convertissant le programme en un programme instrumenté et du premier niveau (sans les fonctions comme valeurs). Ce programme modélise avec précision l'utilisation des ressources, le flux de contrôle de plus haut niveau et les effets de la mémoïsation, en utilisant des types de données algébriques, des ensembles et de la récursion mutuelle. La deuxième phase vérifie ce programme en produisant des conditions de vérification de la forme ∃∀ et en utilisant un raisonnement modulaire. Les formules ∃∀ sont résolues à l'aide d'un nouvel algorithme. Cet algorithme tire profit de contre-exemples pour découvrir les limites en ressources les plus précises pour le modèle donné.

Je présente les résultats de l'utilisation de notre système, lorsque celui-ci vérifie le plus précisément possible les limites en les ressources algorithmiques qui correspondent aux temps d'exécution séquentiels et parallèles, ainsi qu'à l'utilisation de la mémoire du tas et de la pile. Nos tests, huit mille lignes de code en Scala, contiennent des arbres équilibrés comme les arbres bicolores, des tas fusionnables, l'analyse statique de la propagation de constantes, des algorithmes de tri paresseux comme le tri fusion paresseux, des structures de données

paresseuses comme les queues de temps constant d'Okasaki, des listes paresseuses cycliques, des parseurs et des algorithmes de programmation dynamique comme le problème du sac à dos. Les évaluations montrent que notre système est capable d'inférer de difficiles limites en ressources non linéaires, surpassant ainsi les approches existantes. Moyennés sur l'ensemble des tests, les résultats indiquent que, lors de l'exécution, la consommation en ressources est de 80 % de la valeur inférée par notre système lors de l'estimation de steps, et de 88 % lors de l'estimation de alloc.

Mots clefs : vérification, analyse statique, complexité, utilisation des ressources, procédures de décision

# Contents

**Acknowledgements** i

**Preface** iii

**Abstract** v

**List of figures** xiii

**1 Introduction** 1
- 1.1 Overview of the Specification Approach . . . 6
  - 1.1.1 Prime Stream Example . . . 7
- 1.2 Summary of Contributions . . . 11
- 1.3 Outline of the Thesis . . . 12

**2 Semantics of Programs, Resources and Contracts** 13
- 2.1 Syntax of the Core Language . . . 14
- 2.2 Notation and Terminology . . . 16
- 2.3 Resource-Annotated Operational Semantics . . . 18
  - 2.3.1 Semantic Domains . . . 18
  - 2.3.2 Resource Parametrization . . . 19
  - 2.3.3 Structural Equivalence and Simulation . . . 20
  - 2.3.4 Semantic Rules . . . 22
- 2.4 Reachability Relation . . . 23
- 2.5 Contract and Resource Verification Problem . . . 24
  - 2.5.1 Valid Environments . . . 24
  - 2.5.2 Properties of Undefined Evaluations . . . 25
  - 2.5.3 Problem Definition . . . 27
- 2.6 Proof Strategies . . . 27
- 2.7 Properties of the Semantics . . . 28
  - 2.7.1 Encapsulated Calls . . . 30

**3 Solving Resource Templates with Recursion and Datatypes** 31
- 3.1 Resource Instrumentation . . . 32
  - 3.1.1 Instrumentation for Depth . . . 34
- 3.2 Modular, Assume-Guarantee Reasoning . . . 37

3.2.1 Function-Level Modular Reasoning . . . 37
3.2.2 Function-level Modular Reasoning with Templates . . . 42
3.3 Template Solving Algorithm . . . 42
3.3.1 Verification Condition Generation . . . 44
3.3.2 Successive Function Approximation by Unfolding . . . 47
3.3.3 Logic Notations and Terminology . . . 48
3.3.4 The solveUNSAT procedure . . . 49
3.4 Completeness of Template Solving Algorithm . . . 55
3.4.1 Completeness of solveUNSAT Procedure . . . 55
3.5 Solving Nonlinear Formulas with Holes . . . 62
3.6 Finding Strongest Bounds . . . 63
3.7 Analysis Strategies and Optimizations . . . 64
3.8 Divide-and-Conquer Reasoning for Steps Bounds . . . 65
3.9 Amortized Analysis . . . 68

**4 Supporting Higher-Order Functions and Memoization 71**
4.1 Semantics with Memoization and Specification Constructs . . . 72
4.1.1 Semantic Rules . . . 74
4.2 Referential Transparency and Cache Monotonicity . . . 77
4.3 Proof of Referential Transparency . . . 78
4.4 Generating Model Programs . . . 81
4.4.1 Model Transformation . . . 82
4.5 Soundness and Completeness of the Model Programs . . . 89
4.5.1 Correctness of Model Transformation . . . 92
4.5.2 Completeness of Model Transformation . . . 96
4.6 Model Verification and Inference . . . 97
4.6.1 Creation-Dispatch Reasoning . . . 99
4.7 Correctness of Creation-Dispatch Reasoning . . . 100
4.7.1 Partial Correctness of Creation-Dispatch Obligations . . . 101
4.8 Encoding Runtime Invariants and Optimizations . . . 107

**5 Empirical Evaluation and Studies 111**
5.1 First-Order Functional Programs and Data Structures . . . 112
5.1.1 Benchmark Descriptions . . . 112
5.1.2 Analysis Results . . . 115
5.1.3 Comparison with CEGIS and CEGAR . . . 118
5.2 Higher-Order and Lazy Data Structures . . . 120
5.2.1 Measuring Accuracy of the Inferred Bounds . . . 121
5.2.2 Scheduling-based Lazy Data Structures . . . 123
5.2.3 Other Lazy Benchmarks . . . 129
5.3 Memoized Algortihms . . . 130

**6 Related Work** **135**
6.1 Resource Analyses . . . . . 135
6.2 Higher-Order Program Verification . . . . . 137
6.3 Software Verification . . . . . 138

**7 Conclusion and Future Work** **141**

**Bibliography** **154**

**Curriculum Vitae** **155**

# List of Figures


1.1 Relationship between number of steps and wall-clock execution time for a lazy selection sort implementation . . . 2
1.2 Illustration of verifying resource bounds using contracts . . . 6
1.3 Prime numbers until $n$ using an infinite stream. . . . 8
1.4 Specifying properties dependent on memoization state. . . . 9

2.1 Syntax of types, expressions, functions, and programs . . . 15
2.2 Resource-annotated operational semantics of the core language . . . 17
2.3 Definition of the reachability relation . . . 24

3.1 Resource instrumentation for first-order programs. . . . 33
3.2 Illustration of instrumentation. . . . 33
3.3 Example illustrating the depth of an expression. . . . 34
3.4 Illustration of depth instrumentation. . . . 35
3.5 Instrumentation for the depth resource. . . . 36
3.6 Definition of the path condition for an expression belonging to a program $P$ . . . 39
3.7 Counter-example guided inference for numerical holes. . . . 43
3.8 The solveUNSAT procedure . . . 50

4.1 Operational semantics of higher-order specifications and memoization. . . . 73
4.2 Syntax and semantics of the set operations used by the model programs . . . 82
4.3 A constant-time, lazy take operation . . . 83
4.4 Illustration of the translation on lazy take example . . . 84
4.5 Representation of closure and cache keys . . . 84
4.6 Translation of types in a program $P$ with a set of type declarations $Tdef_P$ . . . 85
4.7 Resource and cache-state instrumentation of source expressions . . . 87

5.1 Benchmarks implemented as first-order functional Scala programs . . . 113
5.2 Results of running ORB on the first-order benchmarks . . . 115
5.3 Results of inferring bounds on depths of benchmarks . . . 117
5.4 Results of inferring bounds on the number of heap-allocated objects . . . 117
5.5 Results of inferring bounds on the call-stack usage . . . 118
5.6 Higher-order, lazy benchmarks comprising 4.5K lines of Scala code . . . 120
5.7 Steps and Alloc bounds inferred by ORB for higher-order, lazy benchmarks . . . 120

5.8 Comparison of the resource usage bounds inferred statically against runtime resource usage . . . 121
5.9 Rotate function of the Real-time queue data structure . . . 124
5.10 Definition of Okasaki's Real-time queue data structure . . . 125
5.11 Queue operations of Okasaki's Real-time queue data structure . . . 126
5.12 Invariants of *conqueue* data structure [Prokopec and Odersky, 2015] . . . 128
5.13 Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for lazy merge sort . . . 130
5.14 Memoized algorithms verified by ORB . . . 131
5.15 Accuracy of bounds inferred for memozied programs . . . 131
5.16 Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for Levenshtein distance algorithm . . . 132
5.17 Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for *ks* . . . 132

# 1 Introduction

*How fast can a computer program solve this problem?* Answering this question is at the very heart of computer science. It wouldn't be an exaggeration to say that the word *fast* in the above question primarily distinguishes computer science from conventional mathematics. Developing computer programs that solves a problem faster or with reduced resource usage is a subject of enormous practical value which has obsessed practitioners and theoreticians alike. This quest for better performance has led to remarkable algorithms and theoretical results, which are routinely implemented and deployed at large scales and thus profoundly influencing our modern civilization by driving scientific discoveries, commerce and social interaction. However, a question that developers are often faced with is whether an implementation of an algorithm conforms to the performance expected out of it. The techniques presented in this dissertation are aimed at addressing this challenge.

Unfortunately, statically determining the resource usage of a program has proven to be very challenging. This is not only because the space of possible inputs of realistic programs is huge (if not infinite), but also because of the sophistication in the modern runtimes, like virtualization, on which the programs are executed. On the one hand this complexity poses serious impediment to developing automated tools that can help with reasoning about performance, on the other it has increased the need for developing such tools since programmers are also faced with similar (if not more) difficulties in analyzing the resource usage of programs. These challenges have resulted in wide ranging techniques for static estimation of resource usage of programs that model the resource usage at various levels of abstraction.

**Algorithmic Resources** Approaches such as those described by Wilhelm et al. [2008] and Carbonneaux et al. [2014] aim at estimating resource usage of programs in terms of concrete physical quantities (e.g. seconds, bytes etc.) under controlled environments, like embedded systems, for restricted class of programs where the number of loop iterations is a constant or is independent of the inputs. On the other extreme there are the static analysis tools that derive *asymptotic* upper bounds on the resource usage of general-purpose programs [Albert et al., 2012, Gulwani et al., 2009, Nipkow, 2015]. Using concrete physical quantities to measure

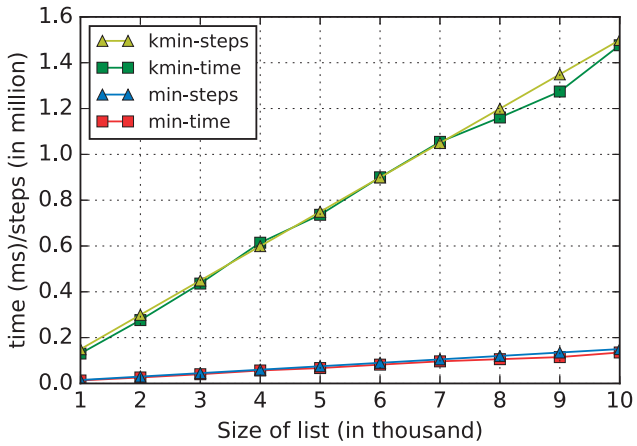



Figure 1.1 – Relationship between number of steps and wall-clock execution time for a lazy selection sort implementation

resource usage has the disadvantage that they are specific to a runtime and hardware, and applicable to only restricted programs and runtimes. However, the alternative of using asymptotic complexities results in overly general estimates for reasoning about implementations, especially for applications like compiler optimizations or for comparing multiple implementations. For instance, a program executing ten operations on each input and another executing a million operations on every input have the same asymptotic complexity of $O(1)$. For these reasons, recent techniques such as Resource Aware ML [Hoffmann et al., 2012, 2017] have resorted to more algorithmic measures of resource usage, such as the number of steps in the evaluation of an expression (commonly referred to as steps) or the number of memory allocations (alloc). These resources have the advantage that they are fairly independent of the runtime, but at the same time provide more concrete information about the implementations. This dissertation, which is a culmination of the prior research works: [Madhavan and Kuncak, 2014, Madhavan et al., 2017], further advances the static estimation of such algorithmic measures of resource usage to functional programs with recursive functions, recursive datatypes, first-class functions, lazy evaluation and memoization.

Although the objective of our approach is not to compute bounds on physical time, our *initial* experiments do indicate a strong correlation between the number of steps performed at runtime and the actual wall-clock execution time for our benchmarks. Figure 1.1 shows a plot of the wall-clock execution time and the number of steps executed by a function that computes the $k^{th}$ minimum of an unsorted list using a *lazy* selection sort implementation. The figure shows a strong correlation between one step performed at runtime and one nanosecond. (The bounds inferred by our tool in this case almost accurately matched the runtime steps usage for this benchmark as discussed in section 5.) Furthermore, for a lazy, bottom-up merge sort implementation [Apfelmus, 2009] one step of evaluation at runtime corresponded to 2.35 nanoseconds (ns) on average with an absolute deviation of 0.01 ns, and for a real-time queue

data structure implementation [Okasaki, 1998] it corresponded to 12.25 ns with an absolute deviation of 0.03 ns. These results further add to the importance of establishing resource bounds even if they are with respect to the algorithmic resource metrics.

**Contracts for Resources** Most existing approaches for analyzing resource usage of programs aim for complete automation but trade off expressive power and the ability to interact with users. Many of these techniques offer little provision for users to specify the bounds they are interested in, or to provide invariants needed to prove bounds of complex computation. For instance, establishing precise resource usage of operations on balanced trees requires the height or weight invariants that ensure balance. As a result, most existing approaches are limited in the programs and resources that they can verify. This is somewhat surprising since resource usage is as hard and as important as proving correctness properties, and the latter is being accomplished with increasing frequency on large-scale, real-world applications such as operating systems and compilers, by using user specifications [Harrison, 2009, Hawblitzel, 2009, Kaufmann et al., 2000, Klein et al., 2009, Leroy, 2009]. This dissertation demonstrates that user-provided contracts are an effective means to make resource verification feasible on complex resources and programs that are well outside the reach of automated techniques. Moreover, it also demonstrates that the advances in SMT-driven verification technology, traditionally restricted to correctness verification, can be fully leveraged to verify resource usage of complex programs.

**Specifying Resource Bounds** Specifying resources using contracts comes with a set of challenges. Firstly, the resources consumed by the programs are not program entities that programmers can refer to. Secondly, the bounds on resources generally involve constants that depend on the implementations, and hence are difficult to estimate by the users. Furthermore, the bounds, and invariants needed to establish the bound, often contain invocations of user-defined recursive functions specific to the program being verified, such as size or height functions on a tree structure.

Our system provides language-level support for a predefined set of algorithmic resources. These resources are exposed to the users through special keywords like steps or alloc. Furthermore, it allows users to specify a desired bound on the predefined resources as templates with numerical holes e.g. as steps $\leq$ ?*size(l) + ? in the contracts of functions along with other invariants necessary for proving the bounds. The templates and invariants are allowed to contain user-defined recursive functions. The goal of the system is to automatically infer values for the holes that will make the bound hold for *all* executions of the function. (Section 2 formally describes the syntax and semantics of the input language.)

**Verifying Resource Specifications** In order to verify such resource templates along with correctness specifications, I developed an algorithm for inferring an assignment for the holes

that will yield a valid resource bound. Moreover, under certain restrictions (such as absence of nonlinearity) the algorithm infers the strongest possible values for the holes and infers the strongest bound feasible for a given template. Specifically, the following are the three main contributions of the inference algorithm. (a) It provides a decision procedure for a fragment of $\exists\forall$ formulas with nonlinearity, uninterpreted functions and algebraic datatypes. (b) It scales to complex formulas whose solutions involved large, unpredicatable constants. (c) It handles highly disjunctive programs with multiple paths and recursive functions. The system was used to verify the worst-case resource usage of purely functional implementations of many complex algorithms including balanced trees and meldable heap data structures. For instance, it was able to establish that the number of steps involved in inserting into a red-black tree implementation provided to the system is bounded by $132\lfloor\log(\mathsf{size}(\mathsf{t}) + 1))\rfloor + 66$. The inference algorithm and the initial results appeared in the publication [Madhavan and Kuncak, 2014], and is detailed in Section 3.

**Memoization and Lazy Evaluation** Another challenging feature supported by our system are first-class functions that rely on (built-in) memoization and lazy evaluation. (Memoization refers to caching of outputs of a function for each distinct input encountered during an execution, and lazy evaluation means the usual combination of call-by-name and memoization supported by languages like Haskell and Scala.) These features are quite important. From a theoretical perspective, it was shown by Bird et al. [1997] that these features make the language strictly more efficient, in asymptotic terms, than eager evaluation. From a practical perspective, they improve the running time (as well as other resource usage) of functional programs sometimes by orders of magnitude. For instance, the entire class of dynamic programming algorithms is built around the notion of memoizing recursive functions. These features have been exploited to design some of most practically efficient, functional data structures known [Okasaki, 1998], and often find built-in support in language runtimes or libraries in various forms e.g. Scala's *lazy vals* and stream library, C#'s LINQ library.

However, in many cases, it has been notoriously difficult to make precise theoretical analysis of the running time of programs that uses lazy evaluation or memoization. In fact, precise running time bounds remain open in some cases (e.g. *lazy pairing heaps* described in page 79 of Okasaki [1998]). Some examples illustrative of this complexity are the *Conqueue* data structure [Prokopec and Odersky, 2015] used to implement Scala's data parallel operations, and Okasaki's persistent queues [Okasaki, 1998] that run in worst-case constant time. The challenge that arises with these features is that reasoning about resources like running time and memory usage becomes state-dependent and more complex than correctness. Nonetheless, they preserve the functional model (referential transparency) for the purpose of reasoning about the result of the computation making them more attractive and amenable to functional verification in comparison to imperative programming models.

**Resource Verification with Memoization** In this dissertation, I also show that the user-driven, contract-based approach can be effective in verifying complex resource bounds in this challenging domain: higher-order functional programs that rely on memoization and lazy evaluation. From a technical perspective, verifying resource usage with these features present unique challenges that are outside the purview of existing verifiers. For instance, consider a function take that returns the first $n$ elements of a stream. If accessing the tail of the stream takes $O(n)$ time then accessing $n$ elements would take $O(n^2)$ time. However, invoking the function take twice or more on the same list would make every call except the first run in $O(n)$ time due to the memoization of the tail of the stream. (Figure 1.3 presents a concrete example.)

Verifying such programs require invariants that depend on the state of the memoization table. Also in some cases, it is necessary to reason about aliasing of references to higher-order functions. This dissertation presents new specification constructs that allow users to specify such properties succinctly. It presents a multi-staged approach for verifying the specifications that gradually encodes the input program and specifications into $\exists\forall$ formulas (VCs) that use only theories efficiently decidable by state-of-the-art SMT solvers. The resulting formulas i.e, VCs are solved using the inference algorithm discussed in the previous paragraph. The encoding was carefully designed so that it does not introduce any abstraction by itself. This meant that users can help the system with more specifications until the desired bounds are established, which adheres with the philosophy of verification. The main technical contributions of this approach are: (a) development of novel specification constructs that allow users to express properties on the state of the memoization table in the contracts and also specify the behavior of first-class functions passed as parameters, (b) design of a new modular, assume-guarantee reasoning for verifying state-dependent contracts in the presence of higher-order functions. This approach and related results appeared in a prior publication: [Madhavan et al., 2017], and is detailed Section 4.

**Evaluation and Results** The approach presented in this dissertation is implemented within the open-source LEON verification and synthesis framework [Blanc et al., 2013]. The implementation is free and open source and available at https://github.com/epfl-lara/leon. The implementation was used to infer precise resource usage of complex functional data structures – balanced trees, heaps and lazy queues, as well as program transformations, static analyses, parsers and dynamic programming algorithms. Some of these benchmarks have never been formally verified before even with interactive theorem provers.

Furthermore, through rigorous empirical evaluation, the precision of the constants inferred by our tool was compared to those obtained by running the benchmarks on concrete inputs (for the resources steps and alloc). Our results confirmed that the bounds inferred by the tool were sound over-approximations of the runtime resource usage, and showed that the worst-case resource usage was, on average, at least 80% of the value inferred by the tool when estimating the number of evaluation steps, and is 88% for the number of heap-allocated objects. For

```
1  import leon.instrumentation._
2  import leon.invariant._
3  object ListOperations {
4    sealed abstract class List
5    case class Cons(head: BigInt, tail: List) extends List
6    case class Nil() extends List
7
8    def size(l: List): BigInt = l match {
9      case Nil() ⇒ 0
10     case Cons(_, t) ⇒ 1 + size(t)
11   }
12
13   def append(l1: List, l2: List): List = (l1 match {
14     case Nil() ⇒ l2
15     case Cons(x, xs) ⇒ Cons(x, append(xs, l2))
16
17   }) ensuring (res ⇒ size(res) == size(l1) + size(l2) && steps ≤ ? *size(l1) + ?)
18
19   def reverse(l: List): List = {
20     l match {
21       case Nil() ⇒ l
22       case Cons(hd, tl) ⇒ append(reverse(tl), Cons(hd, Nil()))
23     }
24   } ensuring (res ⇒ size(res) == size(l) && steps ≤ ? *(size(l)*size(l)) + ?)
25 }
```

Figure 1.2 – Illustration of verifying resource bounds using contracts

instance, our system was able to infer that the number of steps spent in accessing the $k^{th}$ element of an unsorted list $l$ using a lazy, bottom-up merge sort algorithm [Apfelmus, 2009] is bounded by $36(k \cdot \lfloor \log(l.size) \rfloor) + 53l.size + 22$. The number of steps used by this program at runtime was compared against the bound inferred by our tool by varying the size of the list $l$ from 10 to 10K and $k$ from 1 to 100. The results showed that the inferred values were 90% accurate for this example. To the best of my knowledge, our tool is the first available system that can establish such complex resource bounds with this degree of automation.

## 1.1 Overview of the Specification Approach

In this section, I provide a brief overview of how to express programs and specifications in our system using pedagogical examples, and summarize the verification approach. This section is aimed at highlighting the challenges involved in verifying the resource usage of programs that are considered in this dissertation. It also provides an overview of a few specification constructs supported by our system, which will be formally introduced in the later chapters.

Consider the Scala program shown in Figure 1.2 that defines a list as a recursive datatype and defines three operations on it. This example is aimed at highlighting the deep inter-

relationships between verifying correctness properties and resource bounds. The function size computes the size of the list, the function append concatenates a list l2 to a list l1, and the function reverse reverses the list by invoking append and itself recursively. Consider the function reverse. The resource template shown in the postcondition of reverse specifies that the number of steps performed by this function is quadratic in the size of the list. The goal is to infer a bound that satisfies this template.

Intuitively, the reason for this quadratic complexity is because the call to append that happens at every recursive step of reverse takes time linear in the size of the argument passed to it: tl (which equals l.tail). To establish this we need two facts: (a) the function append takes time that is linear in the size of its first formal parameter. (b) The size of the list returned by reverse is equal to the size of the input list, since append is invoked on the list returned by the recursive call to reverse. Therefore, we have the predicate: size(res)== size(l) in the postcondition of reverse. However, in order to establish this, we also need to know the size of the list returned by append in terms of the sizes of its inputs. This necessitates a postcondition for append which asserts that the size of the list returned by append is equal to sum of the sizes of the input lists. Thus, to verify the steps bound, one requires *all* the invariants specified in the program. This example also illustrates the need for an expressive contract language, since even for this small program we need all the invariants shown in the figure to verify its resource bounds.

A major feature offered by our system is that it allows seamless combination of such user-defined invariants with resource templates. The invariants are utilized during the verification of resource bounds to verify the bound and also infer precise values for the constants. The system inferred the bound $11\text{size}(l)^2 + 2$ for the function reverse.

### 1.1.1 Prime Stream Example

In this section, I illustrate specification and verification of programs with higher-order features and lazy evaluation using the pedagogical example shown in Figure 1.3 that creates an infinite stream of prime numbers. The example also illustrates some of the novel specification constructs that are supported by our system for proving precise bounds of such programs.

The class SCons shown in Figure 1.3 defines a stream that stores a pair of unbounded integer (BigInt) and boolean, and has a generator for the tail: tfun which is a function from Unit to SCons. The lazy field tail of SCons evaluates tfun() when accessed the first time and caches the result for reuse. The program defines a stream primes that lazily computes for all natural numbers starting from 1 its primality Notice that the second argument of the SCons assigned to primes is a lambda term (anonymous function) that calls nextElem(2), which when invoked creates a new stream that applies nextElem to the next natural number and so on. The function isPrimeNum($n$) tests the primality of $n$ by checking if any number greater than 1 and smaller than $n$ divides $n$ using an inner function rec. The number of steps it takes is linear in $n$.

The function primesUntil returns all prime numbers until the parameter $n$ using a helper

```
1  private case class SCons(x: (BigInt,Bool), tfun:() ⇒ SCons) {
2    lazy val tail = tfun()
3  }
4  private val primes = SCons((1, true), () ⇒ nextElem(2))
5
6  def nextElem(i: BigInt): SCons = {
7    require(i ≥ 2)
8    val x = (i, isPrimeNum(i))
9    val y = i + 1
10   SCons(x, () ⇒ nextElem(y))
11 } ensuring(r ⇒ steps ≤ ? ∗ i + ?)
12
13 def isPrimeNum(n: BigInt): Bool = {
14   def rec(i: BigInt): Bool = {
15     require(i ≥ 1 && i < n)
16     if (i == 1)
17       true
18     else
19       (n % i != 0) && rec(i − 1)
20   } ensuring (r ⇒ steps ≤ ? ∗ i + ?)
21   rec(n − 1)
22 } ensuring(r ⇒ steps ≤ ? ∗ n + ?)
23
24 def isPrimeStream(s: SCons, i: BigInt): Bool = {
25   require(i ≥ 2)
26   s.tfun ≈ (() ⇒ nextElem(i))
27 }
28
29 def takePrimes(i: BigInt, n: BigInt, s: SCons): List = {
30  require(0 ≤ i && i ≤ n && isPrimeStream(s, i+2))
31   if(i < n) {
32     val t = takePrimes(i+1, n, s.tail)
33     if(s.x._2)
34       Cons(s.x._1, t)
35     else
36       t
37   } else Nil()
38 } ensuring(r ⇒ steps ≤ ? ∗ (n(n−i)) + ?)
39
40 def primesUntil(n: BigInt): List = {
41   require(n ≥ 2)
42   takePrimes(0, n−2, primes)
43 } ensuring(r ⇒ steps ≤ ? ∗ n^2 + ?)
```

Figure 1.3 – Prime numbers until $n$ using an infinite stream.

function takePrimes, which recursively calls itself as long as $i < n$ on the tail of the input stream (line 32), incrementing the index $i$. Consider now the running time of this function. If takePrimes is given an arbitrary stream s, its running time *cannot* be bounded since accessing the field tail at line 32 could take any amount of time. Therefore, we need to know the resource

```
1  def concrUntil(s: SCons, i: BigInt): Bool =
2    if(i > 0)
3      cached(s.tail) && concrUntil(s.tail, i−1)
4    else true
5
6  def primesUntil(n: BigInt): List = {
7
8    // see Figure 1.3 for the code of the body
9
10 } ensuring{r ⇒ concrUntil(primes, n−2) &&
11   (if(concrUntil(primes, n−2) in inSt)
12     steps ≤ ? ∗ n + ?
13    else steps ≤ ? ∗ n^2 + ?) }
```

Figure 1.4 – Specifying properties dependent on memoization state.

usage of the closures accessed by takePrimes, namely s.(tail)*.tfun. However, we expect that the stream s passed to takePrimes is a suffix of the primes stream, which means that tfun is a closure of nextElem. To allow expressing such properties our system reintroduces the notion of *intensional or structural* equivalence, denoted ≈, between closures [Appel, 1996].

**Structural Equality as a means of Specification** In our system, closures are allowed to be compared structurally. Two closures are structurally equal iff their abstract syntax trees are identical without unfolding named functions. This equivalence is formally defined in section 2.3. For example, the comparison at line 27 of Figure 1.3 returns true *iff* the tfun parameter of s is a closure that invokes nextElem on an argument that is equal to i. This equality is found to be an effective and low-overhead means of specification for the following reasons.

(a) Many interesting data structures based on lazy evaluation use aliased references to closures (e.g. Okasaki's scheduling-based data structures [Okasaki, 1998, Prokopec and Odersky, 2015] discussed in section 5.2). Expressing invariants of such data structures requires equating closures. While reference equality is too restrictive for convenient specification (and also breaks referential transparency), semantic or extensional equality between closures is undecidable, and hence introduces high specification/verification burden. Structural equality is well suited in this case.

(b) Our approach is aimed at (but not restricted to) callee-closed programs where the targets of all indirect calls are available at analysis time. (Section 2.3 formally describes such programs.) In such cases, it is often convenient and desirable to state that a closure has the same behavior as a function in the program, as was required in Figure 1.3.

(c) Structural equality also allows modeling reference equality of closures by augmenting closures with unique identifiers as they are created in the program.

While structural equality is a well-studied notion [Appel, 1996], to my knowledge, no prior

work uses it as a means of specification. Using structural equality, it can be specifed that the stream passed as input to takePrimes is an SCons whose tfun parameter invokes nextElem(i+2) (see function isPrimeStream and the precondition of takePrimes). This allows the system to bound the steps, which denotes the number of primitive evaluation steps, of the function takePrimes to $O(n(n-i))$ and that of primesUntil to $O(n^2)$. For primesUntil, our tool inferred that steps $\leq 16n^2 + 28$.

**Properties Dependent on Memoization State.** The quadratic bound of primesUntil is precise only when the function is called for the first time. If primesUntil(n) is called twice, the time taken by the second call would be linear in $n$, since every access to tail within takePrimes will take constant time as it has been cached during the previous call to takePrimes. The time behavior of the function depends on the state of the memoization table (or cache) making the reasoning about resources imperative.

To specify such properties the system supports a built-in operation cached(f(x)) that can query the state of the cache. This predicate holds if the function f is a memoized function and is cached for the value x. Note that it does not invoke f(x). The function concrUntil(s, i) shown in Figure 1.4 uses this predicate to state a property that holds iff the first i calls to the tail field of the stream s have been cached. (Accessing the lazy field s.tail is similar to calling a memoized function tail(s).) This property holds for primes stream at the end of a call to primesUntil(n), and hence is stated in the postcondition of primesUntil(n) (line 10 of Figure 1.4). Moreover, if this property holds in the state of the cache at the beginning of the function, the number of steps executed by the function would be linear in n. This is expressed using a disjunctive resource bound (line 11).

Observe that in the postcondition of the function, one need to refer to the state of the cache at the beginning of the function, as it changes during the execution of the function. For this purpose, our system supports a built-in construct “inSt" that can be used in the postcondition to refer to the state at the beginning of the function, and an “in" construct which can be used to evaluate an expression in the given state. These expressions are meant only for use in contracts. These constructs are required since the cache is implicit and cannot be directly accessed by the programmers to specify properties on it. On the upside, the knowledge that the state behaves like a cache is exploited by the system to reason functionally about the result of the functions, which results in fewer contracts and more efficient verification.

**Verification Strategy.** Our approach, through a series of transformations, reduces the problem of resource bound inference for programs like the one shown in Figure 1.3 to invariant inference for a strict, functional first-order program. It solves it by applying an inductive, assume-guarantee reasoning. The inductive reasoning exploits and uses the monotonic evolution of cache, and the properties that are monotonic with respect to the changes to the cache.

The inductive reasoning works on the assumption that the expressions in the input program *terminate*, which is verified independently using an existing termination checker. Our system uses the Leon termination checker for this purpose [Nicolas Voirol and Kuncak, 2017], but other termination algorithms for higher-order programs [Giesl et al., 2011, Jones and Bohr, 2004, Sereni, 2006] are also equally applicable. Note that memoization only affects resource usage and not termination, and lazy suspensions are in fact lambdas with unit parameters. This strategy of decoupling termination checks from resource verification enables checking termination using simpler reasoning, and then use proven well-founded relations during resource analysis. This allows us to use recursive functions for expressing resource bounds and invariants, and enables modular, assume-guarantee reasoning that relies on induction over recursive calls (previously used in correctness verification) to establish resource bounds. This aspect is discussed in more detail in section 3.2.

## 1.2 Summary of Contributions

In summary, the following are the major contributions of this dissertation:

**I**. I propose a specification approach for expressing resource bounds of programs and the necessary invariants in the presence of recursive functions, recursive datatypes, first-class functions, memoization and lazy evaluation.

- The approach allows specifying bounds as templates with numerical holes in the post-conditions of functions, which are automatically solved by the system (section 2).
- The specifications can use structural-equality-based constructs for specifying properties of higher-order functions (section 2).
- The specification can assert properties on the state of the memoization table (section 3).

**II**. I present a system for verifying the contracts of programs expressed in our language by designing new algorithms and extending existing techniques from contract-based correctness verification.

- I present a novel inference algorithm for solving $\exists\forall$ formulas with recursive functions, inductive datatypes and nonlinearity such as multiplication of two unknown variables. I prove that the algorithm is sound, always terminates, and also is complete under certain restrictions (section 3.3).
- I present an encoding of higher-order functions with memoization as first-order programs with recursive functions, dataypes and sets, and establish its soundness and completeness (section 4.4).
- I present an assume-guarantee reasoning for higher-order functions with memoization,

which exploits properties that monotonically evolve with respect to the changes in the cache. I establish the soundness of this reasoning (section 4.6).

**III**. I present the results of using the system to establish precise resource bounds of 50 benchmarks, comprising 8K lines of functional Scala code, implementing complex data structures and algorithms that are outside the reach of existing approaches. The experimental evaluations show that while the inferred values are over-estimated by the runtime values, the runtime values are 80% of the value inferred by the tool when averaged over all benchmarks (section 5).

## 1.3 Outline of the Thesis

The rest of the dissertation is organized as follows:

- Chapter 2 describes the core syntax and semantics of the input language and specifications. It formalizes the semantics of the resources supported by our system. It defines the problem of resource and contract verification, and formally establishes several properties of the core language.
- Chapter 3 describes the algorithm for inferring resource bounds for first-order programs without higher-order features and memoization. It details the resource instrumentation performed by our system, the modular assume-guarantee reasoning used by system, and the inference algorithm for inferring holes.
- Chapter 4 describes the extensions to the algorithm for inferring resource bounds of programs with first-class functions and memoization. It formally describes the semantics of memoization and related specification constructs. It formally presents and proves the verification approach that can handle programs with these features.
- Chapter 5 presents the results of evaluation of the system on the benchmarks using summary statistics and graphical plots. It also presents the fully verified implementation of the real-time queue data structure.
- Chapter 6 discusses the works related to the topic of this dissertation.

# 2 Semantics of Programs, Resources and Contracts

> The purpose of abstraction is not to be vague, but
> to create a new semantic level in which one
> can be absolutely precise.
> — Edsger W. Dijkstra

In this chapter, I formally present and discuss the core syntax and semantics of the programs accepted by our system and eventually define the problem of resource verification. As a first step, I introduce a core language that captures the relevant syntactic aspects of the input programs. Specifically, the core language supports recursive functions, recursive datatypes, contracts and resource templates. For the sake of succinctness and reducing notational overhead, for certain constructs of the core language I adopt the syntax of lambda calculus instead of following the the syntax of Scala. For instance, anonymous functions are denoted as $\lambda$ terms and variable declarations are replaced by “let" binders. Nonetheless, the constructs of the language have a straightforward translation to Scala. The syntax description of the language also includes higher-order constructs, lazy evaluation and memoization, and specification constructs meant for use with these features. However since these features are quite involved and are orthogonal to the definition of the problem, in this chapter I will focus mostly on first-order constructs and defer the discussion of the semantics of other constructs to later chapters.

The semantics I present here is a big-step, operational semantics that has two unorthodox features. Firstly, the semantics not only characterizes the state changes induced by the language constructs but also characterizes their resource usage. To succinctly formalize usage of multiple resources supported by our system, the semantics is parameterized by “cost" functions. These cost functions capture resource-specific parameters and are independently (re)defined for each resource that is of interest. I also present the definition of these cost functions for the important resources supported by our system: steps, alloc, stack and depth.

The second unorthodox feature of the semantics is that it assigns an operational meaning to contracts and specification constructs. Thus contracts in our system are expressions of the

language and, in principle, are executable on an appropriate runtime that can implement their semantics. This naturally allows contracts to use and manipulate the same entities used by the rest of the program. For instance, the variables, data structures and functions declared in the program are automatically available to the predicates in the contracts without restriction.

In the final sections of this chapter, I define the problem of contract and resource verification for open programs (or libraries) and define notions like encapsulation using the operational semantics of the constructs of the language. These definitions are used to establish the soundness of our system in the later chapters (Chapters 3 and 4).

## 2.1 Syntax of the Core Language

Figure 2.1 show the syntax of the core functional language describing the syntax of the input programs. $E_{src}$ shows the syntax of the expressions that can be used in the implementation. They consists of variables *Vars*, constants *Cst*, primitive operations on integers and booleans *Prim*, a structural equality operator eq, let expressions, match expressions, lambda terms, direct calls to named functions: $f\ x$, and indirect calls or lambda applications: $x\ y$. The rule $Block_\alpha$ is parameterized by the subscript $\alpha$ and defines the let, match and if-then-else combinators that operate over a base expression $e_\alpha$. The integers in our language are unbounded big integers. They correspond to the BigInts of Scala. *Tdef* shows the syntax of user-defined recursive datatypes and *Fdef* shows the syntax of function definitions. The functions are classified into source functions $Fdef_{src}$, which are considered as implementations, and specification functions $Fdef_{spec}$, which can be used only in the specifications (explained shortly). As a syntactic sugar, tuples are considered as a special datatype. Tuple constructions are denoted using $(x_1, \cdots, x_n)$, and selecting the $i^{th}$ element of a tuple is denoted using $x.{}_i$.

For ease of formalization, the language incorporates the following syntactic restrictions without reducing generality. Most expressions except lambda terms are expressed in *A-normal form* i.e, the arguments of the operations performed by the expressions are variables. The conditional expressions such as if-then-else and match constructs are an exception, since the expressions along the branches need to be executed only when the corresponding guards are true. All lambdas are of the form: $\lambda x.f\ (x, y)$ where $f$ is a named function whose argument is a pair (a two element tuple) and $y$ is a captured variable. Note that this lifting of bodies of lambda terms to named functions is a simple syntactic refactoring which does not limit the expressiveness of the language.

Every expression belonging to our language has a *static label* belonging to *Labels* (not shown in Figure 2.1). For instance, the label of an expression $e$ could a combination of the name of the source file that contains the expression $e$ and the position of $e$ in the source file. Let $e^\ell$ denotes an expression with its label. To reduce clutter, the labels are omitted if it is not relevant to the context. A program $P$ is a set of functions definitions in which every function identifier is unique, every direct call invokes a function defined in the program, and the labels of all expressions are unique.

$$
\begin{array}{lll}
 & x, y \in \mathit{Vars} & \textit{(Variables)} \\
 & c \in \mathit{Cst} & \textit{(Variables \& Constants)} \\
 & f \in \mathit{Fids} & \textit{(Function Identifiers)} \\
 & d \in \mathit{Dids} & \textit{(Datatype identifiers)} \\
 & C_i \in \mathit{Cids}, i \in \mathbb{N} & \textit{(Constructor Identifiers)} \\
 & a \in \mathit{TVars} & \textit{(Template Variables)} \\
 & \bar{x} \in \mathit{Vars}^* & \textit{(Sequence of Variables)} \\
 & \bar{\tau} \in \mathit{Vars}^* & \textit{(Sequence of Types)} \\
\mathit{Tdef} & ::= & \textbf{type}\ d := (C_1\ \bar{\tau}, \cdots, C_n\ \bar{\tau}) \\
\tau \in \mathit{Type} & ::= & \mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \tau \Rightarrow \tau \mid d \\
\mathit{Block}_\alpha & ::= & \textbf{let}\ x := e_\alpha\ \mathsf{in}\ e_\alpha \mid x\ \textbf{match}\{(C\ \bar{x} \Rightarrow e_\alpha;)^+\} \mid \textbf{if}\ (x)\ e_\alpha\ \textbf{else}\ e_\alpha \\
pr \in \mathit{Prim} & ::= & + \mid - \mid * \mid \cdots \mid \wedge \mid \neg \\
e_s \in E_{src} & ::= & x \mid c \mid pr\ x \mid x\ \mathsf{eq}\ y \mid f\ x \mid C\ \bar{x} \mid e_\lambda \mid x\ y \mid \mathit{Block}_s \\
e_\lambda \in \mathit{Lam} & ::= & \lambda x.f\ (x, y) \\
e_p \in E_{spec} & ::= & e_s \mid \mathit{Block}_p \mid (f_p\ x) \mid \mathsf{res} \mid \mathit{resource} \leq \mathit{ub} \mid E_{mem} \\
 & & \mid x\ \mathsf{fmatch}\ \{(e_\lambda \Rightarrow e_p;)^+\} \\
E_{mem} & ::= & \mathsf{cached}(f\ x) \mid \mathsf{inSt} \mid \mathsf{outSt} \mid \mathsf{in}(e_p, x) \mid {e_p}^\star \\
\mathit{resource} & ::= & \mathsf{steps} \mid \mathsf{alloc} \mid \mathsf{stack} \mid \mathsf{depth} \\
\mathit{ub} \in \mathit{Bound} & ::= & e_p \mid e_t \\
e_t \in E_{tmp} & ::= & a \cdot x + e_t \mid a \\
\mathit{Fdef}_{src} & ::= & (@\mathsf{memoize})?\ \textbf{def}\ f_s\ x := \{e_p\}\ e_s\ \{e_p\} \\
\mathit{Fdef}_{spec} & ::= & \textbf{def}\ f_p\ x := \{e_p\}\ e_p\ \{e_p\} \\
\mathit{Fdef} & ::= & \mathit{Fdef}_{src} \cup \mathit{Fdef}_{spec} \\
\mathit{Program} & ::= & 2^{(\mathit{Tdef} \cup \mathit{Fdef})}
\end{array}
$$

Figure 2.1 – Syntax of types, expressions, functions, and programs

The annotation @memoize serves to mark functions that have to be memoized. Such functions are evaluated exactly once for each distinct input passed to them at run time. Notice that only source functions are allowed to be memoized. The language as such uses call-by-value evaluation strategy. But this annotation allows the language to simulate call-by-need or lazy evaluation strategy. This feature is discussed in detail in section 4.

Expressions that are bodies of functions can have contracts (also called specifications). Such expressions have the form $\{e_1\}\ e\ \{e_2\}$ where $e_1$ and $e_2$ are the pre-and post-conditions of $e$ respectively. These conditions can use constructs that are not available to the source expressions. In other words, their syntax given by $E_{spec}$ permits more constructs than $E_{src}$. In particular, the postcondition of an expression $e$ can refer to the result of $e$ using the variable res, and can

refer to the resource usage of $e$ using the keywords steps, alloc or depth. Users can specify upper bounds on resources as templates with holes as defined by $e_t \in E_{tmp}$. The holes always appear as coefficients of variables defined or visible in the postconditions. The variables could be bound to more complex expressions through **let** binders. We enforce that the holes are distinct across function definitions. The specification constructs fmatch and those given by $E_{mem}$ are meant for specifying the behavior of first-class functions and the behavior of expressions under memoization, respectively. I will not focus on these constructs here and will explain them in detail in Chapter 4. Before I discuss the formal semantics of the language, I present a few basic notation and terminology used in the rest of the sections.

## 2.2 Notation and Terminology

**Partial Functions** Given a domain $A$, $\bar{a} \in A^*$ denotes a sequence of elements in $A$, and $a_i$ refers to the $i^{th}$ element. Note that this is different from tuple selector $x.i$, which is an expression of the language. The notation $A \rightarrowtail B$ denotes a partial function from $A$ to $B$. Given a partial function $h$, $\hat{h}(\bar{x})$ denotes the function that applies $h$ point-wise on each element of $\bar{x}$. $h[a \rightarrowtail b]$ denotes the function that maps $a$ to $b$ and every other value $x$ in the domain of $h$ to $h(x)$. The notation $h[\bar{a} \rightarrowtail \bar{b}]$ denotes $h[a_1 \rightarrowtail b_1] \cdots [a_n \rightarrowtail b_n]$. The function $h$ is omitted in the above notation if it is an empty function. Let $h_1 \uplus h_2$ be defined as $(h_1 \uplus h_2)(x) =$ **if** $(x \in dom(h_2))$ $h_2(x)$ **else** $h_1(x)$. Let $h_1 \sqsubseteq h_2$ iff the function $h_2$ includes all binding of $h_1$ i.e, $\forall a \in dom(h_1).h_1(a) = h_2(a)$. A closed integer interval from $a$ to $b$ is denoted using $[a,b]$.

**Expression Operations** Let $labels_P$ denote the set of labels of all expressions in a program $P$. Let $type_P(e)$ denote the type of an expression $e$ in a program $P$. Given an expression $e$, let $FV(e)$ denote the set of free variables of $e$. Expressions without free variables are referred to as closed expressions. Given a lambda term $\lambda x.f\ (x,y)$, $y$ is called the *captured variable* of the lambda term. Note that $FV(e_\lambda)$ is a singleton set containing the captured variable. $target(e_\lambda)$ denotes the function called in the body of the lambda. (Recall that the body of every lambda term is call to a named function. ) The operation $e[e'/x]$ denotes the syntactic replacement of the free occurrences of $x$ in $e$ by $e'$. This operation replaces expressions along with their static labels and also performs alpha-renaming of bound variables, if necessary, to avoid variable capturing. A substitution $\varsigma : Vars \rightarrowtail Expr$ is a partial function from variables to expressions. Let $e\,\varsigma$ denote $e[\varsigma(x_1)/x_1] \cdots [\varsigma(x_n)/x_n]$, where $dom(\varsigma) = \{x_1, \cdots x_n\}$. Given a substitution $\iota : TVars \rightarrowtail \mathbb{Z}$, let $e\,\iota$ represent the substitution of the holes by the values given by the assignment. Similarly, let $P\,\iota$ denote the program obtained by replacing the every hole $a$ in the bodies of functions in $P$ by $\iota(a)$. This notation is also extended to formulas later. Programs and expressions without holes as referred to as *concrete* programs and expressions. Let $body_P(f)$ and $param_P(f)$ denote the body and parameter of a function $f$ defined in a program $P$

$$\textsc{Cst}\ \frac{c \in Cst}{\Gamma \vdash c \underset{c_{cst}}{\Downarrow} c, \Gamma} \qquad \textsc{Var}\ \frac{x \in Vars}{\Gamma : (H,\sigma) \vdash x \underset{c_{var}}{\Downarrow} \sigma(x), \Gamma} \qquad \textsc{Prim}\ \frac{pr \in Prim}{\Gamma \vdash pr\ x \underset{c_{pr}}{\Downarrow} pr(\sigma(x)), \Gamma}$$

$$\textsc{Equal}\ \frac{v = \sigma(x) \underset{H}{\approx} \sigma(y)}{\Gamma : (H,\sigma) \vdash x \,\mathsf{eq}\, y \underset{c_{eq}}{\Downarrow} v, \Gamma} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 \Downarrow_p v_1, (H',\sigma') \qquad (H', \sigma[x \mapsto v_1]) \vdash e_2 \Downarrow_q v_2, (H'',\sigma'')}{\Gamma : (H,\sigma) \vdash \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \underset{c_{let} \oplus p \oplus q}{\Downarrow} v_2, (H'',\sigma)}$$

$$\textsc{Lambda}\ \frac{a = fresh(H) \qquad clo = (\lambda x.f\ (x,y), [y \mapsto \sigma(y)])}{\Gamma : (H,\sigma) \vdash \lambda x.f\ (x,y) \underset{C_\lambda}{\Downarrow} a, (H[a \mapsto clo], \sigma)} \qquad \textsc{Cons}\ \frac{a = fresh(H) \qquad H' = H[a \mapsto (cons\ \hat{\sigma}(\bar{x}))]}{(H,\sigma) \vdash cons\ \bar{x} \underset{c_{cons}}{\Downarrow} a, (H',\sigma)}$$

$$\textsc{Match}\ \frac{H(\sigma(x)) = C_i\ \bar{v} \qquad (H, \sigma[\bar{x}_i \mapsto \bar{v}]) \vdash e_i \Downarrow_q v, (H',\sigma')}{\Gamma : (H,\sigma) \vdash x\ \mathsf{match}\ \{C_i\ \bar{x}_i \Rightarrow e_i)\}_{i=1}^{n} \underset{c_{match(i)} \oplus q}{\Downarrow} v, (H',\sigma)}$$

$$\textsc{If}\ \frac{\Gamma \vdash e_i \Downarrow_q v, \Gamma'}{\Gamma \vdash \mathsf{if}\ (\mathsf{x})\ e_1\ \mathsf{else}\ e_2 \underset{c_{if} \oplus q}{\Downarrow} v, \Gamma'} \quad \text{where } i = \begin{cases} 1 & \sigma(x) = true \\ 0 & \sigma(x) = false \end{cases}$$

$$\textsc{ConcreteCall}\ \frac{(H, \sigma[param_P(f) \mapsto u]) \vdash body_P(f) \Downarrow_p v, (H',\sigma')}{\Gamma : (H,\sigma) \vdash f\ u \Downarrow_p v, (H',\sigma)} \qquad \textsc{DirectCall}\ \frac{f \in Fids \qquad \Gamma \vdash (f\ \sigma(x)) \Downarrow_p v, \Gamma'}{\Gamma \vdash f\ x \underset{c_{call} \oplus p}{\Downarrow} v, \Gamma'}$$

$$\textsc{App}\ \frac{H(\sigma(x)) = (\lambda z.e, \sigma') \qquad (H, (\sigma \uplus \sigma')[z \mapsto \sigma(y)]) \vdash e \Downarrow_p v, (H',\sigma')}{\Gamma : (H,\sigma) \vdash x\ y \underset{c_{app} \oplus p}{\Downarrow} v, (H',\sigma)}$$

$$\textsc{Contract}\ \frac{\Gamma \vdash pre \Downarrow_p true, \Gamma_1 \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (H_2,\sigma_2) \qquad (H_2, \sigma_2[R \mapsto q, \mathsf{res} \mapsto v]) \vdash post \Downarrow_r true, \Gamma_3}{\Gamma \vdash \{pre\}\ e\ \{post\} \Downarrow_q v, \Gamma_2}$$

where $R \in \{\mathsf{steps}, \mathsf{alloc}, \mathsf{stack}, \mathsf{depth}\}$

---

Cost function definition for steps:

$$\begin{aligned} c_{match(i)} &= i+1 \\ c_{var} = c_{let} &= 0 \\ c_{op} &= 1 \quad \text{for every other operation } op \\ \oplus &= + \end{aligned}$$

Cost function definition for alloc:

$$\begin{aligned} c_{cons} = c_\lambda &= 1 \\ c_{op} &= 0 \quad \text{for every other operation } op \\ \oplus &= + \end{aligned}$$

Figure 2.2 – Resource-annotated operational semantics of the core language

## 2.3 Resource-Annotated Operational Semantics

Figure 2.2 defines the operational semantics for the core language that also captures the resource consumption of expressions of the language. The semantics is defined only for concrete expressions without holes. (Expressions with holes are not executable.) The semantics is big-step which is well suited for formalizing a compositional approach such as ours. Unfortunately, big-step semantics is not convenient for reasoning about termination or reachability of states at a specific expression (or program point), both of which are necessary for our purposes. Therefore, I define a reachability relation on top of the big-step semantics, similar to the *calls* relation of Sereni, Jones and Bohr [Jones and Bohr, 2004, Sereni, 2006]. This reachability relation acts similar (but not identical) to a small-step semantics.

### 2.3.1 Semantic Domains

The semantics rules presented in Figure 2.2 operate over the semantics domain described below. Let *Adr* denote the addresses of heap-allocated structures namely closures and datatypes. Let *DVal* denote the set of datatype instances, *Clo* the set of closures. Let $H$ be a partial function from addresses to datatypes instances or lambdas and a store $\sigma$ a partial function from variables to values. The state of an interpreter evaluating expressions of our language, referred to as the evaluation environment $\Gamma$, is a triple consisting a heap $H$, a store $\sigma$, and a program, which is a set of function definitions. Formally,

$$
\begin{aligned}
&u, v \in Val = \mathbb{Z} \cup Bool \cup Adr \\
&DVal = Cids \times Val^* \\
&Clo = Lam \times Store \\
&H \in Heap = Adr \rightarrowtail (DVal \cup Clo) \\
&\sigma \in Store = Vars \rightarrowtail Val \\
&\Gamma \in Env \subseteq Heap \times Store \times Program
\end{aligned}
$$

Every environment should also satisfy the following domain invariants, which are certain sanity conditions that are ensured by the operation semantics.

**Def 1** (**Domain Invariants**)**.** *A triple* $(H, \sigma, P)$ *is an environment iff the following properties hold.*

*(a)* $(range(\sigma) \cap Adr) \subseteq dom(H)$
*(b)* $x \in dom(\sigma)$ *implies that* $\sigma(x)$ *inhabits* $type_P(x)$
*(c)* $a \in dom(H)$ *implies that* $H(a)$ *inhabits* $type_P(a)$
*(d)* $H$ *is a acyclic heap*
*(e) For all closures* $(\lambda x.\ f\ (x, y), \sigma') \in range(H)$, $f$ *is defined in* $P$ *and* $y \in dom(\sigma')$

In the above definition the type of an address is the same as the type of the constructor or lambda term that it refers to. The invariant (d) requiring the heap to be acyclic is defined more

formally in section 2.7. Let $fresh(H)$ be a function that picks an address that is not bound in the heap $H$. That is $fresh(H) \in (Adr \setminus dom(H))$. Such a function can be defined deterministically by fixing a well-ordering on the elements of $Adr$ and requiring that $fresh(H)$ always returns the *smallest* address not bound in the heap $H$. That is, $fresh(H) = \min(Adr \setminus dom(H))$.

**Judgements** Let $\Gamma \vdash e \Downarrow_p v, \Gamma'$ be a semantic judgement denoting that under an environment $\Gamma \in Env$, an expression $e$ evaluates to a value $v \in Val$ and results in a new environment $\Gamma' \in Env$, while consuming $p \in \mathbb{Z}$ units of a resource. When necessary $\Gamma$ is expanded as $\Gamma : (H, \sigma, P)$ to highlight the individual components of the environment. Any component of the judgement that is not relevant to the discussion is omitted when there is no ambiguity. In Figure 2.2, the program component is omitted from the environment as it does not change during the evaluation.

### 2.3.2 Resource Parametrization

The operational semantics is parameterized in a way that it can be instantiated on multiple resources using the following two cost functions:

(a) A cost function $c_{op}$ that returns the resource requirement of an operation $op$ such as *cons* or *app*. The operation $c_{op}$ may possibly have parameters. In particular, $c_{match(i)}$ is used to denote the cost of a match operation when the $i^{th}$ case is taken, which should include the cost of failing all the previous cases.

(b) A resource combinator $\oplus : \mathbb{Z}^* \to \mathbb{Z}$ that computes the resource usage of an expression by combining the resource usages of the sub-expressions. Typically, $\oplus$ is either + or max.

The Figure 2.2 shows the definition of the cost functions for resources: (a) the number of steps in the evaluation of an expression denoted steps, and (b) the number of heap-allocated objects created by an expression (viz. a closure or datatype) denoted alloc. For both resources, the resource combinator $\oplus$ is defined as addition (+). In the case of steps, $c_{let}$ and $c_{var}$ are zero as the operations are normally optimized away or subsumed by a machine instruction. The cost of every other operation is 1 except for $c_{match(i)}$. The cost of the match operation ($c_{match(i)}$) is defined proportional to $i$ as the cost of failing all the $i-1$ match cases has to be included. Datatype constructions and primitive operations on integers (which are unbounded big integers in our language) are considered as unitary steps. In the case of alloc, $c_{op}$ is 1 for datatype and closure creations as they involve heap allocations. It is zero for every other operation.

Another resource supported by our tool is the number of (call) stack locations required for holding the local variables and results of function calls during the evaluation of an expression, denoted stack. This resource can also be expressed using the cost functions. A somewhat simplified definition is shown below.

$\oplus = \max$

Cost function definition for stack: $c_{call} = c_{app} = 1 + \#$ of local variables of the callee

$c_{op} = 0$ for every other operation $op$

In the above definition, the number of local variables of the callee is $|FV(body(f))|$ for a direct call $f\ x$ or an application applying a lambda $\lambda x.f\ (x,y)$. Note that when the above definition of $\oplus$ is plugged into the semantics, the stack usage of a let expression: **let** $x := e_1$ in $e_2$ would be computed as the maximum of the stack usages of $e_1$ and $e_2$. This is the expected semantics as the stack space utilized by the calls made during $e_1$ could be reclaimed and reused during the evaluation of $e_2$. The above definition is a much simplified version that hides many complexities. For instance, the number of arguments passed to call and applications are assumed to be one as, in our core language, every function take only one argument – two or more arguments have to be passed (by reference) via a tuple, which is allocated in the heap. Secondly, the actual parameter passing mechanism of a runtime may reserve more space in the call stack more than required for allocating local variables of a function. However, these constant factors (if they are precisely known) can be captured by the cost functions quite easily.

In addition to the above resource, our system also supports a resource depth, which is a measure of algorithmic parallel execution time. However, defining the resource is slightly more involved. The discussion of this resource is deferred to section 3, where the instrumented procedure is described. Our tool also supports a resource rec that counts the number of recursions that appear in the evaluation and is useful for bounding the asymptotic complexity.

### 2.3.3 Structural Equivalence and Simulation

As shown in Figure 2.1 our language supports a equality operator eq for comparing expressions of the language. The semantics shown in Figure 2.2 defines this operation using a structural equivalence relation $\underset{H}{\approx}$ defined on the values *Val* with respect to a $H \in Heap$ (see rule EQUALS). This equivalence is explained and defined below.

Two addresses are structurally equivalent iff they are bounded to structurally equivalent values in the heap. Two datatypes are structurally equivalent iff they use the same constructor and their fields are equivalent. Structural equivalence of closures is similar to syntactic equality of lambdas modulo alpha renaming (but extended to captured variables). Two closures are structurally equivalent iff their lambdas are of the form $\lambda x.f\ (x,y)$ and $\lambda w.f\ (w,z)$ and the captured variables $y$ and $z$ are bound to structurally equivalent values. Formally, $\underset{H}{\approx}$ is defined as the least fix point of the following equations. (The subscript $\underset{H}{\approx}$ is omitted below for clarity).

$$
\begin{array}{rl}
\forall a \in \mathbb{Z} \cup Bool. & a \approx a \\
\forall \{a,b\} \subseteq Adr. & a \approx b \textit{ iff } H(a) \approx H(b) \\
\forall f \in Fids, \{a,b\} \subseteq Val. & (f\ a) \approx (f\ b) \textit{ iff } a \approx b \\
\forall c \in Cids, \{\bar{a},\bar{b}\} \subseteq Val^n. & (c\ \bar{a}) \approx (c\ \bar{b}) \textit{ iff } \forall i \in [1,n].a_i \approx b_i \\
\forall \{e_1,e_2\} \subseteq Lam.\ \forall \{\sigma_1,\sigma_2\} \subseteq Store. & (e_1,\sigma_1) \approx (e_2,\sigma_2) \textit{ iff } target(e_1) = target(e_2) \\
& \quad \wedge \sigma_1(FV(e_1)) \approx \sigma_2(FV(e_2))
\end{array}
$$

For datatypes, this equivalence is similar to value equality supported by languages like Scala and ML, and for closures this equivalence is similar to the traditional notion of *intensional* equality. Although not supported by modern languages since Lisp 1.5, intensional equality proved to be quite a powerful and handy tool in *specifying* resource usage behavior of first-class functions. One of the contributions made by this dissertation is that constructs based on intensional equality of closures offer a referentially transparent yet decidable notion of equality between closures (as opposed to reference or extensional equality) and hence provide great value from the perspective of specifications, especially for proving implementation-dependent properties like *resource consumption*. These aspects are considered in more detail in Chapter 4.

**Structural Simulation Relation** While the above described structural equivalence serves to compared two values under the same $H$, it is often required in the formalism to compare values under different heaps that arise during independent evaluations. For instance, to show a program transformation is equivalent to the original program, one needs to show that they agree on all values, which could be addresses defined under multiple heaps. For these purposes, I define a structural simulation relation $\underset{H_1,H_2}{\approx}$ with respect to two heaps as shown below. This relation is primarily used in the proofs of soundness and completeness of the algorithms. (The subscripts $H_1, H_2$ are omitted in the following definitions for clarity.)

$$
\begin{array}{rl}
\forall a \in \mathbb{Z} \cup Bool. & a \approx a \\
\forall \{a,b\} \subseteq Adr. & a \approx b \textit{ iff } H_1(a) \approx H_2(b) \\
\forall f \in Fids, \{a,b\} \subseteq Val. & (f\ a) \approx (f\ b) \textit{ iff } a \approx b \\
\forall c \in Cids, \{\bar{a},\bar{b}\} \subseteq Val^n. & (c\ \bar{a}) \approx (c\ \bar{b}) \textit{ iff } \forall i \in [1,n].a_i \approx b_i \\
\forall \{e_1,e_2\} \subseteq Lam. \forall \{\sigma_1,\sigma_2\} \subseteq Store. & (e_1,\sigma_1) \approx (e_2,\sigma_2) \textit{ iff } target(e_1) = target(e_2) \\
& \quad \wedge \sigma_1(FV(e_1)) \approx \sigma_2(FV(e_2))
\end{array}
$$

Notice that the only change compared to the definition of structural equivalence ($\underset{H}{\approx}$) is the rule for addresses which now uses different heaps. The following are some properties of structural simulation. (The proof of the following properties are omitted as they are straightforward to derive from the definitions.)

(i) If $H_1 \sqsubseteq H_2$, $\underset{H_1,H_2}{\approx}$ is equivalent to $\underset{H_2}{\approx}$

(ii) $x \underset{H_1,H_2}{\approx} y$ implies $y \underset{H_2,H_1}{\approx} x$ *(Symmetry)*

(iii) $x \underset{H_1,H_2}{\approx} y$ and $y \underset{H_2,H_3}{\approx} z$ implies $x \underset{H_1,H_3}{\approx} y$ *(Transitivity)*

Note that the first property above implies relexivity. The definition of structural equivalence between values can be lifted to stores and environments in a natural way as shown below.

$$\forall \{\sigma_1, \sigma_2\} \subseteq \mathit{Store}.\ \sigma_1 \underset{H_1,H_2}{\approx} \sigma_2 \ \mathit{iff}\ \mathit{dom}(\sigma_1) = \mathit{dom}(\sigma_2) \wedge \forall x \in \mathit{dom}(\sigma_1).\sigma_1(x) \underset{H_1,H_2}{\approx} \sigma_2(x)$$

$$\forall \Gamma_1 : (H_1, \sigma_1, P), \Gamma_2 : (H_2, \sigma_2, P).\ \Gamma_1 \approx \Gamma_2 \ \mathit{iff}\ \sigma_1 \underset{H_1,H_2}{\approx} \sigma_2$$

Structural equivalence as defined above is a *congruence relation* with respect to the operational semantics. That is, evaluation of an expression $e$ under structurally equivalent environments produces structurally equivalent environment and result values, and also has *identical* resource usage.

**Lemma 1.** *For all* $\{\Gamma_1, \Gamma_2\} \subseteq \mathit{Env}$ *such that* $\Gamma_1 \approx \Gamma_2$*, for all expression* $e$*,*

$$\Gamma_1 \vdash e \Downarrow_p u, \Gamma_1{}' \implies \exists v, q, \Gamma_2{}'.\ \Gamma_2 \vdash e \Downarrow_q v, \Gamma_2{}' \wedge \Gamma_1{}' \approx \Gamma_2{}' \wedge u \underset{H_1',H_2'}{\approx} v \wedge p = q$$

*Proof.* The claim directly follows by structural induction over the operation semantic rules shown in Fig. 2.2. This structural induction strategy is discussed in detail in section 2.6 later in this chapter. □

### 2.3.4 Semantic Rules

Consider now the semantics rules shown in Figure 2.2. Most rules are straightforward now that the cost functions and structural equivalence relations have been introduced. Below I describe some of the complex rules. The rule LAMBDA creates a closure for a lambda term $t$, which is a pair consisting of $t$, and a (singleton) assignment for the variable captured from the enclosing context, which is given by $FV(t)$.

The rule CONCRETECALL defines the semantics of a call whose arguments have been evaluated to concrete values (in *Val*). It models the call-by-value parameter passing mechanism: it binds the parameters to argument values, and evaluates the body (an expression with contracts) under the new binding. A call evaluates to a value only if the contracts of the callee are satisfied as given by the rule CONTRACT (discussed shortly). This rule is used by the rule DIRECTCALL which evaluates direct call expressions of the language. The handling of direct calls is separated from that of concrete calls whose arguments are values to make the semantics amenable to extensions for incorporating memoization (discussed in chapter 4). Concrete calls serve as the keys of the memoization table, which can be evaluated independently.

The rule APP handles applications of the form: $x\ y$. It first evaluates $x$ to a closure $(\lambda z.e, \sigma')$, and then evaluates $e$ under the environment $\Gamma : (H, (\sigma \uplus \sigma')[z \mapsto \sigma(y)])$, where the store $(\sigma \uplus \sigma')$ includes the assignment for the captured variables. Note that the assignment in $\sigma'$ takes precedence over $\sigma$ for the captured variables, implying that the values of captured variables are frozen at the time of creation of the closure.

The rule CONTRACT defines the semantics of an expression $\tilde{e}$ of the form $\{pre\}\ e\ \{post\}$ that has contracts. Though contracts are expressions of the language, they are considered to be different from implementation. They are treated as specifications that describe the expected behavior of the implementation rather than being a part of the implementation. They are expected to be statically proven and not checked at runtime (at least under normal scenarios). The expression $\tilde{e}$ evaluates to a value $v$ only if *pre* holds in the input environment and *post* holds in the environment resulting after evaluating $e$. Observe that the value, heap effects, and resource usage of $\tilde{e}$ are equal to that of $e$. In other words, the resource usage and heap effects of evaluating the contracts are ignored. Also note that the resource variables (like steps) are bound to the resource consumption of $e$ before evaluating the postcondition.

## 2.4 Reachability Relation

One of the disadvantages of the big-step semantics is that when an evaluation of an expression under an environment is undefined, it could means two things: (a) either its evaluation diverges i.e, it does not terminate, or (b) its evaluation is stuck e.g. due a runtime exception or contract failure. In order to distinguish between these two situations, I define a relation $\rightsquigarrow$ called *reachability* relation inspired by the *calls* relation of Sereni [2006] and Jones and Bohr [2004] defined for a similar purpose. Intuitively, $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma', e' \rangle$ iff the big-step reduction of $e$ under $\Gamma$ requires reduction of $e'$ under $\Gamma'$. It is formally defined as follows.

For every semantic rule shown in Figure 2.2 with $n$ antecedents: $A_1 \cdots A_m B_1 \cdots B_n$, where $A_1 \cdots A_n$ are not big-step reductions, and each $B_i$, $i \in [1, n]$, is a big-step reduction of the form: $\Gamma_i \vdash e_i \Downarrow_{p_i} v_i, \Gamma_i'$, the reachability relation has $n$ rules for each $1 \leq i \leq n$ of the form: $\dfrac{A_1 \cdots A_m \qquad B_1 \cdots B_{i-1}}{\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma_i, e_i \rangle}$. Figure 2.3 presents the complete definition of this relation.

Note that the reach relation is quite different from a small-step operational semantics. Let $\rightsquigarrow^*$ represent the reflexive, transitive closure of $\rightsquigarrow$. An environment $\Gamma'$ is said to reach $e'$ during the evaluation of $e$ under $\Gamma$ iff $\langle \Gamma, e \rangle \rightsquigarrow^* \langle \Gamma', e' \rangle$.

**Termination of Big-step Evaluation** The evaluation of $e$ under $\Gamma$ is said to *diverge or non-terminate* iff there exists an infinite sequence $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma_1, e_1 \rangle \rightsquigarrow \cdots$. An expression $e$ (or a function $f$) *terminates* iff there does not exist a $\Gamma \in Env$ under which $e$ (or $body(f)$) diverges [Sereni, 2006]. Since the evaluation of an expression under the big-step operational semantics proceeds as a derivation tree that is finitely branching, for a terminating evaluation, there is a

$$\textsc{Let1}\quad \langle \Gamma, \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2\rangle \rightsquigarrow \langle \Gamma, e_1\rangle$$

$$\textsc{Let2}\quad \frac{\Gamma \vdash e_1 \Downarrow v_1, (H', \sigma')}{\langle \Gamma, \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2\rangle \rightsquigarrow \langle (H', \sigma'[x \mapsto v_1]), e_2\rangle}$$

$$\textsc{If1}\quad \frac{\sigma(x) = \mathit{true}}{\langle \Gamma, \mathsf{if}\ (\mathsf{x})\ e_1\ \mathsf{else}\ e_2\rangle \rightsquigarrow \langle \Gamma, e_1\rangle} \qquad \textsc{If2}\quad \frac{\sigma(x) = \mathit{false}}{\langle \Gamma, \mathsf{if}\ (\mathsf{x})\ e_1\ \mathsf{else}\ e_2\rangle \rightsquigarrow \langle \Gamma, e_2\rangle}$$

$$\textsc{Match}\quad \frac{H(\sigma(x)) = C_i\ \bar{v}}{\langle \Gamma : (H, \sigma), x\ \mathsf{match}\ \{C_i\ \bar{x_i} \Rightarrow e_i)\}_{i=1}^{n}\rangle \rightsquigarrow \langle (H, \sigma[\bar{x_i} \mapsto \bar{v}]), e_i\rangle}$$

$$\textsc{ConcreteCall}\quad \langle \Gamma : (H, \sigma), f\ u\rangle \rightsquigarrow \langle (H, \sigma[\mathit{param}(f) \mapsto u]), \mathit{body}(f)\rangle$$

$$\textsc{DirectCall}\quad \frac{f \in \mathit{Fids}}{\langle \Gamma, f\ x\rangle \rightsquigarrow \langle \Gamma, f\ \sigma(x)\rangle}$$

$$\textsc{App}\quad \frac{H(\sigma(x)) = (\lambda z.e, \sigma')}{\langle \Gamma : (H, \sigma), x\ y\rangle \rightsquigarrow \langle (H, (\sigma \uplus \sigma')[z \mapsto \sigma(y)]), e\rangle}$$

$$\textsc{Pre}\quad \langle \Gamma, \{pre\}\ e\ \{post\}\rangle \rightsquigarrow \langle \Gamma, pre\rangle$$

$$\textsc{Body}\quad \frac{\Gamma \vdash \mathit{pre} \Downarrow \mathit{true}}{\langle \Gamma, \{\mathit{pre}\}\ e\ \{\mathit{post}\}\rangle \rightsquigarrow \langle \Gamma, e\rangle}$$

$$\textsc{Post}\quad \frac{\Gamma \vdash \mathit{pre} \Downarrow \mathit{true} \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (H_2, \sigma_2)}{\langle \Gamma, \{\mathit{pre}\}\ e\ \{\mathit{post}\}\rangle \rightsquigarrow \langle (H_2, \sigma_2[R \mapsto q, \mathsf{res} \mapsto v]), post\rangle}$$

Figure 2.3 – Definition of the reachability relation

natural number $n$ such that the length of every sequence of the form $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_1, e_1\rangle \rightsquigarrow \cdots$ is upper bounded by $n$. That is, $\exists n \in \mathbb{N}.\neg\left(\exists k > n, e, \Gamma'.\langle \Gamma, e\rangle \rightsquigarrow^k \langle \Gamma', e\rangle\right)$ iff $e$ terminates under $\Gamma$.

## 2.5 Contract and Resource Verification Problem

### 2.5.1 Valid Environments

While the semantics rules shown in Figure 2.2 may be applicable to arbitrary environments, in reality, the environments under which an expression is evaluated satisfies several invariants which are ensured either by the runtime (e.g. that every free variable in an expression is bound in the environment), or by the program under execution (e.g. that a value of a variable is always positive). As in prior works on data structure verification [Kapur et al., 2006], it is only reasonable to define the problem of contract/resource verification with respect to such valid environments under which an expression can be evaluated, instead of under arbitrary environments. For this purpose, I first define a notion of valid environments.

Let $P_c = P' \| P$ denote a closed program obtained by composing a client $P'$ with an open program $P$. Note that $P_c$ has to satisfy the requirements of the program namely that the labels

of all expressions are unique, the function identifiers are unique etc. The evaluation of a closed program $P_c$ starts from a distinguished entry expression $e_{entry}$ such as a call to the main function under an initial environment $\Gamma_{P_c} : (\emptyset, \emptyset, \emptyset, P_c)$. The valid environments of an expression $e$ belonging to an open program $P$, denoted $Env_{e,P}$, are the environments that reach $e$ during some closed evaluation $P' \| P$. That is,

$$Env_{e,P} = \{\Gamma \mid \exists P'.\langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, e\rangle\}$$

The valid environments reaching an expression may satisfy many properties beyond those ensured by the domain invariants. Though some of which could be program specific, some of them are ensured by the semantics itself. For instance, the following lemma states that in all valid environments reaching an expression $e$, every free variable of $e$ would be bound to a value.

**Lemma 2.** *Let $e$ be an expression in a program $P$ and $\Gamma : (H, \sigma, P) \in Env_{e,P}$. $FV(e) \subseteq dom(\sigma)$.*

The proof of the above lemma is easy to establish from the definition of $Env_{e,P}$ by inducting on the number of steps to reach $\langle \Gamma, e\rangle$ from a closed program $\langle \Gamma_{P' \| P}, e_{entry}\rangle$.

### 2.5.2 Properties of Undefined Evaluations

This section establishes an important property about the semantics presented in Figure 2.2. Under the assumption that all primitive operations are total, when an expression belonging to a type correct program is evaluated under a *valid* environment, there are only two reasons why its evaluation may be undefined as per the operational semantics: (a) the evaluation diverges, or (b) there is a contract violation during the evaluation. This property is very important since the definition of contract verification presented shortly relies on this property. Before formally establishing this property I state and prove a few important notions and lemmas.

**Def 2** (**Contract Violation**)**.** *Given an expression $\tilde{e}$ with contract: $\tilde{e} = \{p\}\ e\ \{s\}$. The contract of $\tilde{e}$ is said to have been violated under $\Gamma$ iff $\Gamma \vdash p \Downarrow false \vee \exists \Gamma'.(\langle \Gamma, \tilde{e}\rangle \rightsquigarrow \langle \Gamma', s\rangle \wedge \Gamma' \vdash s \Downarrow false)$.*

The following lemma states that whenever an evaluation of an expression is undefined under an environment then either the expression has a contract and is violated, or the evaluation of some (immediate) sub-expression of $e$ is undefined.

**Lemma 3.** *Let $e$ be an expression in a* type-correct *program $P$. Let $\Gamma : (H, \sigma, P) \in Env$ be such that $FV(e) \subseteq dom(\sigma)$. If $\neg\exists v.\Gamma \vdash e \Downarrow v$ then either $e$ has a contract and is violated under $\Gamma$, or $\exists \Gamma', e'.\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma', e'\rangle$ and $\neg\exists v.\Gamma' \vdash e' \Downarrow v$.*

*Proof.* This property can be proved by exhaustively checking each of the semantic rules shown Fig. 2.2. Consider the rules CST, VAR, EQUAL, LAMBDA, CONS. Each of these rule do not have a big-step reduction in the antecedent. Every value required by the rule are either the output of

a total function like *fresh* which is always defined, or $\sigma(x)$ or $\sigma(H(x))$, where $x$ is a free variable in $e'$, and $\sigma$ and $H$ are the store and heap components of $\Gamma'$. By definition, $FV(e) \subseteq dom(\sigma)$, and $\Gamma$ satisfies all the domain invariants. Thus, both $\sigma(x)$ and $\sigma(H(x))$ are defined. Hence, the rules must be defined whenever the expression $e$ matches the consequent.

Now consider the rule APP, CONCRETECALL, DIRECTCALL, MATCH, IF, PRIM or LET. In addition to requiring that $\sigma(x)$ or $\sigma(H(x))$ are defined, these rule require more properties on shape (or type) of $\sigma(x)$ in order to be defined for an expression matching the consequent. In the case of INDIRECTCALL, $\sigma(H(x))$ is required to be a closure. In the case of MATCH, $\sigma(H(x))$ is required to be a datatype with the constructors that match the patterns in the MATCH construct. In the case of PRIM, $\sigma(x)$ should have the type of the argument of the primitive operation $pr$. (Recall that every primitive operation is total.) In the case of IF the condition should evaluate to a boolean. In the case of CONCRETECALL the function identifier must have a definition in the program $P$. All of these are properties guaranteed by the type checker. Since $\Gamma$ satisfies all the domain invariants, $\sigma(x)$ should inhabit the $type_P(x)$. Since it is given that the program $P$ type checks, $type_P(x)$ will satisfy the above requirements in each of the rules. Hence, every value required by these rules that are not big-step reductions will be defined. If every big-step reduction $\Gamma' \vdash e' \Downarrow v$ in the antecedent of these rules produce a value, then clearly $\exists v.\Gamma \vdash e \Downarrow v$ (see Fig. 2.2), because every other antecedent will be satisfied as explained above. Therefore, for evaluation of $e$ to be undefined $\exists \Gamma', e'.\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma', e'\rangle$ and $\neg\exists v.\Gamma' \vdash e' \Downarrow v$.

Now consider the rule CONTRACT. Let $e$ be an expression with contracts i.e, $\{p\}\ e'\ \{s\}$. First, if the pre-or post-condition of $e$ evaluates to *false*, the contract of $e$ is violated and hence the claim holds. If $e'$ evaluates to any value and $p$ or $s$ evaluate to *true* then $e$ evaluates to a value. Therefore, for evaluation of $e$ to be undefined $e'$ or $p$ or $s$ does not evaluate to a value. Hence the claim. □

Now it is easy to show that if an evaluation of an expression is undefined in a valid environment then either it has a contract violation or its evaluation diverges.

**Lemma 4.** *Let $e$ be an expression in a* type-correct *program $P$. Let $\Gamma : (H, \sigma, P) \in Env$ be such that $FV(e) \subseteq dom(\sigma)$. If $\neg\exists v.\Gamma \vdash e \Downarrow v$ then (a) there exists an infinite sequence $\langle \Gamma, e\rangle \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \cdots$, or (b) $\exists \Gamma' \in Env$ and an expression with contract $\tilde{e}$ such that $\langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', \tilde{e}\rangle$ and the contract of $\tilde{e}$ is violated under $\Gamma'$.*

*Proof.* Say $\neg\exists v.\Gamma \vdash e \Downarrow v$. Say there is no infinite sequence starting from $\langle \Gamma, e\rangle$, otherwise the claim trivially holds. Assume that there exists a $e'$ and $\Gamma'$ such that $\langle \Gamma, e\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle$ and $e'$ is undefined under $\Gamma'$. By Lemma 3 either $e'$ and hence $e$ has a contract violation or there exists a $e''$ reachable from $e$ in $k+1$ steps whose evaluation is undefined. By induction this holds for all $k$. Since the evaluation of $e$ is terminating, there exists a $k$ such that there is no expression and environment pair reachable from $e$ in $k$ steps. Thus, $e$ must have a contract violation. □

### 2.5.3 Problem Definition

**Contract Verification Problem** Given a program $P$ without templates. The contract verification problem is to decide for every function defined in the program $P$ of the form **def** $f\ x := \tilde{e}$, where $\tilde{e} = \{pre\}\ e\ \{post\}$, whether in every valid environment that reaches $\tilde{e}$ in which $pre$ does not evaluate to $false$, $e$ evaluates to a value. That is, for every **def** $f\ x := \tilde{e} \in P$,

$$\forall \Gamma : (H, \sigma, P) \in Env_{\tilde{e},P}.\ \exists v.\ (\Gamma \vdash pre \Downarrow false) \vee \Gamma \vdash \tilde{e} \Downarrow v$$

For conciseness, the quantification on $v$ is omitted when there is no ambiguity. Note that since contracts in our programs can specify bounds on resources, the above definition also guarantees that the properties on resources hold. The above condition also mandates that whenever the precondition of $\tilde{e}$ holds, $\tilde{e}$ evaluates to value, which implies that it terminates. Thus, the above definition corresponds to *total correctness*.

**Resource Inference Problem** Recall that the resource bounds of functions are allowed to be templates. In this case, the problem is to find an assignment $\iota$ for the holes such that in the program obtained by substituting the holes with their assignment, the contracts of all functions are verified. This is formally stated below. Let $P\ \iota$ denote the program obtained by substituting every hole $a$ in the expressions of $P$ by $\iota(a)$. The resource bound inference problem for a program $P$ with holes is to find an assignment $\iota$ such that for every function **def** $f\ x := \{pre\}\ e\ \{post\}$ in $P\ \iota$, $\forall \Gamma \in Env_{e,P}.\exists v.\ (\Gamma \vdash pre \Downarrow false) \vee \Gamma \vdash \{pre\}\ e\ \{post\} \Downarrow v$. Note that the assignment $\iota$ is global across the entire program. But because of the syntactic restriction that the holes cannot be shared across functions, $\iota$ can be seen as a union of assignments one for each function in the program.

## 2.6 Proof Strategies

In this section, I outline the proof strategies used in this article for proving properties about the operation semantics and relations such as structural equivalence which are recursively defined on the semantic domains.

**Structural Induction over Big-step Semantic Rules** One of the primary ways to establish a property $\rho(\Gamma, e, v, \Gamma', p)$ for a terminating evaluation $\Gamma \vdash e \Downarrow_p v, \Gamma'$ is to use induction over the depth of the evaluation. Given a property $\rho$, one establish by induction on $n \in \mathbb{N}$ that $\forall n \in \mathbb{N}$. $\neg\left(\exists k > n, e, \Gamma''.\langle \Gamma, e\rangle \rightsquigarrow^k \langle \Gamma'', e\rangle\right) \Rightarrow \rho(\Gamma, e, v, \Gamma', p)$. This gives rise to the following structural induction. For every semantics rule RULE, one can assume that the property holds for the big-step reductions in the antecedent and establish that it holds in the consequent. The base cases of the induction are the rules: CST, VAR, PRIM, EQUAL, CONS, LAMBDA, which do not have any big-step reductions in the antecedents. Every other rule is an inductive step. This is referred to as structural induction over the big-step semantic rules. Many of the theorems that

follow are established using this form of structural induction.

**Structural Induction over Semantic Domain Relations** Recall that the relations such as structural equivalence and simulation ($\approx$) are defined recursively on the semantic domains. (More such relations are introduced in the later sections). As is usual, these relations are defined using least fixed points, which is naturally provides an induction strategy explained below. Let $R \subseteq A^n$ be a relation defined by a recursive equation $R = h(R)$ where $h$ is some function, generally defined piece-wise like $\approx$. The solution for the above equation is the least fixed point of $h$. Since relations are sets of pairs, there exists a natural partial order on the relations namely $\subseteq$. The ordered set $(2^{A^n}, \subseteq)$ is a complete lattice, which implies that there exists a unique least fixed point for every monotonic function (by Knaster-Tarski theorem). Also, the least fixed point can be computed using Kleene iteration. Let $R^0 = \emptyset$ and $R^i = h(R^{i-1})$. The least fixed point of $h$, and hence the solution to $R$, is $\bigcup_{i \geq 0} R^i$. This definition of $R$ naturally lends itself to an inductive reasoning: to prove a property on $R$, one needs to establish that (a) the property holds for $\emptyset$, and (b) that whenever it holds for $R^{i-1}$ it holds for $R^i$. In the context of the relation $\approx$, assuming that the property holds for $R^{i-1}$ means that the relation can be assumed to hold in the right-hand sides of the *iff* relation (see section 2.3). I refer to this as structural induction over $R$.

## 2.7 Properties of the Semantics

The following theorems establish interesting properties about the semantics, which are necessary for the soundness proofs presented later, and also serve to illustrate the use of these structural induction techniques.

**Acyclic Heaps** Recall that the heaps of the environments are required to acyclic. A heap $H \in \mathit{Heap}$ is acyclic iff there exists a well-founded, irreflexive (i.e, strict) partial order $<$ on $\mathit{dom}(H)$ such that for every $(a, v) \in \mathit{Heap}$ one of the following properties hold:

(a) $v \in \mathbb{Z} \cup Bool$, (or)
(b) $v = \mathit{cons}\ \bar{u}$ and $\forall i \in [1, |\bar{u}|].\ u_i \in \mathit{Adr} \Rightarrow u_i < a$, (or)
(c) $v = (e_\lambda, \sigma')$ and $\forall a' \in (\mathit{range}(\sigma') \cap \mathit{Adr}).\ a' < a$

**Lemma 5.** *Let $H$ be an acyclic heap. The structural equivalence relation $\underset{H}{\approx}$ is indeed an equivalence i.e, it is reflexive, transitive and symmetric:*
*(a)* $x \underset{H}{\approx} y \wedge y \underset{H}{\approx} z \Rightarrow x \underset{H}{\approx} z$
*(b)* $x \underset{H}{\approx} y \Rightarrow y \underset{H}{\approx} x$
*(c)* $x \underset{H}{\approx} x$

*Proof.* The transitivity and symmetry properties follow by structural induction over the rela-

tion $\underset{H}{\approx}$. For instance, consider the symmetry property and say $x$ and $y$ are constructor instances of the form $c\ \bar{a}$ and $c\ \bar{b}$. By hypothesis, $\forall i.a_i \underset{H}{\approx} b_i$ implies $\forall i.b_i \underset{H}{\approx} a_i$. Hence, $c\ \bar{a} \underset{H}{\approx} c\ \bar{b}$ implies $c\ \bar{b} \underset{H}{\approx} c\ \bar{a}$. Similarly other cases can be established.

The reflexivity property trivially holds for integers and booleans. To prove the property for addresses, we can induct over the well-founded relation $<$ on $Adr$. The base case consists of addresses in the heap that are mapped to values that do not use other addresses. The reflexivity property clearly holds in this case. The inductive case consists of addresses that are mapped to values, namely constructor or closure values. However, by hypothesis, every address they use satisfy the reflexivity property. Given this, the claim holds, since for two closure/constructor values to be structurally equal they have to invoke the same function or use the same constructor. □

The following lemma establishes that the acyclic heap property is preserved by the semantic rules and hence is a domain invariant. Similarly, other domain invariants can also be established.

**Lemma 6.** *Let $(H,\sigma,P)$ be such that $H$ is acyclic. If $(H,\sigma,P) \vdash e \Downarrow_p u,(H',\sigma',P)$ or $\langle(H,\sigma,P),e\rangle \rightsquigarrow \langle(H',\sigma',P),e'\rangle$, the heap $H'$ is also acyclic.*

*Proof.* It suffices to prove the claim for the case where $(H,\sigma,P) \vdash e \Downarrow_p u,(H',\sigma',P)$ the other part immediately follows by the definition of the reachability relation shown in Figure 2.3. For every base rule other than CONS and LAMBDA, $H$ and $H'$ are identical and hence the claim holds trivially. For CONS and LAMBDA rules, the acyclicity property holds since every address used by the value bound to the newly created address belongs to the heap $H$ and hence is acyclic. Consider a inductive rule like LET i.e, $e$ is of the form **let** $x := e_1$ in $e_2$. By hypothesis, the heap after the evaluation of $e_1$ and hence the one after the evaluation of $e_2$ are acyclic. Hence the claim holds. Other cases can be similarly proven. □

**Immutability of Heaps** The semantic rules ensure that the heaps are used in an immutable way i.e, during any evaluation only new entries are added to the heap and existing entries remain unchanged. The following two lemmas establish the immutable nature of the heap using structural induction on the operational semantic rules.

**Lemma 7.** *Let $\Gamma:(H,\sigma,P),\Gamma':(H',\sigma',P)$ and $e$ be an expression. If $\langle\Gamma,e\rangle \rightsquigarrow \langle\Gamma,e'\rangle$ or $\Gamma \vdash e \Downarrow v,\Gamma'$ then $H \sqsubseteq H'$.*

*Proof.* This directly follows from the semantic rules shown in Fig. 2.2. Every time an address is added to the heap, it is chosen to be a fresh address that is not already bound in the heap. □

**Referential Transparency** The following lemma establishes that the evaluation of an expression produces equivalent values when evaluated under different heaps arising during the evaluation (and hence related by the containment ordering). This property can be thought of as a form of referential transparency that only concerns heaps that arise during a single evaluation. Lemma 1 provides a stronger form of referential transparency for equivalent environments arising during possibly different evaluations.

**Lemma 8.** *Let* $\Gamma : (H, \sigma, P) \in Env$ *and* $e$ *be an expression. Let* $\Gamma' = (H', \sigma, P)$ *and* $H \sqsubseteq H'$. *If* $\Gamma \vdash e \Downarrow_p u, \Gamma_o$ *then* $\Gamma' \vdash e \Downarrow_p v, \Gamma'_o$ *and* $u \underset{H_o, H'_o}{\approx} v$. *That is, adding more entries to the heap preserves the result of the evaluation with respect to the structural simulation relation* $\approx$.

*Proof.* This immediately follow from Lemma 1 since $\Gamma \approx \Gamma'$ by definition of $\approx$ for environments. □

### 2.7.1 Encapsulated Calls

Our approach is primarily aimed at programs where the targets of all indirect calls that may be executed are available at the time of the analysis. This includes whole programs that take only primitive valued inputs/parameters, and also data structures that use closures internally but whose *public* interfaces do not permit arbitrary closures to be passed in by their clients. Such data structures are quite ubiquitous and include numerous sophisticated functional data structures. Some examples include lazy data structures proposed by Okasaki [1998] and the Conqueue data structure of Scala's data-parallel library [Prokopec and Odersky, 2015] (discussed in section 5). I would like to remark that proving resource bounds of programs where the full implementation is not available at the time of the analysis is quite challenging and is orthogonal to the work presented in this dissertation. The Related Works chapter (Chapter 6) discusses a plausible future direction for proving such programs. Below, I provide a formal description of the kind of encapsulated programs which are supported by our approach.

An indirect call $c = x\ y$ belonging to a program $P$ is an *encapsulated call* iff in every environment $\Gamma : (H, \sigma, P) \in Env_{c,P}$ that reaches the call, whenever $H(\sigma(x))$ is a closure $:({e_\lambda}^\ell, \sigma')$, $l \in labels_P$. Here, $labels_P$ denotes the set of labels of expressions in the program $P$. A program $P$ is *call encapsulated* iff every indirect call in $P$ is encapsulated. Languages like Scala and C♯ support access modifiers like *private* that permit creation of such encapsulated calls. Though the core language presented in this section does not support access modifiers, our implementation whose inputs are Scala programs does leverage access modifiers to identify encapsulated calls.

# 3 Solving Resource Templates with Recursion and Datatypes

> There can be no doubt that the knowledge of logic is of considerable practical importance for everyone who desires to think and to infer correctly.
> — Alfred Tarski

The input to our system is a functional Scala program where the resource bounds of functions are specified as expressions with numerical holes in their postconditions. Such expressions are referred to as *resource templates*. The syntax of the resource templates were presented in the previous chapter in Figure 2.1. The holes in the templates may appear as coefficients of variables in scope, which themselves could be bound to arbitrary expressions of the program through let-binders. The goal of our system is to infer values for the holes in the template that will yield an upper bound on the resource usage of the function under all possible executions, as stated by the resource verification problem (see section 2.5). The programs inputted to our system can have arithmetic/boolean operations, recursive functions, datatypes, closures and memoization, as described by the core language syntax shown in Figure 2.1. Our system therefore incorporates techniques that can solve resource templates in the presence of these language features. In this chapter I described an analysis that can infer resource template for programs having three of the five features listed above: arithmetic/boolean operations, recursive functions and datatypes. The subsequent chapters discuss major extensions to the analysis for supporting higher-order functions and memoization.

The program fragment considered in this chapter is quite expressive. It supports user-defined recursive functions and hence is Turing complete. (Nevertheless our termination checker generates warnings in presence of potential non-termination.) It also supports recursive data types which allows expressing many immutable data structures such as abstract syntax trees and heaps in a natural way. It further supports linear and non-linear arithmetic and hence enables expressing precise resource bounds. For instance, the algorithms presented in this chapter can show that a function converting a propositional formula into negation-normal form takes no more than $43 \cdot \mathsf{size}(f) - 17$ steps, where $\mathsf{size}(f)$ is the number of nodes in the

abstract syntax tree of the formula $f$. It also proves that the depth of the computation graph (time in an infinitely parallel implementation) is bounded by $5 \cdot h(f) - 2$, where $h(f) \geq 1$ is the height of the formula tree. As another example, it shows that in the worst case the number of steps required for inserting into a red-black tree is given by $132 \cdot \log(1 + \mathsf{size}(t)) + 77$, and that the number of heap-allocations (alloc) performed during the insert operation is bounded by $9 \cdot \log(1 + \mathsf{size}(t)) + 8$. Our evaluations have shown that the algorithm can scale to even more complicated data structures like binomial heaps, and can establish amortized resource bounds given suitable drivers that simulate *most-general* clients. (Section 5 discusses the benchmarks and the results in more detail.)

Our approach for the verifying such programs operates in two phases. In the first phase, it generates an instrumented program that accurately tracks the resource usage of expressions (described in section 3.1). The resource bounds of the input program become invariants of the instrumented program. In the second phase, using an assume-guarantee reasoning, the algorithm generates a $\exists\forall$ formula called verification condition (VC) such that any satisfying assignment to the formula yields a solution for the holes (described in section 3.2). The coefficients in the templates that are holes become existentially quantified variables of the VC. A novel decision procedure described in section 3.3 is used to infer values for the existentially quantified variables of the VC.

The coefficients in practice tend to be sufficiently large and numerous that simply trying out small values does not scale. The decision procedure therefore employs one of the most powerful techniques for finding unknown coefficients in invariants: Farkas' lemma. This method converts a $\exists\forall$ problem on parametric linear constraints into a purely existential problem over non-linear constraints. A major challenge addressed by the algorithm is that it provides a practical technique that makes such expensive non-linear reasoning work on formulas that contain many disjunctions, invocations of user-defined recursive functions and recursive data types (such as trees and DAGs). Our algorithm handles these difficulties through an incremental and counterexample-driven algorithm which soundly encodes datatypes and recursive functions, and fully leverages the ability of an SMT solver to handle disjunctions efficiently.

## 3.1 Resource Instrumentation

Our approach decouples the encoding of the semantics of resources from their analysis. This is accomplished by an exact instrumentation of programs with their resource usage that does not approximate conditionals or recursive invocations. The instrumentation severs two main purposes: (a) it makes the rest of the approach agnostic to the resource being verified thus aiding portability across multiple resources, and (b) it eliminates any precision loss in the encoding of the resource and restricts the *incompleteness* to a single source, namely the verification algorithm. The latter aspect is very important since it allows the users to help the verification algorithms with more specifications until the desired bounds are established.

$$
\begin{array}{rcl}
[\![x]\!] & = & (x, c_{var}) \\
[\![pr\ x]\!] & = & (pr\ x, c_{pr}) \quad \text{if } pr \in Prim \\
[\![x \text{ eq } y]\!] & = & (x \text{ eq } y, c_{eq}) \\
[\![C\ \bar{x}]\!] & = & (C\ \bar{x}, c_{cons}) \quad \text{if } C \in Cids \\
[\![\textbf{let } x := e_1 \text{ in } e_2]\!] & = & \textbf{let } u := [\![e_1]\!] \text{ in} \\
 & & \quad \textbf{let } w := [\![e_2[u.1/x]]\!] \text{ in} \\
 & & \quad\quad (w.1, c_{let} \oplus u.2 \oplus w.2) \\
[\![\textbf{if } x\ e_1 \textbf{ else } e_2]\!] & = & \textbf{if } x \textbf{ let } u := [\![e_1]\!] \text{ in } (u.1, c_{if} \oplus u.2) \\
 & & \textbf{else let } u := [\![e_2]\!] \text{ in } (u.1, c_{if} \oplus u.2) \\
[\![x \textbf{ match}\{C_i\ \bar{x_i} \Rightarrow e_i\}_{i=1}^{n}]\!] & = & x \textbf{ match } \{(C_i\ \bar{x_i} \Rightarrow \textbf{let } u := [\![e_i]\!] \text{ in } (u.1, c_{match(i)} \oplus u.2))_{i=1}^{n}\} \\
[\![f\ x]\!] & = & \textbf{let } w := f\ (x) \text{ in } (w.1, c_{call} \oplus w.2) \\
[\![\{pre\}\ e\ \{post\}]\!] & = & \{([\![pre]\!]).1\} \\
 & & \quad [\![e]\!] \\
 & & \{\textbf{let } y = [\![post[\text{res}.1/\text{res}][\text{res}.2/R]]\!] \text{ in } y.1\} \\
[\![\textbf{def } f\ x := e]\!] & = & \textbf{def } f\ x := [\![e]\!]
\end{array}
$$

Figure 3.1 – Resource instrumentation for first-order programs.

```
def revRec(l1:List, l2:List) : List = {
  l1 match {
    case Nil() ⇒ l2
    case Cons(x,xs) ⇒
      revRec(xs, Cons(x, l2))
  }
} ensuring(res ⇒ steps ≤ ?∗size(l1) + ?)
```

(a)

```
def revRec(l1:List,l2:List):(List,Int) = {
  l1 match {
   case Nil() ⇒ (l2, 2)
   case Cons(x,xs) ⇒
     val r = revRec(xs, Cons(x,l2))
     (r._1, 7 + r._2)
  }
} ensuring(res ⇒ res._2 ≤ ?∗size(l1) + ?)
```

(b)

Figure 3.2 – Illustration of instrumentation.

Figure 3.1 formally presents the instrumentation as a program transformation $[\![\cdot]\!]$ that is parameterized by the cost functions $\oplus$ and $c_{op}$. The function $[\![\cdot]\!]$ accepts an expression and returns a pair (i.e, a two element tuple) of expressions where the first component is the results of the input expression and the second component tracks the resource usage of the input expression. As in the semantics shown in Figure 2.3, the resource consumed by an expression $e$ is computed as a function of the resources consumed by its sub-expressions using the cost functions described in section 2.2. However, note that here the cost combinator $\oplus$ is applied over integer-valued expressions (instead of natural numbers). The resource usage of a procedure is exposed to its callers by augmenting the return value of the procedure with its resource usage. The resource consumption of a function call is determined as the sum of the resources consumed by the called function (which is exposed through its augmented return value) plus the cost of invoking the function.

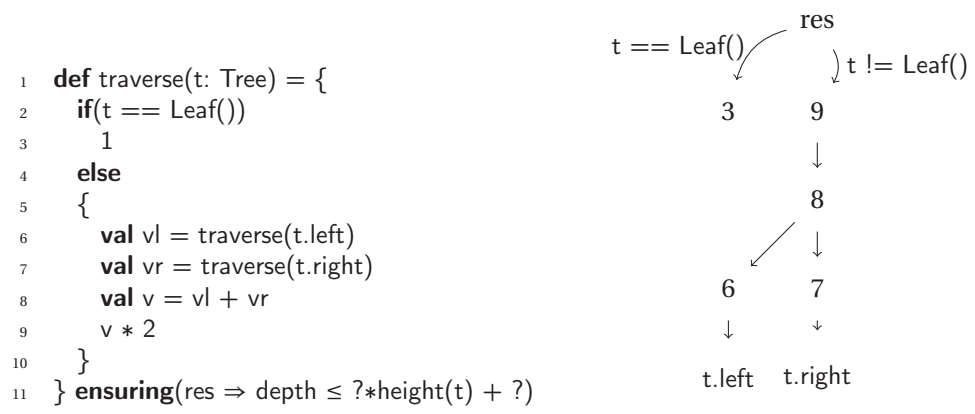



Figure 3.3 – Example illustrating the depth of an expression.

Fig. 3.2 illustrates the instrumentation on a simple Scala program that reverses a list l1 and appends a list l2 to it. The recursive function size counts the length of its list argument (omitted for brevity). Fig. 3.2(b) shows the instrumented program that would be generated by our system. The integer constants in the program are obtained after accumulating the constants along the corresponding static paths in the program.

### 3.1.1 Instrumentation for Depth

Depth [Blelloch and Maggs, 1996] is a measure of degree of parallelism in an expression and, intuitively, corresponds to the longest chain of data dependencies between the operations of an expression. It can be viewed as an estimate of parallel execution time when infinite parallel computation power is available and can be exploited without any overheads. Furthermore, it has been shown by Blelloch and Maggs [1996] that depth can be combined with steps to derive a good approximation of the parallel running time on a specific parallel architecture. From our perspective, this resource is interesting as it is measure of parallelism inherent to an implementation, and is independent of the runtime. Moreover, in principle, a similar instrumentation strategy can be used to measure evaluation steps in the presence structured parallelism constructs like *fork-join* parallelism.

To explain this resource, I will use a Scala function that traverses a tree shown in Figure 3.3(a). (Note that vals in Scala are similar to let-binders.) The *data dependence graph* for the procedure is shown at the right side of Fig. 3.3. The nodes of the graph correspond to the expression at the line number given by the label. The depth of a node $n$ in the tree, denoted as $d(n)$, is the maximum of the depth of its children plus the cost of the operation performed at the node. For instance, $d(8)$ is equal to $\max\big(d(6)+d(7)\big)$ plus the cost of the operation + (which is 1 for every primitive operation).

The depth of if-else and match expressions are guarded by the branch conditions. For instance,

```
def traverse(t: Tree):(Tree,Int)= {
  if(t == Leaf())
    (1, 2)
  else {
    val lr = traverse(t.left)
    val rr = traverse(t.right)
    val v = lr._1 + rr._1
    (v * 2, lr._2 + rr._2 + 9)
  }
} ensuring(res ⇒ res._2 ≤ ?*height(t) + ?)
```

Figure 3.4 – Illustration of depth instrumentation.

$d(\text{res})$ is $\max\big(d(cond), d(3)\big) + c_{if}$ if the condition t=Leaf() is true, and is $\max\big(d(cond), d(4)\big) + c_{if}$ otherwise. $d(cond)$ is the depth of computing the branch condition. This instrumentation is based on the assumption that the testing of the branch condition can be performed in parallel with the evaluation of the bodies of the then and else branches.

The depth of a function is the depth of its result, which is the root of the data dependence graph. The most interesting nodes of the graph are the nodes 6 and 7 that correspond to recursive invocations of the procedure traverse. The depth of a function call is the depth of its callee, which is exposed by its second return value, *plus* the depth of its argument and the cost of a function call. In essence, the depth of the root of the graph (which is the depth of traverse (t)) is a constant $k_1$ if t == Leaf(), otherwise is $\max(d(\text{traverse(t.left)}), d(\text{traverse(t.right)})) + k_2$ if t != Leaf(). Note that this quantity is propositional to the height of the input tree t (unlike the resource steps which is proportional to the size of the tree). Therefore, the depth of the function is specified as a template linear in the height of the tree. For this example, our system inferred the solution as $\text{depth} \leq 9 * \text{height(t)} + 2$. Figure 3.4 shows the instrumented program generated by our tool for the traverse function.

It is difficult to express the semantics of this resource only using the cost functions presented in section 2.3, partly because the depth instrumentation for expressions naturally proceeds top-down as opposed to other resources. Therefore, I separately formalize the instrumentation for the depth resource in Figure 3.5. The transformation function $[\![\cdot]\!]$ is extended in this case to accept a partial function $d$ that maps variables to the depth of the computation that produces their values. At the start of a function, the depths of the parameters of the function are initialized to zero (see $[\![\textbf{def}\ f\ x := e]\!]$), as they are already computed by the callers of the function. The main aspect that distinguishes depth from steps is the handling of the let-expression: **let** $x := e_1$ in $e_2$. The let-binder $x$ is bound to the depth of the expression it is assigned to: $e_1$. But, the depth of $e_1$ does not immediately contribute to the depth of the enclosing let-expression. It does so only when the binder $x$ is used in $e_2$. This ensures that the depth of the expression $e_2$ depends only on the depth of the free variables used by the expression and not on all let-bindings that precede it in the program order. Another aspect

$$
\begin{array}{rcl}
[\![x]\!]\ d & = & (x, d(x)) \\
[\![pr\ x]\!]\ d & = & (pr\ x, c_{pr} + d(x)) \quad \text{if } pr \in Prim \\
[\![x\ \mathsf{eq}\ y]\!]\ d & = & (x\ \mathsf{eq}\ y, c_{eq} + \max(d(x), d(y))) \\
[\![C\ \bar{x}]\!]\ d & = & (C\ \bar{x}, c_{cons} + \max_{1 \le i \le |\bar{x}|} x_i) \quad \text{if } C \in Cids \\
[\![\mathbf{let}\ x\ :=\ e_1\ \mathsf{in}\ e_2]\!]\ d & = & \mathbf{let}\ u := [\![e_1]\!]\ d\ \mathsf{in} \\
 & & \quad \mathbf{let}\ w := [\![e_2[u.1/x]]\!]\ d[x \mapsto u.2]\ \mathsf{in} \\
 & & \quad\quad (w.1, c_{let} + \max(u.2, w.2)) \\
[\![\mathbf{if}\ x\ e_1\ \mathbf{else}\ e_2]\!]\ d & = & \mathbf{if}\ x\ \mathbf{let}\ u\ :=\ [\![e_1]\!]\ d\ \mathsf{in}\ (u.1, c_{if} + \max(d(x), u.2)) \\
 & & \mathbf{else\ let}\ u\ :=\ [\![e_2]\!]\ d\ \mathsf{in}\ (u.1, c_{if} + \max(d(x), u.2)) \\
[\![x\ \mathbf{match}\{C_i\ \bar{x}_i \Rightarrow e_i\}_{i=1}^{n}]\!]\ d & = & x\ \mathbf{match}\ \{(C_i\ \bar{x}_i \Rightarrow \\
 & & \quad \mathbf{let}\ u := [\![e_i]\!]\ d\ \mathsf{in}\ (u.1, c_{match(i)} + \max(d(x), u.2)))_{i=1}^{n}\} \\
[\![f\ x]\!]\ d & = & \mathbf{let}\ w := f\ (x)\ \mathsf{in}\ (w.1, c_{call} + d(x) + w.2)) \\
[\![\{pre\}\ e\ \{post\}]\!]\ d & = & \{([\![pre]\!]\ d).1\} \\
 & & \quad [\![e]\!]\ d \\
 & & \{\mathbf{let}\ y = [\![post[\mathsf{res}.1/\mathsf{res}][\mathsf{res}.2/R]]\!]\ d\ \mathsf{in}\ y.1\} \\
[\![\mathbf{def}\ f\ x := e]\!] & = & \mathbf{def}\ f\ x := [\![e]\!]\ [x \mapsto 0]
\end{array}
$$

Figure 3.5 – Instrumentation for the depth resource.

that may concern the readers is that the depth of a variable $x$ is (re)accounted for at every point of its use instead of only at the point of first use. But this does not make a difference to depth calculation due to the combinator max. In essence, the instrumentation presented here ends up computing the maximum of the depth of the variable $x$ over all its uses, which is same as accounting for the depth of $x$ once.

**Simultaneously Instrumenting for Multiple Resources** The instrumentation described in this section can be extended to track multiple resources simultaneously. Instrumenting an expression $e$ for $n$ resources would result in an expression that computes an $n+1$-tuple where the first element of the tuple computes the result of $e$ and the remaining $n$ elements track the usage of the $n$ resources by the expression $e$. The $i^{th}$ element of the tuple (where $i \geq 2$) is updated using the corresponding $i^{th}$ component of the instrumentation of the subexpressions of $e$ and the cost functions defining the $i^{th}$ resource. This ability of the system to simultaneous track multiple resources is particularly essential for the divide-and-conquer reasoning described later in section 3.8.

**From Resource Bounds to Invariants** After the instrumentation phase, the resource bounds of functions become invariants of the return value of the function that tracks its resource usage: the variable res._2 in the above examples. Every inductive invariant for the instrumented function obtained by solving for the holes is a valid bound for the resource consumed by the

original function. Moreover, the strongest invariant (formalized later) is also the strongest bound on the resource. Notice that the instrumentation increases the program sizes, introduces numerous tuples and, in the case of *depth* instrumentation, creates many max operations that involve disjunctions. In the following sections I discuss a template-based invariant inference technique that can handle these features effectively.

## 3.2 Modular, Assume-Guarantee Reasoning

This section discusses the reasoning that our system uses to reduce the problem of checking contracts of a (first-order) function to that of proving validity of predicates (i.e, boolean-valued expressions). The constraints generated by this reasoning are solved by translation to formulas in a suitable logic and by applying customized decision procedures, which is the subject of the next section. The aim of this section is to separate the principle underlying reduction of contract checking to constraint solving from the encoding of the constraints into logic and its decision procedure. This separation simplifies the understanding of the soundness and completeness of the system, and also allows enhancing or adapting one aspect while reusing the other. In particular, in Chapter 4, I describe an extension to the assume-guarantee reasoning that enables a more effective and less-overhead contract verification for higher-order functions with memoization.

### 3.2.1 Function-Level Modular Reasoning

Under this reasoning, to establish that the contract of a function $f$ is valid, one can assume (or hypothesize) that the pre-and post-condition of the functions called by $f$ (including itself) hold at the call sites within the function $f$, and guarantee that the postcondition of $f$ holds for every valid environment that satisfy the precondition of $f$. Also, the precondition of each function has to be guaranteed at its call sites. This assume-guarantee reasoning relies on induction over the number of calls made by the function $f$, which is finite and well-founded if the function $f$ is terminating. In essence, this reasoning provides a sound way of checking *partial correctness* of contracts. As mentioned in the Introduction, our system verifies the termination of functions independently using termination checker of the underlying LEON verifier [Nicolas Voirol and Kuncak, 2017]. Thus it ensures that all functions terminate on all valid environments and hence their contracts hold for all valid environments.

I would like to remark that while this reasoning is a widely employed technique in correctness verification, it has hitherto not been used in resource verification primarily because of the need for an independent termination checking algorithm different from the resource analysis. In this dissertation, I demonstrate that decoupling resource verification from termination makes the former tangible on complex programs where the resource bounds and invariants themselves rely on recursive functions (which is almost always the case with our benchmarks). In such cases, establishing a meaningful time (or resource) bound on functions meant for use in specifications is unnecessary and also sometimes infeasible without changing the

implementation. For instance, in the program shown in Figure 3.5, it is meaningless to prove a time bound for the function height. On the other hand, proving termination requires much simpler reasoning and effort, but when established it permits a powerful inductive reasoning such as the one described here for proving other properties including resource bounds. As we proceed to a language with support for lazy evaluation and memoization (Chapter 4) this decoupling becomes all the more important.

In the sequel, I formalize the assume-guarantee reasoning and subsequently show the soundness of this reasoning for contract and resource verification (see section 2.5 for their definition).

**Semantic Implication.** Let $e_1$ and $e_2$ be two predicates i.e, boolean-valued expressions. Let $e_1 \rightarrow e_2$ denote that for every environment that has a binding for the free variables of $e_1$ and $e_2$, whenever $e_1$ does not evaluate to *false*, $e_2$ evaluates to *true* i.e,

$$e_1 \rightarrow e_2 \quad \textit{iff} \quad \forall \Gamma \in \{(H,\sigma,P) \in Env \mid FV(e_1) \cup FV(e_2) \subseteq dom(\sigma)\}.\Gamma \vdash e_1 \Downarrow \mathit{false} \vee \Gamma \vdash e_2 \Downarrow \mathit{true}$$

The operation $\rightarrow$ can be considered as an implication with respect to the operational semantics of the language. Let $\models_P e_1 \rightarrow e_2$ denote that under the *assumption* that all functions invoked by $e_1$ and $e_2$ terminate in all environments that reaches them, and their pre-and post-conditions hold, $e_1 \rightarrow e_2$ is *guaranteed*. That is,

$$\begin{aligned} \models_P e_1 \rightarrow e_2 \quad \textit{iff} \quad & \forall \Gamma \in \{(H,\sigma,P) \in Env \mid FV(e_1) \cup FV(e_2) \subseteq dom(\sigma)\}. \\ & \neg\mathscr{A}(\Gamma, e_1) \vee \neg\mathscr{A}(\Gamma, e_2) \vee \Gamma \vdash e_1 \Downarrow \mathit{false} \vee \Gamma \vdash e_2 \Downarrow \mathit{true} \end{aligned}$$

Where $\mathscr{A}(\Gamma, e)$ denotes an assumption defined by: $\bigwedge \{\exists v.\Gamma' \vdash (f\ x) \Downarrow v \mid (\Gamma', (f\ x)) \in \mathit{Calls}(\Gamma, e)\}$ and $\mathit{Calls}(\Gamma, e) = \{(\Gamma', (f\ x)) \mid \langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', (f\ x)\rangle\}$

That is, $\mathit{Calls}(\Gamma, e)$ denotes a set of environment, call pairs that are reachable during the evaluation of $e$ under $\Gamma$, and the assumption $\mathscr{A}(\Gamma, e)$ assumes that the calls terminate and all their contracts hold. The function-level, modular reasoning described above corresponds to establishing the following constraints, which are also referred to as assume-guarantee obligations:

***Obligations of Function-level modular reasoning:***
For each function definition **def** $f\ x := \{pre\}\ e\ \{post\}$ in a program $P$,

(2.I) $\models_P pre \rightarrow post[e/\mathsf{res}]$

(2.II) For each call site $(f\ y)^\ell$ in $P$, $\models_P path((f\ y)^\ell) \rightarrow pre(f\ y)$

Recall that the variable res refers to the result of the function in its postcondition. Let $pre(f\ y)$ denote the precondition of $f$ after parameter translation i.e, $p[y/x]$ if **def** $f\ x := \{p\}\ e\ \{s\} \in P$. The path condition $path((f\ y)^\ell)$ denotes the *static path*, possibly with disjunctions and function calls, to the expression labeled $\ell$ from the entry point of the function containing the label $\ell$. For instance, for the instrumented program shown in Figure 3.2(b) the path condition of the

$$
\begin{array}{rcl}
\mathscr{C} \in \mathit{PathContext} & = & [\,] \mid \textbf{let}\ x\ :=\ e_1\ \textsf{in}\ \mathscr{C} \mid \textbf{let}\ x\ :=\ \mathscr{C}\ \textsf{in}\ e_2 \mid \textbf{if}\ x\ \mathscr{C}\ \textbf{else}\ e_2 \\
 & & \mid \textbf{if}\ x\ e_1\ \textbf{else}\ \mathscr{C} \mid x\ \textbf{match}\ \{C_1\ \bar{x} \Rightarrow \mathscr{C}\} \mid \{\mathscr{C}\}\ e\ \{s\} \\
 & & \mid \{p\}\ \mathscr{C}\ \{s\} \mid \{p\}\ e\ \{\mathscr{C}\} \\
\mathit{path}([\,]) & = & \mathit{true} \\
\mathit{path}(\textbf{let}\ x\ :=\ e_1\ \textsf{in}\ \mathscr{C}) & = & x = e_1 \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(\textbf{let}\ x\ :=\ \mathscr{C}\ \textsf{in}\ e_2) & = & \mathit{path}(\mathscr{C}) \\
\mathit{path}(\textbf{if}\ x\ \mathscr{C}\ \textbf{else}\ e_2) & = & x \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(\textbf{if}\ x\ e_1\ \textbf{else}\ \mathscr{C}) & = & \neg x \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(x\ \textbf{match}\ \{C_1\ \bar{y} \Rightarrow \mathscr{C}\}) & = & (x = C_1\ \bar{y}) \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(\{\mathscr{C}\}\ e\ \{s\}) & = & \mathit{path}(\mathscr{C}) \\
\mathit{path}(\{p\}\ \mathscr{C}\ \{s\}) & = & p \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(\{p\}\ e\ \{\mathscr{C}\}) & = & p \wedge \mathsf{res} = e \wedge \mathit{path}(\mathscr{C}) \\
\mathit{path}(e^{\ell}) & = & \mathit{path}(\mathscr{C}), \qquad \text{if } \exists\ \textbf{def}\ f\ x := \mathscr{C}[e^{\ell}] \in P
\end{array}
$$

Figure 3.6 – Definition of the path condition for an expression belonging to a program $P$

recursive call revRec is l1 = Cons(x,xs). Figure 3.6 formally defines how the path conditions are constructed by our system. (It is assumed in the definition that the names of the variables are unique.) The path conditions are generated by traversing the syntax tree of the body of the function containing the expression with label $\ell$. Let a context $\mathscr{C}$ denote an expression with a hole, where the hole appears at the position of the expression with label $\ell$ whose path condition has to be determined. The function *path* is therefore defined on all such context which is given by *PathContext*. For brevity, path condition is defined for match construct with only one case. It is straightforward to generalize this to arbitrary cases. Note that if the expression $e^{\ell}$ is in the body or postcondition of a function, the precondition of the function is a conjunct of the path condition. The path condition for an expression $e^{\ell}$ thus generated has the property that every environment that reaches the expression makes the path condition true, as stated by the following lemma.

**Lemma 9.** *For any expression* $\mathscr{C}[e^{\ell}]$, $\forall \Gamma \in \mathit{Env}.\ \langle \Gamma, \mathscr{C}[e^{\ell}]\rangle \rightsquigarrow^* \langle \Gamma', e^{\ell}\rangle \Rightarrow \Gamma' \vdash \mathit{path}(\mathscr{C}) \Downarrow \mathit{true}$

*Proof.* The proof follows by induction on the structure of the context $\mathscr{C} \in \mathit{PathContext}$. □

Observe that this modular reasoning requires that the assume/guarantee constraints hold for all environments in which the parameters are bound to a (type-correct) value (by the definition of $\rightarrow$). However, as per the definition of contract verification presented in section 2.5 it suffices to consider only valid environments that reach the function bodies. This means that pre-and post-conditions of functions should capture all necessary invariants needed for the verification of contracts. That is, every other global program invariant that is known to hold for a function (e.g. discovered through an independent static analysis) have to be encoded using the pre-and post-conditions if they have to be used by this reasoning.

Below I present the proof of soundness of this assume-guarantee reasoning for the contract

verification problem. I start with a lemma that establishes that for any function and environment that has a binding for the parameter of the function, for any natural number $n$ either the evaluation of function terminates and satisfies its contracts or there exists a function call reachable from the body of the function in more than $n$ steps with respect to the $\rightsquigarrow$ relation.

**Lemma 10.** *If the function-level, assume/guarantee constraints given by constraints (2.I) and (2.II) hold for a program $P$, the following property holds for all $n \in \mathbb{N}$.*

$$\forall (\textbf{def}\ f\ x := \tilde{e}) \in P\ s.t.\ \tilde{e} = \{p\}\ e\ \{s\}. \forall \Gamma \in \{(H, \sigma, P) \in Env \mid x \in dom(\sigma)\}.$$
$$\Big(\exists k > n, h \in Fids, y \in Vars, \Gamma'. \langle \Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle \Gamma', (h\ y)\rangle\Big) \vee \big(\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v\big)$$

*Proof.* We prove this using induction on $n$. Intuitively, $n$ imposes a limit on the number of direct function calls we need to consider while proving that the contract of the function $f$ holds. The base case are evaluations that make zero direct calls. For every function **def** $f\ x := \tilde{e} \in P$ where $\tilde{e} = \{p\}\ e\ \{s\}$, we need to prove that

$$\forall \Gamma \in \{(H, \sigma, P) \in Env \mid x \in dom(\sigma)\}. \Big(\exists k > 0, h \in Fids, y \in Vars. \langle \Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle \Gamma', (h\ y)\rangle\Big) \vee$$
$$\big(\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v\big)$$

Consider a $\Gamma$ such that $\neg\big(\exists k > 0, h, y. \langle \Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle \Gamma', (h\ y)\rangle\big)$. Otherwise the claim trivially holds. This essentially means that we do not encounter a direct call either during the evaluation of $p$ or $\tilde{e}$ under $\Gamma$. Therefore,

$$Calls(\Gamma, p) \cup Calls(\Gamma, \tilde{e}) = \emptyset \tag{3.1}$$
$$\Rightarrow \mathscr{A}(\Gamma, p) \wedge \mathscr{A}(\Gamma, \tilde{e}), \qquad \text{by the def. of } \mathscr{A} \tag{3.2}$$
$$\Rightarrow p \rightarrow s[e/\mathsf{res}] \text{ under } \Gamma, \quad \text{since } \models_P p \rightarrow s[e/\mathsf{res}] \tag{3.3}$$
$$\Rightarrow \Gamma \vdash p \Downarrow false \vee \Gamma \vdash s[e/\mathsf{res}] \Downarrow true \tag{3.4}$$
By the operational semantics of contract expressions Fig. 2.2,
$$\Rightarrow \exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \{true\}\ e\ \{s\} \Downarrow v \tag{3.5}$$
Since every call-free evaluation terminates in our language
and by Lemma 4,
$$\Gamma \vdash p \Downarrow false \vee \Gamma \vdash p \Downarrow true \tag{3.6}$$
By 3.5 and 3.6, $\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \{p\}\ e\ \{s\} \Downarrow v$ (3.7)

Hence the claim holds in the base case.

*Inductive step:* Assume that the claim holds for all evaluations with $m$ calls. We now show that the claim holds for all evaluations with $m+1$ calls. That is, we need to prove that

$$\forall \Gamma \in \{(H, \sigma, P) \in Env \mid x \in dom(\sigma)\}. \Big(\exists k > m+1, h \in Fids, y \in Vars, \Gamma'. \langle \Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle \Gamma', (h\ y)\rangle\Big)$$
$$\vee \big(\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v\big)$$

As before, let us consider a $\Gamma$ such that $\neg\left(\exists k > m+1, h, y, \Gamma'.\langle\Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle\Gamma', (h\ y)\rangle\right)$. Otherwise the claim trivially holds. That is, all direct calls made by $\tilde{e}$ under $\Gamma$ have depth at most $m+1$. Let $S$ denote the top-level calls made by $\tilde{e}$. These are all calls that appear in the syntax tree of $e$. Formally,

$$S = \{(\Gamma', (g\ x)) \mid \exists i \in \mathbb{N}.\langle\Gamma, \tilde{e}\rangle \rightsquigarrow^i \langle\Gamma', (g\ x)\rangle \wedge \neg\exists j < i, h.(\langle\Gamma, \tilde{e}\rangle \rightsquigarrow^j \langle\Gamma'', (h\ x)\rangle)\} \tag{3.8}$$

Note that by the definition of $\rightsquigarrow$, every call transitively made during the evaluation of $\tilde{e}$ should be reachable (w.r.t $\rightsquigarrow$) from the body of a callee in $S$ in $\leq m$ depth (otherwise $\tilde{e}$ would invoke a call at a depth $> m+1$ violating the assumption). That is,

$$\forall(\Gamma', (g\ y)) \in S \text{ s.t. } \textbf{def } h\ x := \tilde{e} \in P.\neg\left(\exists i > m.\langle\Gamma'[x \mapsto \sigma'(y)], \tilde{e}\rangle \rightsquigarrow^i \langle\Gamma'', (g\ x)\rangle\right)$$

By induction hypothesis the above implies that

$$\begin{aligned}&\forall(\Gamma', (g\ y)) \in S \text{ s.t. } \textbf{def } h\ x := \{pre_h\}\ e_h\ \{post_h\} \in P.\\ &\quad \exists v.\Gamma'[x \mapsto \sigma'(y)] \vdash pre_h \Downarrow false \vee \Gamma'[x \mapsto \sigma'(y)] \vdash \{pre_h\}\ e_h\ \{post_h\} \Downarrow v\end{aligned}$$

Based on the operational semantics and the definition of $pre$, the above can be rewritten as

$$\forall(\Gamma', (g\ y)) \in S.\exists v.\Gamma' \vdash pre(g\ y) \Downarrow false \vee \Gamma' \vdash (g\ y) \Downarrow v \tag{3.9}$$

As a consequence of the above fact we also know that every call invoked inside $pre(g\ y)$ terminates and results in a value. That is,

$$\forall(\Gamma', (g\ y)) \in S.\ \mathscr{A}(\Gamma', pre(g\ y)) \tag{3.10}$$

Now consider the definition of the path condition $path$ of a call $(g\ y)^\ell$ with label $\ell$ contained in the body $\tilde{e}$ of a function $f$. By Lemma 9,

$$\forall\Gamma \in Env.\langle\Gamma, \tilde{e}\rangle \rightsquigarrow^* \langle\Gamma', (g\ y)^\ell\rangle \Rightarrow \Gamma' \vdash path((g\ y)^\ell) \Downarrow true \tag{3.11}$$

$$\Rightarrow \forall(\Gamma', (g\ y)) \in S.\ \Gamma' \vdash path(g\ y) \Downarrow true \tag{3.12}$$

$$\Rightarrow \forall(\Gamma', (g\ y)) \in S.\ \mathscr{A}(\Gamma', path(g\ y)) \tag{3.13}$$

That is, every environment that reaches $(g\ y)$ will satisfy the path condition of $(g\ y)$. We are

given that the following assertion holds:

$$\forall \text{ call-site c in P } . \models_P path(c) \rightarrow pre(c) \tag{3.14}$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \neg\mathscr{A}(\Gamma', path(g\ y)) \vee \neg\mathscr{A}(\Gamma', pre(g\ y))$$

$$\vee\ \Gamma' \vdash path(g\ y) \Downarrow false \vee \Gamma' \vdash pre(g\ y) \Downarrow true \tag{3.15}$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S.\ \Gamma' \vdash pre(g\ y) \Downarrow true, \quad \text{by 3.10, 3.12, 3.13} \tag{3.16}$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in S. \exists v.\ \Gamma' \vdash (g\ y) \Downarrow v, \quad \text{by 3.9} \tag{3.17}$$

$$\Rightarrow \forall (\Gamma', (g\ y)) \in Calls(\Gamma, \tilde{e}). \exists v.\ \Gamma' \vdash (g\ y) \Downarrow v, \text{ by the def. of } Calls \tag{3.18}$$

$$\Rightarrow \mathscr{A}(\Gamma, \tilde{e}) \wedge \mathscr{A}(\Gamma, p) \tag{3.19}$$

Also, 3.18 implies that evaluations of $p$ and $\tilde{e}$ terminates.

As in the base case, the above fact, 3.19 and $\models_P p \rightarrow s[e/\mathsf{res}]$ imply that

$$\exists v. \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \{p\}\ e\ \{s\} \Downarrow v \tag{3.20}$$

□

**Theorem 11** (***Partial correctness of function-level, modular reasoning***)**.** *Let* **def** $f\ x := \tilde{e}$ *where* $\tilde{e} = \{p\}\ e\ \{s\}$ *be a function definition in* $P$*. If the function-level, assume/guarantee obligations given by constraints (2.I) and (2.II) hold for a program* $P$*,* $\forall \Gamma \in Env_{\tilde{e},P}$ *such that there exists no infinite sequence* $\langle \Gamma, \tilde{e}\rangle \rightsquigarrow \langle \Gamma', e'\rangle \rightsquigarrow \cdots$*,* $\exists u.\ \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow u$*.*

*Proof.* Let $\Gamma \in Env_{\tilde{e},P}$. If there exists no infinite sequence $\langle \Gamma, \tilde{e}\rangle \rightsquigarrow \langle \Gamma', e'\rangle \rightsquigarrow \cdots$, then there exists a $n \in \mathbb{N}$ such that $\neg\left(\exists k > n, e, \Gamma'. \langle \Gamma, \tilde{e}\rangle \rightsquigarrow^k \langle \Gamma', e\rangle\right)$. We know that $\Gamma \in Env_{\tilde{e},P}$ implies that $x \in dom(\sigma)$ (Lemma 2). Hence, by Lemma 10, $\exists u.\ \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow u$. □

### 3.2.2 Function-level Modular Reasoning with Templates

The function-level assume-guarantee reasoning presented above can be extended to programs where the contracts of functions have templates. In this case, the goal is to find an assignment $\iota$ for holes such that the program obtained by substituting the holes with its image in $\iota$ satisfy the assume-guarantee constraints (2.I) and (2.II). Let $P\ \iota$ be program in which every hole $a$ in the expressions of the program is replaced by $\iota(a)$. The assume-guarantee obligations for a program $P$ with templates is given by:

$\exists \iota : TVars \mapsto \mathbb{N}$ such that for each function definition **def** $f\ x := \{pre\}\ e\ \{post\}$ in the program $P\ \iota$, the constraints (2.I) and (2.II) hold.

## 3.3 Template Solving Algorithm

In this section, I describe the algorithm for inferring values for the holes that will make the modular assume-guarantee obligations hold. Figure 3.7 pictorially depicts the algorithm in the

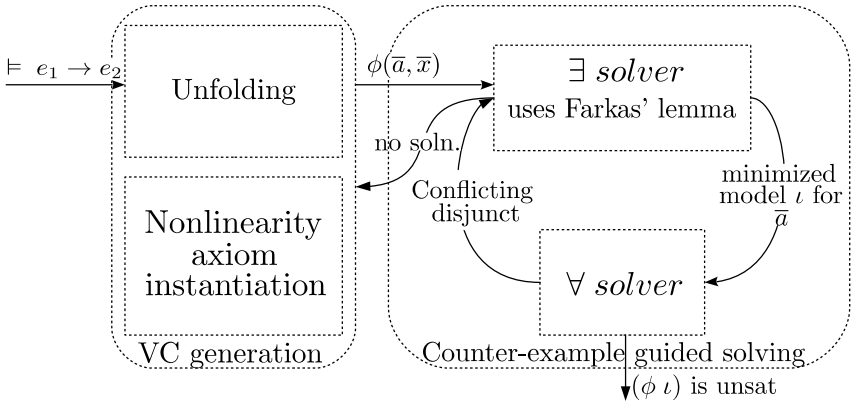



Figure 3.7 – Counter-example guided inference for numerical holes.

form of a block diagram. As a first step, a verification condition is generated from the assume-guarantee obligations (described in section 3.3.1), which is a formula of the form $\exists.\forall.\psi$ or equivalently $\exists.\forall.\neg\phi$, where $\phi \equiv \neg\psi$. The verification conditions are such that their validity implies that the assume-guarantee obligation holds. In the next step, using a counterexample driven procedure solveUNSAT (section 3.3.4), the existence of an assignment $\iota : \mathit{TVars} \mapsto \mathbb{N}$ that will make $\phi\ \iota$ unsatisfiable is checked. If the procedure solveUNSAT fails to infer a value for the holes, the VCs are refined by unfolding functions as described in section 3.3.2, and the process is repeated until a solution is found or a timeout is reached.

Given a formula $\phi(\bar{a}, \bar{x})$, where $\bar{a} \in \mathit{TVars}$, the procedure solveUNSAT discovers a solution for $\bar{a}$ that will make $\phi$ unsatisfiable using an iterative but terminating algorithm that progresses in two phases: an existential solving phase (phase I), and a universal solving phase (phase II). Phase I discovers candidate assignments $\iota$ for the free variables $\bar{a}$. It initially starts with an arbitrary guess, and subsequently refines it based on the counterexamples produced by Phase II. Phase II checks if the candidate assignment $\iota$ makes $\phi$ unsatisfiable. That is, if $\phi\ \iota$ is unsatisfiable. If not, it chooses a disjunct $d(\bar{x}, \bar{a})$ satisfiable under $\iota$. The disjunct is converted to a weaker disjunct that has only numerical variables by axiomatizing uninterpreted functions and algebraic datatypes in a complete way. This numerical disjunct is then given back to phase I. Phase I generates and solves a quantifier-free nonlinear constraint $C(\bar{a})$, based on Farkas' Lemma [Colón et al., 2003], to obtain the next candidate assignment for $\bar{a}$ that will make $d(\bar{x}, \bar{a})$ and other disjuncts previously seen unsatisfiable. Each phase invokes the Z3 [de Moura and Bjørner, 2008] and CVC4 [Barrett et al., 2011] SMT solvers in portfolio mode on quantifier-free formulas.

The procedure solveUNSAT is sound and complete if the formula $\phi$ belongs to the combined theory of uninterpreted functions, algebraic datatypes, sets and real arithmetic, and if $\phi$ is a *linear parametric formula.* These are formulas in which every nonlinear term is of the form $a \cdot x$ for some $a \in \mathit{TVars}$ and $x \notin \mathit{TVars}$. The detailed proof of soundness and completeness of the solveUNSAT is discussed in section 3.4. Any non-linearity in the VC between variables not

in *TVars* is eliminated before providing it to the solveUNSAT algorithm. This is represented by the block *Nonlinearity axiom instantiation* in Figure 3.7 and described in section 3.5. With this overview I now explain the individual components of the algorithm in detail.

### 3.3.1 Verification Condition Generation

This section discusses the translation of the assume-guarantee obligations to a logical formula called verification condition (VC) that uses only a restricted set of theories typically supported by SMT solvers, and can be handled by the solveUNSAT procedure detailed in section 3.3.4. The key ideas of the encoding presented here is based on the algorithm used by the LEON verifier [Blanc et al., 2013, Suter, 2012]. However, an important difference is that the VC in our case has holes, which become existentially quantified variables of the VC. Below I present an overview of the VC generation algorithm using the list-reversal example shown in Figure 3.2.

At a high-level, the logical encoding of a predicate belonging to a first-order fragment of our language is straightforward for most constructs except function calls, since most (first-order) operations in our language have a direct mapping to a suitable theory operation supported by SMT solvers. For instance, primitives types such as Int and Bool map to the integer and boolean sorts. User-defined datatypes map to algebraic datatypes. If-else and Match expressions correspond to disjunctions, and let expressions to equalities. However, there are two non-trivial aspects to VC generation algorithm used by our system: (a) *Clausification* of predicates using control literals, and (b) encoding of recursive functions as uninterpreted functions through unfolding. The former aspect is used to efficiently obtain a satisfiable disjunct from the VC given a satisfying assignment for the variables of the VC, which is used by the algorithm for inferring holes (section 3.3). The latter aspect enables using abstractions of recursive functions which can be refined (i.e, made more precise) on demand.

Consider the instrumented list reversal program presented in Figure 3.2(b). It is shown below in the syntax of the core language. (The variables *a* and *b* are template variables in *TVars* and correspond to the ? in the Scala programs.)

```
def revRec(l1,l2) =
                                { true }
                                 l1 match {
                                  case Nil() ⇒ (l2, 2);
                                  case Cons(x,xs) ⇒
                                    let nl := Cons(x,l2) in
                                      let r := revRec(xs, nl) in
                                        let u := 7 + r.2
                                          (r.1, u);
                                 }
                                { res ⇒ let w := size(l1) in res.2 ≤ a*u + b}
```

Consider the assume-guarantee obligations that would have to be established for this function:

$\models_P true \rightarrow post(\text{revRec})[body(\text{revRec})/\text{res}]$. Note that since the precondition is true there is no obligation generated for the precondition. For now, consider the holes $a$ and $b$ as some constants. Expanding the body and postcondition of the function, the assume-guarantee obligation reduces to $\models_P e_{post}$, where $e_{post}$ is defined as follows:

$$
\begin{array}{l}
e_{post} \triangleq \quad (\text{l1 } \textbf{match } \{ \\
\qquad \textbf{case } \text{Nil}() \Rightarrow (\text{l2}, 2); \\
\qquad \textbf{case } \text{Cons}(\text{x},\text{xs}) \Rightarrow \\
\qquad\quad \textbf{let } \text{nl} := \text{Cons}(\text{x},\text{l2}) \text{ in } \textbf{let } r := \text{revRec}(\text{xs}, \text{nl}) \text{ in} \\
\qquad\qquad \textbf{let } u := 7 + r.2 \text{ in } (r.1, u) \\
\quad \}).2 \leq (\textbf{let } w := \text{size}(\text{l1}) \text{ in } a \cdot w + b)
\end{array}
$$

The predicate $e_{post}$ is converted to a predicate $e_{norm}$ that in a normal form wherein match constructs are reduced to disjunctions and let expressions to equalities. This is accomplished by a transformation that introduces new boolean variables called *control literals*, denoted $b_i$, at the points of disjunctions in the expressions. Each control literal corresponds to a disjunction-free segment of the expression. For instance, the normal form for the predicate $e_{post}$ is shown below.

$$
\begin{array}{ll}
e_{norm} \triangleq & b_1 \implies \text{l1} = \text{Nil}() \wedge t = (\text{l2}, 2) \\
& \wedge (b_2 \implies \text{l1} = \text{Cons}(\text{x},\text{xs}) \wedge \text{nl} = \text{Cons}(\text{x},\text{l2}) \wedge r = \text{revRec}(\text{xs}, \text{nl}) \wedge u = 7 + r.2 \wedge t = (r.1, u) \\
& \wedge (b_3 \implies w = \text{size}(\text{l1})) \\
& \wedge (b_4 \implies t.2 \leq a \cdot w + b) \\
& \wedge (b_1 \vee b_2) \wedge b_3 \implies b_4
\end{array}
$$

For any given values of the holes $a$ and $b$, the predicates $e_{norm}$ and $e_{post}$ are equivalid i.e, whenever one of the predicates is true for all environments (that have a binding for all free variables excluding holes), the other predicate is also true for all of its corresponding environments. In other words, $\models_P e_{post}$ is equivalent to $\models_P e_{norm}$. (Notice that all let binders, irrespective of whether they appear in postcondition or in the body, become a part of the antecedent e.g. notice that the control literal $b_3$ is a part of the antecedent.)

The above transformation is closely related to the linear-time conversion to conjunctive normal form (CNF) commonly referred to as *clausification* or Tseitin encoding [Tseitin, 1968]. But, there the goal is to generate an equisatisfiable formula whereas here it suffices to preserve validity of formulas. In fact, the formula $e_{norm}$ is almost in CNF where each clause in the formula $b_i \implies A_1 \wedge \cdots \wedge A_n$ corresponds to $n$ CNF clauses $(\neg b_i \vee A_1) \wedge \cdots \wedge (\neg b_i \vee A_n)$. Notice that a path exercised during an evaluation of $e_{norm}$ under an environment $\Gamma$ will have the control literals corresponding to the disjuncts of the path evaluate to *true*.

By the definition of $\models_P$ (see section 3.2) , the contracts of the function calls in the assume-guarantee predicates can be assumed to hold for the environments under which they are invoked. This implies that the postconditions of the recursive call revRec(xs, nl) can be assumed in the assume-guarantee obligation $\models_P e_{norm}$. Therefore, we have $\models_P e_{norm} \equiv \models_P \psi$

where

$$\psi \triangleq \quad e_{norm} \wedge b_2 \implies z = \mathsf{size}(\mathsf{xs}) \wedge r.2 \leq a \cdot z + b$$

Note that the clause conjoined with the predicate $e_{norm}$ corresponds to the postcondition of the recursive call revRec(xs, nl). The clause is guarded by the same control literal $b_2$ under which the recursive call is invoked. This ensures that the contract is assumed only along the path in which the recursive call happens.

While the predicate $e_{norm}$ does not have match or let constructs, it still has recursive functions such as size and recRec. Say we treat the functions in the predicate $\psi$ as uninterpreted i.e, they can return any value as long as they return equal values for equal arguments. Under this interpretation, the predicate $\psi$ (as well as $e_{norm}$) can be interpreted as a logical formula in the theory $\mathcal{T}$ of uninterpreted functions, algebraic datatypes and integer arithmetic. Every environment $\Gamma$ that has a binding for the free variables of the predicate can be seen as an assignment $\sigma_\Gamma$ of *ground terms* belonging to the theory $\mathcal{T}$ to the free variables of the formula $\psi$. The predicate will evaluate to true under an environment $\Gamma$ iff it is *satisfiable* under the assignment $\sigma_\Gamma$. Therefore, if the predicate $\psi$ is $\mathcal{T}$-valid then it implies that $\models_P \psi$ holds. (The converse does not hold due the approximation of functions as uninterpreted.)

**From Validity to Unsatisfiability** The formula $\psi$ presented above needs to be checked for validity. However typically SMT solvers are well tuned for establishing satisfiability and unsatisfiability of formulas. Therefore, we phrase the above problems as checking unsatisfiability by negating the formula $\psi$. This negation can be performed without destroying the normal form by preserving the clauses that define control literals, since these clauses appear in the antecedent of an implication. For instance, the negated formula $\phi = \neg\psi$ is shown below:

$$\begin{aligned}
\phi \triangleq \quad & b_1 \implies \mathsf{l1} = \mathsf{Nil}() \wedge t = (\mathsf{l2},\ 2) \\
& \wedge (b_2 \implies \mathsf{l1} = \mathsf{Cons}(\mathsf{x},\mathsf{xs}) \wedge \mathsf{nl} = \mathsf{Cons}(\mathsf{x},\mathsf{l2}) \wedge r = \mathsf{revRec}(\mathsf{xs},\ \mathsf{nl}) \wedge u = 7 + r.2 \wedge t = (r.1, u) \\
& \wedge (b_2 \implies z = \mathsf{size}(\mathsf{xs}) \wedge r.2 \leq a \cdot z + b) \\
& \wedge (b_3 \implies w = \mathsf{size}(\mathsf{l1})) \\
& \wedge (b_4 \implies t.2 > a \cdot w + b) \\
& \wedge (b_1 \vee b_2) \wedge b_3 \wedge b_4
\end{aligned}$$

Notice the change of $\leq$ in clause guarded by $b_4$ to $>$ and that $b_4$ is now cojoined in the root clause. Though this may seem somewhat magical, the reason why most of the formula remain intact is because of the simple fact that negation of an implication $A \implies B$ is $A \wedge \neg B$. It requires no change to $A$, which forms the bulk of the formula $\psi$. It is easy to see that $\models_P \psi$ is equivalent to $\models_P \neg\phi$. Actually, in our implementation, the initial expression $e_{post}$ itself is negated before it is converted to the normal form.

**Parametric Unsatisfiability** Consider now the holes in the predicate $\psi$, which were so far treated as some constants. The above reasoning shows that a substitution for the holes that will make the formula $\psi$ valid (or equivalently $\phi$ unsatisfiable), will also make the postconditions of functions satisfy the assume-guarantee obligations. Therefore, such an assignment is a valid solution for the resource inference problem (see section 3.2.2), given that the holes cannot be shared across function definitions. Thus, the actual goal is to find an assignment $\iota : \mathit{TVars} \mapsto \mathbb{N}$ for the holes such that $\phi\,\iota$ is unsatisfiable. This can be seen as deciding the formula $\exists \bar{a}.\forall \bar{x}.\neg\phi$ under a theory $\mathcal{T}$, where the universal quantified variables $\bar{x}$ consist of variables and function symbols in $\phi$, and the existentially quantified variables consist of integer valued holes. Such a formula is referred to as the verification condition (VC).

Note that since $\phi$ treats functions as uninterpreted, the unsatisfiability of $\phi\,\iota$ is a stronger condition than what is required for verifying the assume-guarantee obligation, where the functions are defined by the semantics of their bodies. In the sequel, I discuss how the verification condition can be refined to create better approximations of the recursive functions.

### 3.3.2 Successive Function Approximation by Unfolding

In our approach, VCs are constructed incrementally wherein each increment makes the VC more precise by unfolding the function calls that have not been unfolded in the earlier increments. This process is referred to as *VC refinement*. The functions in the VCs at any given step are treated as uninterpreted functions. Hence, every VC created is a sufficient but not necessary condition for the corresponding assume-guarantee obligation to hold.

The VC refinements happen on demand if the current VC cannot be solved by the inference algorithm (discussed in section 3.3). For instance, consider the formula $\phi$ presented above and say there does not exist an $\iota : \mathit{TVars} \mapsto \mathbb{N}$ such that $\phi\,\iota$ is $\mathcal{T}$-unsatisfiable. Now the next VC refinement will create a formula $\phi'$ by unfolding the calls to functions size and revRec in $\phi$. For instance, unfolding the call size(l1) in $\phi$ would conjoin the following predicates to $\phi$.

$$
\begin{aligned}
&(b_3 \implies (b_4 \vee b_5))\\
&\wedge (b_4 \implies \mathsf{l1} = \mathsf{Nil}() \wedge w = 0)\\
&\wedge (b_5 \implies \mathsf{l1} = \mathsf{Cons}(\mathsf{x},\mathsf{xs}) \wedge p = \mathsf{size}(\mathsf{xs}) \wedge w = 1 + p
\end{aligned}
$$

As before, the assume-guarantee reasoning permits assuming the contracts of any function call that is introduced by the unfolding of a call in the VC, e.g. the function size(xs) in the above clauses, provided the assume-guarantee obligations of the callee are also verified. This in essence provides a $K$-inductive reasoning for a function that is unfolded $K$ times. The refinement process stops if and once a VC is solved. While the unfolding of VC enhances the capabilities of the verification technique, it quite evidently introduces a major performance bottleneck because of the potential blow-up that may result due to unfolding. In the later sections, I describe several optimizations that are used to control this blow up. The most important optimization is *model-driven unfolding* which only unfolds along specific disjuncts

that could not be solved in the earlier iterations.

### 3.3.3 Logic Notations and Terminology

Before I present the solveUNSAT procedure I introduce a few notations and helper functions for manipulating logical formulas.

**Many Sorted First-order Theory** Let $\Sigma$ be a signature consisting of a set of sorts $S$, constants $C$, function symbols $F$ and predicate symbols $P$. Let $Ids$ denote the set of logical variables or identifiers. Each constant or variable has an associated sort. The set of constants or variables with a sort $\sigma$ is denoted as $C_\sigma$ and $Ids_\sigma$ respectively. Each function and predicate symbol has an associated arity $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ and $\sigma_1 \times \cdots \times \sigma_n$ respectively.

The set of $\Sigma$-terms of sort $\sigma$ is the smallest set constructed as follows: (a) Every variable and constant of sort $\sigma$ belong to $\Sigma$-terms with sort $\sigma$, (b) If $f$ is a function symbol in $\Sigma$ of arity $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ and, for all $i = 1, \cdots, n$, $t_i$ is a $\Sigma$-term of sort $\sigma_i$, then $f(t_1, \cdots, t_n)$ belongs to $\Sigma$-terms of sort $\sigma$. A $\Sigma$-atom (also called as a literal) is a term of the form $p(t_1, \cdots, t_n)$ or $\neg p(t_1, \cdots, t_n)$, where $p$ is a predicate symbol of arity $\sigma_1 \times \cdots \times \sigma_n$ and each $t_i$ is a $\Sigma$-term of sort $\sigma_i$. A $\Sigma$-formula $\phi$ is a first-order formula constructed using the $\Sigma$-atom and the usual logical operators: $\wedge$, $\vee$ and $\neg$ (and also derived operations $\Longrightarrow$, $\Longleftrightarrow$), and quantifiers $\forall$ and $\exists$. Let $FV(\phi)$ denote the set of free variables in a formula $\phi$, and $FV_\sigma(\phi)$ denote the free variables of sort $\sigma$.

A many sorted, first-order theory $\mathscr{T} = (\Sigma, \mathscr{A})$ is a pair of signature $\Sigma$ and axioms $\mathscr{A}$, where $\mathscr{A}$ is a set of closed $\Sigma$-formulas that do not have any free variables. The axioms $\mathscr{A}$ assign a meaning for the function and predicate symbols in $\mathscr{T}$. One theory of particular interest here is the combined theory of real arithmetic, uninterpreted functions and algebraic datatypes (ADTs) [Barrett et al., 2007, Zhang et al., 2004]. A $\mathscr{T}$-interpretation is a map that maps each sort $\sigma$ to a non-empty domain $A_\sigma$, variables and constants of sort $\sigma$ to elements of $A_\sigma$, and function and predicate symbols of $\Sigma$ of arity $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$ to functions and predicates over $A_{\sigma_1} \times \cdots \times A_{\sigma_n} \rightarrow A_\sigma$, such that the assignment satisfies the axioms of the theory: $\mathscr{A}$. A $\Sigma$-formula $\phi$ is satisfiable iff there exists a $\mathscr{T}$-interpretation $I$ of variables, function and predicate symbols under which $\phi$ evaluates to true under the usual semantics for $\wedge$, $\vee$ and $\neg$. This is denoted by $I \models \phi$. A formula $\phi$ is $\mathscr{T}$-valid, denoted $\models_{\mathscr{T}} \phi$ iff every $\mathscr{T}$-interpretation satisfies the formula $\phi$. A formula $\phi$ is $\mathscr{T}$-unsatisfiable, denoted $\not\models \phi$ iff $\phi$ is not satisfiable or, equivalently, $\neg\phi$ is $\mathscr{T}$-valid. The prefix $\mathscr{T}$- and $\Sigma$- are omitted in the above nomenclature if the theory under consideration is clear from the context.

A $\Sigma$-term is ground if it is quantifier-free and closed i.e, does not have free variables. Not all ground terms belonging to a signature need to be allowed by a theory. A set of canonical ground terms belonging to a theory $\mathscr{T}$ is a subset of ground terms belonging to the signature of the theory that satisfy the axioms of the theory. Let $Val_{\mathscr{T}}$ denote the set of ground

terms of a theory $\mathcal{T}$. For instance, for the theory of real arithmetic, uninterpreted functions and algebraic datatypes (ADTs), the canonical ground terms consist of real numbers and constructor symbols applied over other ground terms. That is, $Val_{\mathcal{T}} = \mathbb{R} \cup cons(Val^*_{\mathcal{T}})$ where *cons* denotes a constructor symbol. A substitution $\alpha : (Ids \cup F) \mapsto (Val_{\mathcal{T}} \cup (Val_{\mathcal{T}} \rightarrow Val_{\mathcal{T}}))$ is a map from variables (and function symbols) to canonical ground terms (and functions over canonical ground terms) such that each variable of sort $\sigma$ is bound to a term of sort $\sigma$. Typically, the assignments provided by SMT solvers for a satisfiable formula bind the variables in the formula to these canonical ground terms defined by the underlying theory.

Given a substitution $\alpha$, let $\phi\ \alpha$ denote the formula obtained by replacing every *variable* $x \in dom(\alpha)$ by $\alpha(x)$, and function term $f(t_1, \cdots, t_n)$ by $\alpha(f)(\alpha(t_1), \cdots, \alpha(t_n))$. Given a sequence of variables $\bar{x} : (x_1, \cdots, x_n)$, let $\hat{\alpha}(\bar{x})$ denote the point wise application of $\alpha$ on the elements of the sequence i.e, $(\alpha(x_1), \cdots, \alpha(x_n))$. A substitution $\alpha$ is a *satisfying assignment* of a formula $\phi$ with respect to a theory $\mathcal{T}$ iff $\phi\ \alpha$ is ground and satisfiable under $\mathcal{T}$. A satisfying assignment is isomorphic to an interpretation that satisfies $\phi$. Thus, a satisfying assignment is also referred to as a model. The overloaded representation $\alpha \models \phi$ is used to denote that $\alpha$ is a satisfying assignment of $\phi$.

**Conjunctive Normal Form and Disjuncts** A formula $\phi$ is in conjunctive normal form (CNF) if it is of the form $\phi = \bigwedge_{j=1}^{m} C_j$ and $C_j : \bigvee_{i=1}^{n_j} a_{ij}$, where $a_{ij}$ is an atom. (Note that an atom is either a predicate or its negation.) Let the term *disjunct* refer to a conjunction of atoms and the term *clause* to refer to a disjunction of atoms. Given a formula $\phi$ in CNF, we say that $d$ is a *disjunct of* $\phi$ if $d$ is a disjunct and for every clause in the formula $\phi$, $d$ contains one atom belonging the clause. That is, $d$ is of the form $a_{i_1 1} \wedge a_{i_2 2} \wedge \cdots \wedge a_{i_m m}$ for some $i_1, i_2, \cdots, i_m$. Let $d \in disjunct(\phi)$ denote that $d$ is a disjunct of $\phi$.

For convenience, *disjuncts* are treated as sets of atoms. Given disjuncts $R$ and $S$, $x \in R$ denotes that $x$ is an atom of $R$, $R \subseteq S$ denotes that every atom of $R$ is present in $S$, $R \cup S$ denotes the *conjunction* of atoms in $R$ and $S$, and $(R \setminus S)$ denotes the disjunct obtained by dropping the atoms in $S$ from $R$. Note that these operations are syntactic operations on disjuncts.

In the sequel, it is assumed that all formulas are in CNF form. Let $[m..n]$ denote a closed integer interval from $m$ to $n$.

### 3.3.4 The solveUNSAT procedure

Figure 3.8 presents our algorithm for solving an alternating satisfiability problem. Given a formula $\phi$ whose free variables are a superset of holes, the goal is to find an substitution $\iota$ for holes such that replacing holes according to $\iota$ results in unsatisfiable formula. This algorithm relates to the block diagram shown in Figure 3.7 as follows. The lines 3, 4, and 15 – 21 correspond to the Phase I: $\exists$-solver that finds a candidate assignment for the holes. The lines 6–14 correspond to the Phase II: $\forall$-solver that constructs a numerical counterexample disjunct for

**input:** A formula $\phi$ and a set of variables holes $\subseteq FV(\phi)$
**output:** A substitution for holes such that ($\phi$ holes) is unsatisfiable
or $\emptyset$ if no such substitution can be found

```
1  def solveUNSAT(holes, φ) {
2    purify φ
3    construct an arbitrary initial mapping ι : holes ↦ ℝ
4    var C = true
5    while(true) {
6      let φ_inst be obtained from φ by replacing every t ∈ holes by ι(t)
7      if (φ_inst is unsatisfiable) return ι
8      else {
9          choose α such that α ⊨ φ_inst
10         let α' be ι ⊎ α
11         choose a disjunct d of φ such that α' ⊨ d
12         let δ be elimFunctions(d)
13         choose a disjunct d' of δ such that α' ⊨ d'
14         let d_num be elim(d')
15         let C_d be unsatConstraints(d_num)
16         C = C ∧ C_d
17         if (C is unsatisfiable) return ∅
18         else {
19           choose m such that m ⊨ C
20           let ι be the projection of m onto holes
21         }
22     }
23   }
24 }
```

Figure 3.8 – The solveUNSAT procedure

the candidate assignment to the holes. The **while** loop corresponds to repeating the phases until the algorithm finds a solution or fails. I now explain the algorithm in detail by illustrating it on the formula $\phi$ presented in the earlier section.

**Purification** The algorithm first *purifies* the formula $\phi$ which expresses every atom referring to uninterpreted functions or ADTs in the form $r = f(v_1, v_2, \ldots, v_n)$ or $r = cons(v_1, v_2, \ldots, v_n)$ where $f$ is a function symbol, *cons* is the constructor of an ADT and $r, v_1, \ldots, v_n$ are variables. (Due to the normalization that is performed during VC generation, every function and constructor application in the formula $\phi$ is already in this form.) The atoms with ADT selectors in the formula such as $t.2 > a \cdot w + b$ are converted to constructor applications by introducing free variables e.g. $t = cons(s_1, s_2) \wedge s_2 > a \cdot w + b$. Note that the introduction of free variables preserves the unsatisfiability of the formula. Hence, the purified formula is unsatisfiable iff the original formula is.

**Choosing a Satisfiable Disjunct** Initially, the algorithm starts with some arbitrary assignment $\iota$ for the holes $a$ and $b$ (line 3 of the algorithm). Say $\iota(a) = \iota(b) = 0$ initially. Next, it computes the formula $\phi\,\iota$ by replacing $a$ and $b$ by 0 (line 6), which results in a formula without holes. In particular, in $\phi\,\iota$ the clause guarded by $b_4$ and $b_2$ would become $b_2 \implies z = \mathsf{size}(\mathsf{xs}) \wedge r = (r_1, r_2) \wedge r_2 \leq 0$ and $b_4 \implies t = (t_1, t_2) \wedge t_2 > 0$. Here, $r = (r_1, r_2)$ and $t = (t_1, t_2)$ are introduced by purification. (Recall that tuples are also ADTs.)

If the formula becomes unsatisfiable because of the substitution then a solution has been found, so the algorithm returns. Otherwise, a satisfying assignment $\alpha$ is constructed for the instantiated formula as shown in line 9. In the next step, we combine the substitutions $\iota$ and $\alpha$ and construct $\alpha'$. Note that $\iota$ is a substitution for holes and $\alpha$ is a substitution for other variables. Hence $\alpha'$ is a satisfying assignment for $\phi$. Using the assignment $\alpha'$ the algorithm chooses a disjunct of the formula $\phi$ that is satisfied by $\alpha'$. For the example, the disjunct chosen could be

$$d_{ex} : b_1 \wedge \mathsf{l1} = \mathsf{Nil}() \wedge t = (\mathsf{l2}, 2) \wedge w = \mathsf{size}(\mathsf{l1}) \wedge t = (t_1, t_2) \wedge t_2 > a \cdot w + b$$

This operation of choosing a disjunct that is true under a satisfying assignment can be performed efficiently in time linear in the size of the formula by using the values of the control literals in the assignment $\alpha'$. The algorithm then invokes the function elimFunctions on the disjunct $d$ (line 12), which eliminates the function symbols and ADT constructors from the disjunct $d$, as explained below.

**Eliminating Function Symbols and ADT Operations** Let $d$ be a disjunct with holes holes defined over a set of variables and function symbols. This is reduced to a formula $\delta$ that does not have any uninterpreted functions and ADT constructors using the axioms of uninterpreted functions and ADTs as described below. Let $F$ and $T$ be the set of atoms with function applications and constructor applications in the purified formula. The eliminated formula $\delta$ is defined as follows:

$$\begin{array}{ll} & \text{let } \delta_1 = \bigwedge \left\{ (\bigwedge_{i=1}^{n} v_i = u_i) \Rightarrow (r = r') \mid \{r = f(v_1, \ldots, v_n), r' = f(u_1, \ldots, u_n)\} \subseteq F \right\} \text{ in} \\ \delta \triangleq & \text{let } \delta_2 = \bigwedge \left\{ (\bigwedge_{i=1}^{n} v_i = u_i) \Leftrightarrow (r = r') \mid \{r = cons(v_1, \ldots, v_n), r' = cons(u_1, \ldots, u_n)\} \subseteq T \right\} \text{ in} \\ & (d \setminus (F \cup T)) \wedge \delta_1 \wedge \delta_2 \end{array}$$

The notation $\delta \setminus (F \cup T)$ denotes a formula obtained by removing i.e, substituting with true every atomic predicate in $F$ or $T$. Notice that the above elimination procedure uses only the fact that the ADT constructors are *injective*, while the theory of ADTs satisfy more axioms of ADT. Due to this it may appear that this process introduces incompleteness in our approach. But somewhat counter-intuitively this is complete. In section 3.4, I establish this non-trivial completeness property of the approach.

Applying the above reduction to the disjunct $d_{ex}$ shown above results in a constraint of the

form sketched below. (Note that tuples are ADTs.)

$$\delta_{ex} = b_1 \wedge t_2 > a \cdot w + b \wedge ((|2 = t_1 \wedge t_2 = 2) \Leftrightarrow t = t))$$

The formula $\delta$ obtained by eliminating uninterpreted function symbols and ADTs typically has several disjunctions. In fact, if there are $n$ function symbols and ADT constructors in $d$ then $\delta$ could potentially have $O(n^2)$ disjunctions and $O(2^{n^2})$ disjuncts. However, the algorithm only explores the disjuncts of this formula on demand as explained in the sequel.

**Choosing a Disjunct from elimFunctions($d$)** Having constructed a formula $\delta$, the algorithm then chooses a disjunct $d'$ of $\delta$ that is true under the satisfying assignment $\alpha'$. There has to exist such a disjunct since $\delta'$ is weaker than $d$. The following constructive approach described below shows how the disjunct is chosen by our algorithm. This is important from the perspective of proving completeness. Let $d$ be a disjunct and $\rho \models d$. Define an operation chooseEF($d, \rho$) that chooses a disjunct $d'$ of elimFunctions($d$) such that $\rho \models d'$. By definition, elimFunctions($d$) has three parts: $(d \setminus (F \cup T)) \wedge \delta_1 \wedge \delta_2$. Let $d_n$ denote $(d \setminus (F \cup T))$. All atoms in $d_n$ are added to $d'$, i.e, $d_n \subseteq d'$.

Let $r = f(v_1, \cdots, v_n)$ and $r' = f(u_1, \cdots, u_n)$ be two atoms in $d$ and let $\psi$ denote $(\bigwedge_{i=1}^{n} v_i = u_i) \Rightarrow (r = r')$. $\psi$ is a clause i.e, a disjunction of atoms. An atom of $\psi$ that is satisfied by $\rho$ is chosen as follows. (a) If $\exists i.\ \rho(v_i) \neq \rho(u_i)$, choose $v_i = u_i$. (b) Otherwise, if $\forall i.\ \rho(v_i) = \rho(u_i)$, the following must hold $\rho(r) = \rho(r')$ since $\rho \models d$. Therefore, choose $r = r'$.

For every clause $\psi$ in $\delta_1$ or $\delta_2$, choose an atom of $\psi$ as described above and add it to $d'$. By construction, $\rho \models d'$. $d'$ is a disjunct of elimFunctions($d$) as we have chosen an atom from every clause of elimFunctions($d$).

For the running example, chooseEF($d, \alpha'$) will return the disjunct

$$d'_{ex} : b_1 \wedge |2 = t_1 \wedge t_2 = 2 \wedge t = t \wedge t_2 > a \cdot w + b$$

**Eliminating Non-numerical Predicates from a Disjunct (elim)** The disjunct $d'$ may contain numerical operations like $t_2 = 2$, boolean predicates such as $b_1$ and also equalities or disequalities between variables of ADT sort like $|2 = t_1$. The operation elim at line 14, produces a disjunct that has only numerical operations. Let $d_t$ denote the atoms consists of variables of non-numeric sort. Let $d_n$ denote the atoms that do not contain any holes and only contain variables of numerical sort, and let $d_p$ denote the remaining (numerical) atoms that has holes and numerical variables.

For the example disjunct $d'_{ex}$, $d_t$ is $|2 = t_1$, $d_n$ is $t_2 = 2$ and $d_p$ is $t_2 > a \cdot w + b$. The disjunct $d_t$ can be dropped as $d_t$ cannot be falsified by any instantiation of the holes. This is because $d_p$ and $d_t$ will have no common variables. The remaining disjunct $d_n \wedge d_p$ is completely

numerical. However, $d_n \wedge d_p$ is simplified further as explained below. Our algorithm constructs a formula $d'_n$ by eliminating variables in $d_n$ that do not appear in $d_p$ by applying the quantifier elimination rules of Presburger arithmetic on $d_n$ [Oppen, 1973]. In particular, the algorithm applies the one-point rule that uses equalities to eliminate variables and the rule that eliminates relations over variables for which only upper or lower bounds exist. $d_n \wedge d_p$ is unsatisfiable *iff* $d'_n \wedge d_p$ is unsatisfiable.

Typically, $d_n$ has several variables that do not appear in $d_p$. This elimination helps reduce the sizes of the disjuncts and in turn the sizes of the nonlinear constraints generated from the disjunct. Our experiments indicate that the sizes of the disjuncts are reduced by 70% or more by this step.

**Solving Numerical Disjunct with Holes** Finally, the disjunct $d_{num}$ obtained at line 14 of the algorithm is a purely numerical disjunct but it has holes. The goal is to find a assignment for the holes that will falsify the disjunct. For this purpose, the algorithm uses a well know approach for solving templates in numerical programs which is based on Farkas' Lemma [Colón et al., 2003, Cousot, 2005b, Gulwani et al., 2008]. This approach reduces the problem for finding a value of the holes in $d_{num}$ to that of satisfying a quantifier-free nonlinear constraint (Farkas' constraint). This reduction is sketched below.

The approach is based on the fact that a conjunction of linear inequalities is unsatisfiable if one can derive a contradiction $1 \leq 0$ by performing the following three operations: (a) multiplying the inequalities by non-negative values, (b) subtracting the smaller terms in the inequalities by non-negative values and (c) adding the coefficients in the inequalities. E.g, $ax + by + c \leq 0 \wedge x - 1 \leq 0$ is unsatisfiable if there exist non-negative real numbers $\mu_0, \mu_1, \mu_2$ such that $\mu_1 \cdot (ax + by + c) + \mu_2 \cdot (x - 1) - \mu_0 \leq 0$ reduces to $1 \leq 0$. Hence, the coefficients of $x$ and $y$ should become 0 and the constant term should become 1. This yields a nonlinear constraint $\mu_1 a + \mu_2 = 0 \wedge \mu_1 b = 0 \wedge \mu_1 c - \mu_2 - \mu_0 = 1 \wedge \mu_0 \geq 0 \wedge \mu_1 \geq 0 \wedge \mu_2 \geq 0$. The values of $a$ and $b$ in every satisfying assignment for this nonlinear constraint will make the original inequalities unsatisfiable.

There are two important points to note about this approach. Firstly, handling strict inequalities in the presence of real valued variables requires an extension based on *Motzkin's transposition theorem* as explained by Rybalchenko and Sofronie-Stokkermans [2007]. This extension is needed in our case since the holes in the VCs are encoded as real-valued variables and the VCs involve strict and non-strict inequalities. Moreover, in this setting, it is not possible to reduce strict inequalities to non-strict inequalities using an integer reasoning. For instance, consider a parametric VC: $(res = 0 \vee (res = r + 1 \wedge ar + b \leq 0)) \wedge ares + b > 0$ with parameters $a$ and $b$. The assignment $a = -1, b = 0$ will make the VC false and hence is a valid solution, whereas the assignment $a = -1, b = 1/2$ is not a valid solution. However, converting the strict inequality in the VC to a non-strict inequality using an integer reasoning to a formula: $(res = 0 \vee (res = r + 1 \wedge ar + b \leq 0)) \wedge ares + b - 1 \geq 0$ is unsound since $a = -1, b = 1/2$ falsifies

the new formula and hence is a valid solution to the new formula. Therefore, it is necessary to preserve the non-strict inequality in the formula and use Motzkin's transposition theorem.

Another important aspect of using this Farkas'-lemma-based approach is that it is complete only for linear real formulas but not for linear integer formulas. However, the incompleteness did not manifest in any of our evaluations of the system on practical benchmarks. More precisely, there wasn't a benchmark used in the experiments for which the inference algorithm failed to discover a valid assignment for the holes due to the incompleteness in applying Farkas' Lemma. Similar observation has also been documented in the previous works such as by Gulwani et al. [2008]. However, it is not known if the incompleteness in applying Farkas' Lemma prevented the algorithm from discovering minimum solutions, since evaluating the minimality of the inferred constants is quite difficult. Nevertheless, as described in section 5, the minimality of the inferred constants are evaluated empirically.

**Computing the Next Candidate Assignment** The inference algorithm constructs the nonlinear Farkas' constraints (line 15) for falsifying the disjunct $d_{num}$. The nonlinear constraints are conjoined with previously generated constraints if any (lines 15,16). A satisfying assignment to the new constraint will falsify every disjunct explored thus far. In our implementation, such a satisfying assignment is obtained using the Z3 SMT solver with the NLSAT extension [de Moura and Bjørner, 2008].

The satisfying assignment is chosen to be the next candidate substitution $\iota$ for the parameters and the above process is repeated. If the nonlinear constraint $C$ is unsatisfiable at any given step then the algorithm concludes that there exists no solution that would make $\phi$ unsatisfiable. In this case, the VC is refined by unrolling the functions calls as explained in section 3.3.1 and the entire solveUNSAT procedure is reapplied on the refined VC.

**Form Reals to Integer values for Holes** Notice that the procedure solveUNSAT only provides a real number assignment for the holes. The algorithm is also complete only for formulas that use real arithmetic operations. The is due to the use of Farkas Lemma for solving numerical disjuncts with holes ($d_{num}$). However, as noted by previous works as well [Gulwani et al., 2008], the incompleteness in applying Farkas Lemma for solving integer constraints seldom manifests in practice, because of the nature of the integer operations used by the input programs in practice. Moreover, if the numerical disjuncts belong to linear integer arithmetic (i.e, if there is no term in the disjunct that multiplies two variables e.g. a hole with another variable), one could use a linear constraint solver to infer values of the holes and hence retain completeness.

To obtain integer solutions for the holes from the assignment $\iota$ returned by the procedure solveUNSAT, the system uses the ceil of the values assigned to the holes in $\iota$. This strategy is sound since it is known that the holes appear as coefficients of the upper bounds of the resource usage. (If instead they appeared as lower bounds one may have to compute the floor.)

## 3.4 Completeness of Template Solving Algorithm

In this section, I establish the completeness of the template solving algorithm (Figure 3.7). The correctness the algorithm is obvious from the design of the algorithm solveUNSAT. The procedure solveUNSAT returns a model $\iota$ iff $\iota$ makes the formula $\phi$ unsatisfiable. The termination of the algorithm follows from the termination of solveUNSAT, since the unfolding of functions in the VC happens only a bounded number of times (or upto a timeout). The procedure solveUNSAT terminates since, in every iteration of the solveUNSAT algorithm, at least one satisfiable disjunct of elimFunctions($d$) is made unsatisfiable, where $d$ is a disjunct of $\phi$. The number of disjuncts that can be falsified by the solveUNSAT procedure is bounded by $O(2^{n^2})$, where $n$ is the number of atoms in $\phi$. Note that, in practice, our tool explores a very small fraction of the disjuncts (see section 5). However, the completeness of the algorithm is quite non-trivial, which is the subject of this section. First, let us consider the completeness of the solveUNSAT procedure.

### 3.4.1 Completeness of solveUNSAT Procedure

The procedure solveUNSAT is complete for a class of formulas referred to as *parametric linear formulas* with ADTs and uninterpreted functions. Below I describe this logic fragment and prove completeness of the algorithm for this logic fragment. The key property that ensures completeness is that the operation elimFunctions is applied only on a satisfiable disjunct $d$. This guarantees that the predicates in $d$ involving ADT variables do not have any inconsistencies. Since the parameters can only influence the values of numerical variables, axioms that check for inconsistencies among the ADT predicates can be omitted. This property enables us use only the axioms that restricts the values of the numerical variables. This is the key idea behind the proof presented below..

Let $\mathscr{T} = (\Sigma, \mathscr{A})$ be the combined theory of real arithmetic, uninterpreted functions and ADTs with equality. Let $\Sigma = (S, C, F, P)$. The sorts $S$ consists of reals, booleans and ADT sorts. The constants include real numbers and boolean values. The function symbols include arbitrary function identifiers *Fids*, constructor symbols *Cids*, and arithmetic operations $+$, $-$ and $\cdot$. The predicate symbols include $=$, $\leq$ and $<$. The axioms of the theory include the axioms of addition, subtraction and multiplication, arithmetic comparison operations, axioms of the theory of algebraic datatypes, which consists of three axioms [Zhang et al., 2004]: (a) *Injectivity:* $C(x_1, \cdots, x_n) = C(y_1, \cdots, y_n) \iff \bigwedge_{i \leq i \leq n} x_i = y_i$, (b) *Disjointness:* $C_1(x_1, \cdots, x_n) \neq C_2(y_1, \cdots, y_n)$ if $C_1$ and $C_2$ are distinct constructors, and (c) *Acyclicity:* $t(x) \neq x$, if $t$ is built solely by constructors and $t$ properly contains $x$. (We ignore the axioms related to selectors which express selectors using constructors. This is already taken care of by purification.)

A $\Sigma$-formula is *parametric linear* iff every nonlinear term $a \cdot x$, where $a$ and $x$ are variables, $a \in TVars$ and $x \notin TVars$. Let $\mathscr{F}$ denote the set of parametric linear formulas.

**Lemma 12.** *If $d \in \mathscr{F}$ is a disjunct and $\rho \models d$ then $\rho \models chooseEF(d, \rho)$ and $\rho \models elimFunctions(d)$*

*Proof.* The lemma follows by the construction of chooseEF$(d, \rho)$ and elimFunctions$(d)$. □

**Lemma 13** (**Soundness of elimFunctions**)**.** *Let $d \in \mathscr{F}$ be a satisfiable disjunct. Let $\iota$ be a substitution for holes and $dom(\iota) = FV(d) \cap TVars$. If $\not\models$ (elimFunctions$(d)$ $\iota$) then $\not\models d\ \iota$*

*Proof.* We prove the contrapositive form of the lemma: if $d\ \iota$ is satisfiable then elimFunctions$(d)$ $\iota$ is satisfiable. Let $\alpha \models d\ \iota$. I now show that $\alpha \models$ elimFunctions$(d)$ $\iota$.

Consider the assignment $\alpha' : \alpha \uplus \iota$ defined as follows:

$$\alpha' = \begin{cases} \alpha(x) & x \in dom(\alpha) \\ \iota(x) & x \in dom(\iota) \end{cases}$$

Clearly, $\alpha'$ is a model of $d$. By Lemma 12, $\alpha' \models$ elimFunctions$(d)$, which implies that $\alpha \models$ elimFunctions$(d)$ $\iota$ □

The converse of the above theorem does not hold i.e, whenever $d$ is unsatisfiable it does not imply elimFunctions$(d)$ is unsatisfiable, since it is a weaker formula. However, interestingly if $d$ is a satisfiable disjunct then the following property holds, which suffices for the completeness of the solveUNSAT procedure.

**Lemma 14** (***(Completeness of elimFunctions)***)**.** *Let $d \in \mathscr{F}$ and $\rho \models d$. Let $d' = chooseEF(d, \rho)$. Let $\iota \in Subst$ be a substitution for holes i.e, $dom(\iota) = FV(d) \cap TVars$. If $\not\models d\ \iota$ then $\not\models d'\ \iota$.*

*Proof.* I prove the contrapositive form of the lemma: if $d'\ \iota$ is satisfiable then $d\ \iota$ is satisfiable. Let $\alpha' \models d'\ \iota$. Let $d\ \iota = d_{num} \wedge d_{uf} \wedge d_{adt}$ where $d_{num}$ is a conjunction of atoms containing only numerical variables $Vars_n$, $d_{uf}$ is a conjunction of atoms of the form $r = f(x_1, \cdots, x_n)$ for some function $f \in Fids$, and $d_{adt}$ is a conjunction of atoms of the form $r = cons(x_1, \cdots, x_n)$ or $u \neq v$ or $u = v$, for some constructor $cons \in Cids$ and ADT variables $\{u, v\} \subseteq Vars_t$.

A model $\alpha$ of $d\ \iota$ is constructed as follows. Initially, I start with an initial assignment $\alpha_{init}$ that binds the numerical variables $Vars_n$ in $d\ \iota$ to concrete values. Each atom $a$ of $d\ \iota$ is incrementally considered and the initial assignment is extended by introducing bindings for previously unseen identifiers in $a$ in such a way that $a$ is satisfied. The assignment obtained after processing every atom of $d\ \iota$ models $d\ \iota$.

Let the initial assignment $\alpha_{init}$ be the projection of $\alpha'$ on to variables in $Vars_n$. That is, $\alpha_{init}(x) = \alpha'(x)$ if $x$ is a numerical variable.

**Part-I: Proof of $\alpha_{init} \models d_{num}$**

$d_{num}$ is a part of $d'\ \iota$ i.e, $d_{num} \subseteq d'\ \iota$. This follows from the definition of elimFunctions$(d)$. Therefore, $\alpha' \models d_{num}$. By definition, $d_{num}$ has only numerical variables in $Vars_n$. Therefore, $\alpha_{init} \models d_{num}$.

**Part-II: Extending $\alpha_{init}$ to variables in $d_{adt}$**

I introduce a few definitions and operations that will be useful in the rest of the proof. Say that a substitution $\alpha$ *embeds* ($\preccurlyeq$) in a substitution $\rho$ iff the following conditions hold:
1. $dom(\alpha) \subseteq dom(\rho)$
2. $\alpha$ preserves all equalities and disequalities between ADT variables $Vars_t$ implied by $\rho$.

$$\forall \{x, y\} \subseteq dom(\alpha) \cap Vars_t. \quad \rho(x) = \rho(y) \iff \alpha(x) = \alpha(y)$$

3. For every variable $x \in dom(\alpha) \cap Vars_t$, $\rho(x)$ and $\alpha(x)$ are *shape isomorphic*. That is, $\rho(x)$ and $\alpha(x)$ may differ only in the numerical values.

Say that an assignment $\alpha$ is a $S$-model for some set $S$ of atoms iff $\alpha$ is a satisfying assignment of every atom in $S$. Given a $S$-model $\alpha$, an assignment $\rho$ such that $\alpha \preccurlyeq \rho$ and a variable $x \in dom(\rho) \setminus dom(\alpha)$, we define $\alpha \rhd_\rho x$ as an assignment that extends $\alpha$ by introducing a mapping for $x$ such that $(\alpha \rhd_\rho x) \preccurlyeq \rho$ and $\alpha \rhd_\rho x$ is a $S$-model.

$$(\alpha \rhd_\rho x)(u) = \begin{cases} \alpha(v) & \exists v \in dom(\alpha).\ \rho(v) = \rho(u) \\ freshen_\alpha(\rho(u)) & \text{otherwise} \end{cases}$$

where $freshen_\alpha(v)$ return a fresh numeral that does not occur in $\alpha$ if $v$ is a numeral. If $v$ is an ground ADT term, it replaces every numeral in $v$ by a fresh numeral that does not occur in $\alpha$.

It is easy to see that $(\alpha \rhd_\rho x) \preccurlyeq \rho$ as all the three properties of embedding are preserved by the above definition. It is also obvious that $\alpha \rhd_\rho x$ is a $S$-model as $x$ does not appear in $S$ since $x \notin dom(\alpha)$ and $\alpha$ is a $S$-model. With these definitions, we now proceed to the proof.

Let $d_{adt} = d_{cons} \wedge d_{comp}$, where $d_{cons}$ is a conjunction of atoms of the form $r = cons(x_1, \cdots, x_n)$ and $d_{comp}$ is a conjunction of atoms of the form $u \neq v$ or $u = v$. Let the atoms in $d_{cons}$ be ordered as $a_1 : r_1 = cons_1(X_1), \cdots, a_n : r_n = cons_n(X_n)$ such that for each $r_i$, the *size* of the ground ADT term $\rho(r_i)$ is larger than or equal to the size of every $\rho(r_j)$, $j \in [1..i]$. The size of a ground ADT term is the number of constructors used by the term. This ordering ensures that $\rho(r_i)$ cannot have as sub-structures $\{\rho(r_j) \mid j > i\}$. Given a set of ground ADT terms such an ordering is always possible as the ADTs are *acyclic*.

I construct, as explained below, a sequence of assignments $\alpha_0, \cdots, \alpha_n$, where $\alpha_0 = \alpha_{init}$ such that each $\alpha_i$ satisfies the following properties: (a) $\alpha_i \models \bigwedge_{j=1}^{i} a_i$ and (b) $\alpha_i \preccurlyeq \rho$. Note that $\alpha_0$, which equals $\alpha_{init}$, satisfies the above properties. Given an assignment $\alpha_{i-1}$ that satisfies the above properties, we now show how to construct an assignment $\alpha_i$ satisfying the above properties.

Let the atom $a_i$ be $r = cons(x_1, \cdots, x_m)$. Let $S = \{a_j \mid j \in [1..i]\}$. By definition of $S$, $\alpha_{i-1}$ is a $S$-model. Let $W \subseteq \{x_1, \cdots, x_m\}$ be the set of variables that do not belong to $dom(\alpha_{i-1})$. Let

$W = \{w_1, \cdots, w_k\}$. Define $\alpha_{ext}$ as

$$\alpha_{ext} = (\cdots((\alpha_{i-1} \rhd_\rho w_1) \rhd_\rho w_2) \cdots) \rhd_\rho w_k$$

By the definition of $\rhd$, $\alpha_{ext}$ is a $S$-model and $\alpha_{ext} \preccurlyeq \rho$. $\alpha_{ext}$ has a mapping for all $x_1, \cdots, x_m$. Let $t$ denote the ground ADT term $cons(\widehat{\alpha_{ext}}(x_1, \cdots, x_m))$. Define $\alpha_i$ as $\alpha_{ext}[r \mapsto t]$.

**Property 1.** *If $r' \in dom(\alpha_{ext})$ and $\rho(r') = \rho(r)$ then, $\alpha_{ext}(r') = t$*

*Proof.* Since $r' \in dom(\alpha_{ext})$ it should either be an argument passed to the constructor *cons* (say $x_i$) or should belong to $dom(\alpha_{i-1})$. If $r' = x_i$ then, $\rho(r)$ must contain itself, as $\rho \models a_i : r = cons(\cdots, x_i, \cdots)$ and $\rho(r') = \rho(r)$. This is not possible as $\rho(r)$ is acyclic. Therefore, $r'$ cannot be an argument $x_i$. Therefore, say $r' \in dom(\alpha_{i-1})$. There are two scenarios in which $r'$ could have been assigned a mapping in $\alpha_{i-1}$: (a) there exists $a_j : z = cons'(y_1, \cdots, y_m)$, $j < i$ and $r' = y_i$ for some $i \in [1..m]$ or (b) there exists $a_j : r' = cons'(y_1, \cdots, y_m)$, $j < i$.

Consider case (a). Say $r'$ is an argument $y_i$ of $cons'$. Clearly, $y_i$ is a sub-structure of $z$. Since, $\rho$ is a model of $a_j$, $\rho(y_i)$ is a sub-structure of $\rho(z)$. By our choice of ordering of the atoms, $\rho(r)$ is at least as large as $\rho(z)$ as $a_j$ precedes $a_i$ in the ordering. Therefore, $\rho(y_i)$ is strictly smaller than $\rho(r)$. Since $\alpha_{ext} \preccurlyeq \rho$, by property 3.4.1, $\rho(r)$ and $\alpha_{ext}(r)$ have the same size, and $\rho(y_i)$ and $\alpha_{ext}(y_i)$ have the same size. Therefore, $\alpha_{ext}(y_i)$ is strictly smaller than $\alpha_{ext}(r)$. Since, $r' = y_i$, $\alpha_{ext}(r')$ is strictly smaller than $\alpha_{ext}(r)$. Which contradicts our premise that $\alpha_{ext}(r') = \alpha_{ext}(r)$. Therefore, this case is not possible.

Consider case (b). If $cons'$ and *cons* are different constructors then $\rho(r) = \rho(r')$ is not possible. Therefore, assume that $cons$ and $cons'$ are equal. In the rest of the proof, I use *cons* instead of $cons'$.

Recall that $d' = \mathsf{chooseEF}(d, \rho)$. Since $\rho(r) = \rho(r')$ the atom $r = r'$ would belong to $d'$. Similarly, $\bigwedge_{i=1}^{m}(x_i = y_i) \subseteq d'$ as for all $i$, $\rho(x_i) = \rho(y_i)$. If, for some $i$, $\{x_i, y_i\} \subseteq Vars_n$ then $\alpha_{ext} \models (x_i = y_i)$. This is because $(x_i = y_i) \in d'$, $\alpha' \models d'$, $\alpha_{init}$ is a projection of $\alpha'$ on to $Vars_n$ and $\alpha_{init} \subseteq \alpha_{ext}$. Therefore, say $\{x_i, y_i\} \subseteq Vars_t$. Since $\rho(x_i) = \rho(y_i)$ and $\alpha_{ext} \preccurlyeq \rho$, $\alpha_{ext}(x_i) = \alpha_{ext}(y_i)$. Hence, $\alpha_{ext}(x_i) = \alpha_{ext}(y_i)$, for all $i \in [1..m]$. Hence, $\alpha_{ext}(r') = t$. □

**Property 2.** *If $r' \in dom(\alpha_{ext})$ and $\rho(r') \neq \rho(r)$ then, $\alpha_{ext}(r') \neq t$.*

*Proof.* $r' \in dom(\alpha_{ext})$ implies that $r' \in W$ or $r' \in dom(\alpha_{i-1})$, where $W$ is the set of constructor arguments that do not belong to $dom(\alpha_{i-1})$. Clearly, if $r' \in W$ then, $\alpha_{ext}(r') \neq t$ by the definition of $t$. Hence, the claim holds. Say $r' \in dom(\alpha_{i-1})$. There are two scenarios in which $r'$ could have been assigned a mapping in $\alpha_{i-1}$: (a) there exists $a_j : z = cons'(y_1, \cdots, y_m)$, $j < i$ and $r' = y_i$ for some $i \in [1..m]$ or (b) there exists $a_j : r' = cons'(y_1, \cdots, y_m)$, $j < i$.

Consider case (a). As explained in the proof of the above claim, $\alpha_{ext}(r')$ is strictly smaller than $\alpha_{ext}(r)$ in this case. Therefore, $\alpha_{ext}(r') \neq t$ holds. Hence, the claim holds.

Consider case (b). If *cons* and *cons*′ are different constructors then $\alpha_{ext}(r') \neq t$ and hence the claim holds. Hence, consider the case where *cons* and *cons*′ are the same constructor.

Given $\rho(r) \neq \rho(r')$. Therefore, the values of atleast one of the constructor arguments should be different in $\rho$. Recall that $d' = \mathsf{chooseEF}(d, \rho)$. By definition of chooseEF, an atom of the form $x_i \neq y_i$, for some $i$, wherein $\rho(x_i) \neq \rho(y_i)$ will belong to $d'$. If $\{x_i, y_i\} \subseteq Vars_n$ (numerical variables) then $\alpha_{ext} \models x_i \neq y_i$. This is because $(x_i \neq y_i) \in d'$, $\alpha' \models d'$, $\alpha_{init}$ is a projection of $\alpha'$ on to $Vars_n$ and $\alpha_{init} \subseteq \alpha_{ext}$. Consider the case where $\{x_i, y_i\} \subseteq Vars_t$. $x_i \neq y_i$ belongs to $d'$ only because $\rho(x_i) \neq \rho(y_i)$. Since $\alpha_{ext} \preccurlyeq \rho$, $\alpha_{ext}(x_i) \neq \alpha_{ext}(y_i)$ (property 3.4.1 of $\preccurlyeq$ definition). Hence, $\alpha_{ext}(r') \neq t$. □

**Property 3.** *$t$ is shape isomorphic to $\rho(r)$. That is, $t$ and $\rho(r)$ differ only in numerical values.*

*Proof.* By definition, $t = cons(\widehat{\alpha_{ext}}(x_1, \cdots, x_m))$. For each $x_i \in Vars_t$, $\alpha_{ext}(x_i)$ is either $\alpha_{i-1}(x_i)$ or is obtained by applying an extension operation $\alpha \triangleright_\rho x_i$ for some $\alpha \preccurlyeq \rho$. In the former case, $\alpha_{ext}(x_i)$ and $\rho(x_i)$ are shape isomorphic by hypothesis. In the latter case, they are shape isomorphic by the definition of $\triangleright$. Since, $\rho$ satisfies $a_i : r = cons(x_1, \cdots, x_m)$, $\rho(r) = cons(\hat{\rho}(x_1, \cdots, x_n))$. The fact that, for all $i \in [1..m]$, $\rho(x_i)$ is shape isomorphic to $\alpha_{ext}(x_i)$ implies the claim. □

By the above three properties, $\alpha_i \preccurlyeq \rho$. By definition of $t$, $(r \mapsto t) \models a_i$. Therefore, $\alpha_i$ models $(S \cup \{a_i\})$. By induction the claims hold for all $\alpha_i, 0 \leq i \leq n$. Therefore, $\alpha_n \models d_{cons}$ and $\alpha_n \preccurlyeq \rho$. Let $x_1, \cdots, x_m$ be the set of variables in $dom(\rho) \cap Vars_t$ that do not have a mapping in $\alpha_n$. Define an assignment $\alpha_{adt} = (\cdots((\alpha_n \triangleright_\rho v_1) \triangleright_\rho v_2) \cdots) \triangleright_\rho v_m)$. By definition of $\triangleright$, $\alpha_{adt} \models d_{cons}$ and $\alpha_{adt} \preccurlyeq \rho$. Furthermore, $\alpha_{adt}$ has a mapping for all ADT variables in $d$.

Now it is almost trivial to prove that $d_{comp}$ is satisfied by $\alpha_{adt}$. Since $\alpha_{adt}$ implies all equalities and disequalities implied by $\rho$ (property 3.4.1 of $\preccurlyeq$ relation), and since $\rho$ satisfies $d_{comp}$, $\alpha_{adt}$ also satisfies $d_{comp}$.

**Part-III: Extending $\alpha_{adt}$ to uninterpreted functions in $d_{uf}$**

Here I show how the assignment $\alpha_{adt}$ can be extended to uninterpreted functions in the disjunct $d$. Let $d_{uf} = \bigwedge_{i=1}^{n} a_i$. Similar to Part-II, a sequence of assignments $\alpha_0, \alpha_1, \cdots, \alpha_n$ is created, where $\alpha_0 = \alpha_{adt}$, such that each $\alpha_i$ is a satisfying assignment of all atoms $a_1$ to $a_i$. For convenience, the assignments are allowed to track partial maps for function symbols.

$\alpha_0$ trivially satisfies the claim. Assume that $\alpha_{i-1}$ satisfies all atoms up to $a_{i-1}$. Construct a $\alpha_i$ as described below that satisfies all atoms up to $a_i$. Let $a_i : r = f(u_1, \cdots, u_n)$. Let $E = \widehat{\alpha_{i-1}}(u_1, \cdots, u_n)$.

Define $\alpha_i$ as $\alpha_{i-1}$ if $f$ is defined for $E$ at $\alpha_{i-1}$. Otherwise, let $\alpha_i$ be $(\alpha_{i-1} \cup \{f \mapsto (E \mapsto \alpha_{i-1}(r))\})$, where the $\cup$ operation combines the partial definitions of $f$ given by the left and the right operands. Note that $\alpha_{i-1}$ should be defined at $r$ since $r$ is either of numerical sort or ADT sort

and hence should be defined in $\alpha_{adt}$. Now to show that $\alpha_i \models a_i$, we only need to show the following property, which completes the proof.

**Property 4.** *If $f$ is defined for $E$ at $\alpha_{i-1}$ then $(\alpha_{i-1}(f))(E) = \alpha_{i-1}(r)$*

*Proof.* Since $f$ is defined for $E$ in $\alpha$, it should have been added to some $\alpha_j$, $j < i$. Which implies that $a_j : r' = f(v_1, \cdots, v_m)$, $\widehat{\alpha_{j-1}}(v_1, \cdots, v_m) = E$ and $\alpha_j(f)(E) = \alpha_{j-1}(r')$. I now show that $\alpha_{j-1}(r')$ (which equals $\alpha_{i-1}(r')$) and $\alpha_{i-1}(r)$ are equal. There are two cases to consider: (a) $\rho(r) = \rho(r')$ and (b) $\rho(r) \neq \rho(r')$.

Consider case (a). If $\{r, r'\} \subseteq Vars_t$ then $\alpha_{i-1}(r) = \alpha_{i-1}(r')$ as $\alpha_{i-1} \preccurlyeq \rho$. Hence, the claim holds. Say $\{r, r'\} \subseteq Vars_n$. Since $\rho(r) = \rho(r')$, by the choice of the disjunct $d'$, $r = r'$ belongs to $d'$. This implies that $\alpha_{init}(r)$ (which is $\subseteq \alpha_{i-1}(r)$) models $r = r'$ as $\alpha' \models (r = r')$ and $\alpha_{init}$ is projection of $\alpha'$ on to $Vars_n$. Therefore, the claim holds.

Consider case (b): $\rho(r) \neq \rho(r')$. In this case, by the choice of $d'$, a predicate $u_i \neq v_i$, for some $i \in [1..m]$, wherein $\rho(u_i) \neq \rho(v_i)$ will belong to $d'$. By an argument similar to the previous cases, it can be shown that in this case, $\alpha_{i-1}(u_i) \neq \alpha_{i-1}(v_i)$ irrespective of whether $u_i, v_i$ belong to $Vars_n$ or $Vars_t$. Therefore, $\widehat{\alpha_{i-1}}(v_1, \cdots, v_n) \neq E$ and hence, $\widehat{\alpha_{j-1}}(v_1, \cdots, v_n) \neq E$ (since $\alpha_{j-1} \subseteq \alpha_{i-1}$). This implies that the entry for $E$ is not added by the atom $a_j$ contrary to our assumption. Hence, this case is not possible. □

**Theorem 15** (***Completeness of solveUNSAT***)**.** *Let $\phi \in \mathscr{F}$ be a quantifier-free parametric linear formula with holes: holes. Let $\iota$ = solveUNSAT($\phi$, holes).*
*1. The procedure solveUNSAT is correct. That is, if $\iota \neq \emptyset$ then $\phi\,\iota$ is unsatisfiable.*
*2. The procedure solveUNSAT is complete. That is, if $\iota = \emptyset$ then there does not exist an assignment for params that will falsify $\phi$.*
*3. The procedure solveUNSAT terminates.*

*Proof.* The solveUNSAT algorithm shown in Figure 3.8 returns a non-empty model $\iota$ only at line 7. This line will be reached only when $\phi_{inst} = \phi\,\iota$ is unsatisfiable. Hence, solveUNSAT is correct.

The solveUNSAT algorithm returns $\emptyset$ only at line 17. When line 17 is reached the following properties hold. (a) $\alpha' \models d$, where $d \in disjunct(\phi)$, (b) $d'$ = chooseEF($d, \alpha'$), and (c) there exists no assignment $\iota$ for holes such that $d'\,\iota$ is unsatisfiable. The property (c) follows from the completeness of Farkas' Lemma for linear real arithmetic. By Lemma 14, and properties (a),(b) and (c) imply that there does not exist an assignment for holes that will falsify $d$ and hence, $\phi$. Therefore, the procedure is complete.

Let $d$ be a disjunct of $\phi$. Let $\{d_1, \cdots, d_m\}$ be the disjuncts of elimFunctions($d$). $d$ can be chosen at line 9 of the algorithm at most $m$ times. This is because, every time $d$ is chosen, at least one $d_i, i \in [1..m]$ that was satisfiable in the earlier iterations will be falsified by the new assignment computed at line 20. Once all disjuncts $d_i, i \in [1..m]$ are falsified by an assignment $\iota$ for holes,

elimFunctions($d$) $\iota$ will be unsatisfiable. By Lemma 13, $\not\models$ elimFunctions($d$) $\iota$ implies $\not\models d\ \iota$. Hence, $d$ will be falsified when all the disjuncts of elimFunctions($d$) are falsified. Therefore, $d$ can be chosen at line 9 of the algorithm at most $m$ times. Since there are only a finite number of disjuncts in $\phi$, the number of iterations of the algorithm is bounded.

In each iteration, the algorithm solves for the non-linear constraints $C$. Since $C$ is a quantifier-free real arithmetic formula it has a decision procedure. Therefore, the algorithm terminates. □

I now extend the Theorem 15 to also include the theory of sets. Though the core language described until now does not have built-in support for sets, our implementation provides built-in support for sets, since they have decidable logic fragments and are supported by off-the-shelf SMT solvers. Sets are also quite essential for the transformation for handling memoization described later in section 4.

**Theorem 16.** *Given a quantifier-free parametric linear formula $\phi$ with free variables $\bar{x}$ and $\bar{a} \in TVars^*$, belonging to a theory $\mathcal{T}$ that is a combination of theories of uninterpreted functions, algebraic datatypes, sets and real arithmetic, finding an assignment $\iota$ such that $dom(\iota) = |\bar{a}|$ and $(\phi\ \iota)$ is $\mathcal{T}$-unsatisfiable is decidable.*

*Proof.* Express the problem as trying to decide the validity of a formula of the form: $\exists \bar{a} \in TVars^*.\forall \bar{x'}.(\forall \bar{f}.\ \phi') \wedge (\forall \bar{s}.\ \phi_{set})$, where, $\bar{f}$ are the uninterpreted function symbols in $\phi$, $\bar{s}$ are the variables of set sort, $\bar{x'}$ are variables of other sorts, $FV(\phi') \subseteq \bar{a} \cup \bar{x'}$, $FV(\phi_{set}) \subseteq \bar{s} \cup \bar{x'}$, and $\phi_{set}$ is a formula in $\mathcal{T}_{set}$ that has only set operations: set construction, union, intersection and complement. This is possible because the existentially quantified variables $\bar{a}$ are only numerical variables and appear only as coefficients of numerical valued expression. They do not appear in functions or relations that involve sets. Since the theory of sets admit decidable quantifier elimination [Kuncak et al., 2006], the above formula could be reduced to an equivalent formula of the form $\exists \bar{a}.\forall \bar{x'}, \bar{f}.\phi''$, where $FV(\phi'') = \bar{a} \cup \bar{x'}$, that do not have any variables of set sort. This formula can be decided using the solveUNSAT algorithm by Theorem 15. □

I now present the main theorem of this Chapter that establishes the completeness of the inference algorithm for a program fragment with recursive functions and recursive datatype, albeit when the arithmetic operations are interpreted as real arithmetic operations. The theorem is based on the fact that for a certain class of functions manipulating ADTs called *sufficiently surjective* functions [Suter et al., 2010, 2011], it suffices to unfold them a finite number of times to decide any formula involving these functions.

**Theorem 17** (***Completeness of Inference Algorithm***)**.** *Let $P$ be a program with contracts that may have holes in the post-conditions, recursive functions, real-valued variables and algebraic datatypes, where every arithmetic operation is parametric linear, and every recursive function in the program is sufficiently surjective. The inference algorithm depicted by Figure 3.7 is a*

*decision procedure for inferring values for the holes that when substituted in the postconditions makes all contracts in the functions in P valid.*

*Proof.* Follows from the completeness of solveUNSAT procedure for parametric linear formulas with uninterpreted functions and ADTs (Theorem 15), and the completeness of unfolding of functions for modeling the behavior of *sufficiently surjective* functions [Suter et al., 2010, 2011] □

## 3.5 Solving Nonlinear Formulas with Holes

Nonlinearity is common in resource bounds. In this section, I discuss the approach for handling nonlinear formulas with holes like $\phi_{ex} : wz < xy \wedge x < w - 1 \wedge y < z - 1 \wedge ax + b \leq 0 \wedge ay + b \leq 0$ where $a, b$ are holes. The goal here is to find an assignment for $a$ and $b$ that will make the formula $\phi_{ex}$ unsatisfiable. Notice that the formula has terms like $wz$ that multiply two variables that are not holes. The algorithm solveUNSAT is aimed at solving only parametric linear formula where all nonlinear terms are of the form $ay$, where $a \in TVars$ and $y \notin TVars$. I now present the approach used by the system to eliminate such nonlinearity from the VC by axiomatizing the nonlinear operations. This component is represented by the block *Nonlinear axiom instantiation* in Figure 3.7.

Our approach encodes multiplication as an uninterpreted operation that is axiomatized by axioms such as: $\forall x, y.\ xy = (x-1)y + y$, $\forall x, y.\ xy = x(y-1) + x$, monotonicity properties like $(x \geq 0 \wedge y \geq 0 \wedge w \geq x \wedge z \geq y) \Rightarrow xy \leq wz$, associativity and distributivity over +. The exponential and logarithmic function are defined using recursive functions in the standard library, along with some of their oft-required properties. For instance, the exponential function $2^x$ is defined as **def** tpow $x := \{(x \geq 0)\}$ **if** $(x = 0)$ *1* **else** $2 \cdot$ tpow$(x - 1)$ $\{true\}$, which corresponds to the axiom: $2^x = 2 \cdot 2^{x-1}$, for $x \geq 1$, and 0 otherwise. A ⌊log⌋ function often used by our benchmarks in defined by the function log shown below in Scala syntax:

```
def log(x: BigInt): BigInt = {
  require(x >= 1)
  if (x ≤ 1) 0
  else 1 + log(x/2)
} ensuring(res ⇒ res ≥ 0)
```

```
def logMono(x, y): Boolean = {
  require(x >= y && y >= 1)
  (if(y <= 1) true else logMono(x/2, y/2)) &&
    (log(x) >= log(y))
} holds
```

The properties such as monotonicity of log and $2^x$ are also expressed and verified in the library. For instance, the function logMono shown above establishes the monotonicity property of log. (The recursive call logMono(x/2, y/2) encodes an induction strategy for proving this property by exploiting the assume-guarantee reasoning described in section 3.2.) These axioms can be incorporated into the verification conditions by recursive instantiation as explained below.

Axioms such as $xy = (x-1)y + y$ that are recursively defined are instantiated similar to unfolding a recursive function during VC refinements. For example, in each VC refinement, for every atomic predicate $r = xy$ that occurs in the VC, a new predicate $r = (x-1)y+y$ is added if it does not exist (syntactically) . A binary axiom such as monotonicity is instantiated on every pair of terms in the VC on which it is applicable. For instance, if $r = f(x)$, $r' = f(x')$ are two atoms in the VC and if $f$ has a monotonicity axiom, the predicate $(x \leq x' \Rightarrow r \leq r') \wedge (x' \leq x \Rightarrow r' \leq r)$ is conjoined to the VC. This approach can be extended to N-ary axioms. Every other user-defined axiom should be manually instantiated by the user by asserting it on appropriate arguments in the pre-or post-condition of the function whose verification requires the axiom.

Consider the example formula $\phi_{ex}$ shown above. Instantiating the multiplication axioms a few times will produce the following formula (simplified for brevity):

$$wz < xy \wedge xy = (x-1)(y-1) + x + y - 1 \wedge ((x \geq 0 \wedge y \geq 0 \wedge x \leq w \wedge y \leq z) \implies xy \leq wz) \\ \wedge x < w - 1 \wedge y < z - 1 \wedge ax + b \leq 0 \wedge ay + b \leq 0$$

This formula can be solved without interpreting multiplication i.e, treating it as uninterpreted. For instance, $a = -1, b = 0$ is a solution for the holes that will make the formula unsatisfiable.

## 3.6 Finding Strongest Bounds

In this section, I describe the approach used by the system for computing strongest bounds that satisfy a given template. We know that every hole in the template appears as a coefficient of some expression. As a first step, the system approximates the rate of growth of an expression in the template by counting the number of function invocations (including nonlinear operations) performed by the expression. The holes are ordered in the descending order of the estimated rate of growth of the associated expression, breaking ties arbitrarily. Let this ordering be denoted by $\sqsubseteq$.

For instance, given a template $\mathsf{res} \leq a \cdot f(g(x, f(y)) + c \cdot g(x) + a \cdot x + b$, the holes in the template are ordered as $a \sqsubseteq c \sqsubseteq b$. Define an order $<^*$ on the substitutions $TVars \mapsto \mathbb{R}$ by extending $<$ lexicographically with respect to the ordering $\sqsubseteq$ as follows:

$$\forall \{\iota, \gamma\} \subseteq (TVars \mapsto \mathbb{R}).\ \iota <^* \gamma \ \ \textit{iff}\ \ \exists a \in TVars.\, \iota(a) < \gamma(a) \wedge \forall b \in TVars.\, b \sqsubseteq a \implies \iota(b) = \gamma(b)$$

A *locally* minimum solution $\iota_{min}$ for the holes is found with respect to $\leq^*$ using a binary search as explained below.

Let $\iota$ be the solution found by the solveUNSAT procedure. We know that $\iota$ is obtained by solving a set of nonlinear constraints $C$ (see line 15 of Figure 3.8). A minimum satisfying assignment $\iota_{min}$ for $C$ with respect to the total order $\leq^*$ is computed by performing a *binary search* on the solution space of $C$ starting with the initial upper bound given by $\iota$. The binary search stops when, for each hole $a$, the difference between the values of $a$ in the upper and lower bounds is found to be $\leq 1$. Note that the difference between the upper and lower bounds

needs to bounded from below (and cannot be required to be zero) since the values of holes are real numbers. The minimal assignment $\iota_{min}$ may not falsify $\phi$ although $\iota$ does. This is because $C$ only encodes the constraints for falsifying the disjuncts of $\phi$ explored until some iteration. The algorithm uses $\iota_{min}$ as the next candidate model and continues the iterations of the solveUNSAT algorithm.

In general, the inferred bounds are not guaranteed to be the strongest as the verification conditions generated are sufficient but not necessary conditions. However, it would be the strongest solution if the functions in the program are *sufficiently surjective* [Suter et al., 2010, 2011], if the recursive functions are unfolded to sufficient depth, there are no nonlinear operations and there is no loss of completeness due to applying Farkas' Lemma on integer formulas. Our system also supports finding a concrete counterexample, if one exists, for the values smaller than those that are inferred, which is more pragmatic approach to testing minimality of the inferred constants.

## 3.7 Analysis Strategies and Optimizations

**Inference of Auxiliary Templates** The system supports generation of *templates* for inferring program invariants automatically for some functions. This is to alleviate the users from having to specify simple properties such as that a result of a function is positive, which may be necessary for establishing resource bounds. For every function $f$ for which a template has not been provided by the user, a default template is constructed which is a linear combination of integer valued arguments and return values of $f$. For instance, for the function size(l) the default template is a∗res+b≤0 (where, res is the return value of size). This allows the algorithm to infer that res≥0 is a valid postcondition of the function size automatically. Note that this property is necessary to verify the running example: revRec function shown in Figure 3.2

**Inter-procedural Analysis** When the input program has more than one function, our system solves the resource bound templates for the functions modularly in a bottom-up fashion. In our system, the resource bound templates of the callees are solved independent of the callers. The bounds of the callees are then used while analysing the callers. This strategy enhances the scalability of the system by allowing the inference algorithm to focus on one function at a time, and also allows establishing resource bounds of open programs like data structure libraries.

However, as an exception, the auxiliary templates that are inferred automatically are solved in the context of the callers in order to find non-trivial, context-specific invariants. This is because, if an automatically generated template such as a∗res+b≤0 is solved independent of a context then it difficult to prevent inference of trivial solutions such as $a = 0$, $b = 0$, which is equivalent to saying that the postcondition is *true*.

**Targeted Unfolding** Recall that the inference algorithm unfolds functions in a VC if the VC is not solvable by solveUNSAT (i.e, when the condition at line 17 is true). Our system uses a demand-driven unfolding process in which only those functions encountered in the disjuncts explored by the solveUNSAT procedure are unfolded. This avoids unfolding functions along disjuncts that are already unsatisfiable in the VC. These include (but not restricted to) paths in the program that are unsatisfiable within the depth of unfolding of functions in the VC.

**Prioritizing Disjunct Exploration** Typically, the VCs that are generated have a large number of disjuncts some of which are easier to reduce to false compared to others. The algorithm is directed towards choosing the easier disjuncts by using timeouts on the nonlinear constraints solving process. Whenever a timeout happens while solving a nonlinear constraint, the disjunct that produced the nonlinear constraint in the VC is blocked so that it is not chosen again. This can be accomplished by introducing a new predicate that denotes that the conjunction of the control literals corresponding to the disjunct is false. Our system by default uses a timeout of 20 seconds. However, a different timeout can be provided as a command line input to the system. All our experiments were carried out using the default timeout. This strategy, though conceptually simple, made the analysis converge faster on many benchmarks.

## 3.8 Divide-and-Conquer Reasoning for Steps Bounds

In this section, I describe an alternative reasoning supported by our system for establishing steps bounds involving nonlinear multiplication (i.e, multiplication of two program variables or terms). Often, the bounds on the number of evaluation steps involve nonlinear multiplication. In such cases, by default, our system relies on the inference algorithm to handle multiplication using built-in axioms as described in section 3.5. However, the presence of nonlinear multiplication makes the inference algorithm more incomplete and also much slower since the instantiation of axioms of multiplication increases the sizes of the verification conditions. In the sequel, I present a compositional, light-weight reasoning for proving nonlinear bounds when the reason for nonlinearity is the presence of nested computations. For example, consider the implementation of an insertion sort algorithm shown below.

```
def sortedIns(e: BigInt, l: List): List = {
  l match {
    case Cons(x,xs) ⇒ if (x ≤ e) Cons(x,sortedIns(e, xs)) else Cons(e, l)
    case _ ⇒ Cons(e,Nil())
  }
} ensuring(res ⇒ size(res) == size(l) + 1 && steps ≤ ? * size(l) + ?)

def sort(l: List): List = (l match {
  case Cons(x,xs) ⇒ sortedIns(x, sort(xs))
  case _ ⇒ Nil()
}) ensuring(res ⇒ size(res) == size(l) && steps ≤ ? * (size(l)*size(l)) + ?)
```

Intuitively, the number of steps taken by the sort function is quadratic in the size(l) , since it performs $O$(size(l)) recursions each taking $O$(size(l)) number of steps (as they invoke the sortedIns function). Our system provides built-in support for encoding this "divide-and-conquer" reasoning. For instance, one can guide our system to infer the steps bound for the sort function using this divide-and-conquer reasoning as illustrated below:

```
@compose
def sort(l: List): List = (l match {
  case Cons(x,xs) ⇒ sortedIns(x, sort(xs))
  case _ ⇒ Nil()
}) ensuring(res ⇒ size(res) == size(l) &&
        rec ≤ ? ∗ size(l) + ? &&
        tpr ≤ ? ∗ size(l) + ? &&
        steps ≤ ? ∗ (size(l)∗size(l)) + ?)
```

The keyword rec denotes the number of recursive calls invoked by a function. This resource also counts calls to mutually recursive functions determined based on a static call graph. The keyword tpr, which is stands for *steps per recursive call*, denotes the number of steps in the evaluation of the body of the function excluding the steps taken by the recursive (or mutually recursive) calls. The cost functions for these resources are formally defined below.

**Cost function definition for rec:**

$\oplus = +$

$$c_{call} = c_{app} = \begin{cases} 1 & \text{if the callee is mutually recursive with the caller} \\ 0 & \text{Otherwise} \end{cases}$$

$c_{op} = 0$ for every other operation $op$

**Cost function definition for tpr:**

$\oplus = +$

$$c_{call} = c_{app} = \begin{cases} 1 & \text{if the callee is } \textit{not} \text{ mutually recursive with the caller} \\ 0 & \text{Otherwise} \end{cases}$$

$c_{op}$ is equivalent to $c_{op}$ for steps for every other operation $op$

Note that the definition of cost functions for rec and tpr requires knowing the function that is under execution as well as the callee function that is invoked by a call expression. One can assume that this information is additionally tracked by the operational semantics and implicitly passed to the cost functions. Indeed, in our system, the phase that performs instrumentation of these resources has access to a static call graph and hence can retrieve these details.

**Maximizing TPR over Recursive Calls** A closer inspection of the definition of the cost function for tpr would reveal that tpr actually measures the number of steps in the evaluation of

the *first* call to the function (excluding steps taken by the recursive calls). However, what is necessary for the divide-and-conquer reasoning is an upper bound on the value of tpr for any call to the function (not necessarily the first). To infer such an upper bound, we additionally impose a constraint that the bound provided in the contracts (possibly with holes) on the tpr resource monotonically decreases across the recursive calls made by the function. This is achieved by conjoining the following obligation with the modular assume-guarantee obligations generated for every function $g$ that has a bound on the tpr resource.

(2.8.I) For each *recursive* call site $(f\ y)^{\ell}$ in $g$, $\models_P path((f\ y)^{\ell}) \rightarrow tprl_g \geq tprl_f[y/param_P(f)]$

In the above condition, $tprl_\alpha$ ($\alpha \in \{f, g\}$) denotes the upper bound (possibly with holes) provided in the postcondition of a function $\alpha$ for the tpr resource. The above obligation ensures that the inferred tpr bound is maximum across all recursive calls. To understand why, consider a recursive call $\langle Env', (f\ y)\rangle$ made by the function $f$ under an input environment *Env*. Let $tprbnd_g$ be the bound inferred by the system for the function $g$ by inferring values for the holes that satisfy the assume-guarantee obligations. By the soundness of the inference algorithm, the expression $tprbnd_f[y/param_P(f)]$ will evaluate to a value greater than the value of the tpr for the call $(f\ y)$ under the environment reaching the call site: $Env'$. By the obligation 2.8.I, $tprbnd_g \geq tprbnd_f[y/param_P(f)]$ holds for all environments satisfying the path condition to the call site, and hence specifically for $Env'$. Thus, the value of $tprbnd_g$ under $Env'$ (and hence under *Env*) upper bound the value of tpr for the recursive call $(f\ y)^{\ell}$ under $Env'$. By induction, it also upper bounds the value of tpr for every transitive recursive call made by the function $g$ under the input environment *Env*.

**Upper Bounding Steps using TPR and Rec** In the above obligation, *tprtmpl* denotes the upper bound (with holes) provided by the user for the tpr resource. The annotation @compose informs the system to use a divide-and-conquer reasoning for inferring the steps bound. In this case, the system independently infers the tpr and rec bounds and infers the step bound using the following assume-guarantee obligation:

(2.8.I) For each function $f$ in $P$, $\models_P (\mathsf{steps}^\sharp \leq tprbnd_f \cdot recbnd_f) \rightarrow \mathsf{steps}^\sharp \leq stepstmpl_f$

In the above equation, $\mathsf{steps}^\sharp$ represents the instrumented term of function $f$ corresponding to the steps resource. $tprbnd_f$ and $recbnd_f$ represents the bounds of tpr and rec resources inferred by the system for the function $f$ based on the user-provided templates. $stepstmpl_f$ denotes the template provided by the user for the steps usage of the function $f$. Note that while the above assume-guarantee obligation involves nonlinear multiplication of *tprbnd* and *recbnd*, the condition is typically much easier to solve since the terms (or monomials) in *stepstmpl* will at least include the terms in the product of *tprbnd* and *recbnd*. It is to be noted that this divide-and-conquer reasoning is useful in every scenario where the steps bound that is needed to be established has a higher degree (counting only multiplication operations) compared to the bounds of the tpr and rec resources.

I would like remark that in the evaluations that were carried out using the system every steps bound that was established with this reasoning was also provable directly by the inference algorithm. Nevertheless, I believe that on large real-world programs the reduction in the instantiation of axioms of multiplication achieved by this divide-and-conquer reasoning may provide significant speed ups. Furthermore, by breaking down the proof argument for nonlinear steps bounds, this reasoning also offers a fine-grained control to the users of the tool for establishing nonlinear bounds.

## 3.9 Amortized Analysis

In this section, I briefly describe how the techniques presented here can be used to verify amortized bounds. Computing amortized bound does not require any additional machinery or extensions to the algorithm. Instead, the potential functions necessary for establishing amortized bounds can be directly expressed as invariants in the contracts. For instance, consider an amortized data structure with a method **def** $p\,x := \tilde{e}$. Let $\phi$ denote a potential function from the state of the data structure to a value (e.g. an Int). If the method $p$ has an amortized steps count $f(x)$ and if the potential function is chosen correctly then the worst-case steps count of the method $p$ is upper bounded by $f(x) + (\phi(x) - \phi(\text{res}))$, where res is the data structure that is returned by $p$. This worst-case steps bound can be specified in the postcondition of the method $p$. Once this bound is established, a client that invokes the method $p$ $n$-times can easily be established to have a worst-case steps count of $\sum_{i=1}^{n} f(x_i)$, where $x_i$ is the input of the $i^{th}$ iteration. Note that it suffices to specify a template for the potential functions as well, since any constant factors involved in the potential functions can be automatically inferred by tool.

For a concrete example consider a binary increment of an unbounded natural number shown below. Here, BigNum represents the unbounded natural number and is implemented as a list of zeros and ones. The function ones counts the number of ones in a BigNum.

```
def incr(l: BigNum) : BigNum = {
  l match {
    case Nil() ⇒ Cons(One(), l)
    case Cons(Zero(), tail) ⇒ Cons(One(), tail)
    case Cons(_, tail) ⇒
      Cons(Zero(), incrtail))
  }
} ensuring (res ⇒ steps ≤ ? ∗ ones(res) + ? ∗ ones(l) + ?)

def ones(l: BigNum) : Int = {
  l match {
    case Nil() ⇒ 0
    case Cons(Zero(), tail) ⇒ ones(tail)
    case Cons(_, tail) ⇒ 1 + ones(tail)
  }
```

```
}
```

The function ones is essentially a potential function, and the amortized steps count of incr is a constant. The system infers the bound $10(\text{ones}(\text{l}) - \text{ones}(\text{res})) + 18$ for the incr method. To formally prove the amortized constant running time of the method, one can define a (most-general) client as shown below that invokes incr multiple times as given by a parameter nop. The amortized constant steps count of incr would follow if it is established that the worst-case steps count of the client is linear in nop (plus the initial potential).

```
def client(nop: BigInt, l: BigNum) : BigNum = {
  if(nop == 0) l
  else
   client(nop−1, incr(l))
} ensuring (res ⇒ steps ≤ ? ∗ nop + ? ∗ ones(l) + ?)
```

For the above function, our system inferred the bound $\text{steps} \leq 10\text{ones}(\text{l}) + 23\text{nop} + 2$. If there is more than one method in the data structure, the client function shown above could be extended to accept an unconstrained parameter that specifies the method that needs to be executed in each iteration. As readers may notice, synthesizing a client function is quite mechanical when the methods of an amortized data structures are known precisely. Whether such client functions can be synthesized completely automatically within the system is a subject of future work.

# 4 Supporting Higher-Order Functions and Memoization

There is no general complexity-preserving translation
of lazy programs into an eager functional language.
— R.Bird, G.Jones and O.de Moor

In this chapter, I extend the technique presented in the earlier chapters to programs that use higher-order features and memoization. Recall that the core language syntax shown in Figure 2.1 supported an annotation @memoize on functions. This annotation serves to mark functions that have to be memoized. Such functions are evaluated exactly once for each distinct input passed to them at run time. The main challenge that arises in the presence of these features is that while the source programs use rich abstractions such as memoization and higher-order functions, the SMT solvers support more basic logical theories like theory of uninterpreted functions, algebraic datatypes, and arithmetics. Much like a compiler that translates a high-level program down to machine instructions, the goal of the system is to translate these programs to formulas efficiently decidable by SMT solvers. However, this has to be accomplished without sacrificing completeness.

**Realizing Lazy Evaluation** Memoized named functions when combined with first-class functions are more general than *lazy suspensions* [Dolstra, 2009]. Lazy suspensions can be implemented using lambdas with unit parameter and a memoized function force defined as: @memoize **def** force (f: Unit⇒T)= (f Unit). A closure $\lambda x.e$, where $x$ has unit type, creates a suspension of $e$, and (force $\lambda x.e$) evaluates $e$ and memoizes the result for subsequent forces. Note that memoization in our case uses structural equality to compare closures, which does not model lazy evaluation accurately, since lazy evaluation stores the cached values with the instances of closures. In other words, lazy evaluation memoizes closures using reference equality. Though the language does not support *reference equality* but only structural equality, the former can be encoded in the programs of our language by adding a unique identifier to the datatypes that represent closures. Our implementation, however, allows the input programs to use the *lazy val* keyword of Scala and performs this transformation to memoized functions

internally. I therefore do not explicitly formalize lazy evaluation in the core language.

**Challenges in Incorporating Cache** In this chapter, I extend the semantics presented in section 2.3 with an built-in cache that memoizes the values of functions calls invoking memoized functions. With this extension it becomes possible to define the semantics of the specification constructs that are meant for expressing properties that depend on the memoization state. The extended semantics is presented in section 4.1. However, the downside of these extension is that the language is no longer referentially transparent with respect to the changes in the cache, though it is with respect to the changes in the heap. This is somewhat expected since the language allows constructs such as cached that query the cache. However, these constructs are only restricted to the specification expressions and are not a part of the source expression.

Surprisingly, even the source language expressions loose full referential transparency with these constructs. This is because of expressions with contracts. For instance, when a source language expression invokes a function, the function returns a value only if the contracts of the callee hold. If the callee uses contracts that are anti-monotonic with respect to the cache e.g. !cached($f\ x$), which may evaluate to true in a smaller cache but false in a larger cache then a function that returned a value in a smaller cache may be undefined in a larger cache. Despite this undesirable effect constructs like cached are indispensable for verifying non-trivial benchmarks with memoization (as illustrated by the primetake example of 1.3). To eliminate this undesirable effect and restore referential transparency of the source expressions, I introduce the notion of cache monotonic properties (in section 4.2), which are properties that evolve monotonically with the changes in the cache. Interestingly, in almost all cases the properties that are needed to establish resource bounds are (or can be converted to) cache monotonic properties. E.g. the concrUntil property of 1.3. Intuitively, this phenomenon seems to result from anti-monotonicity of resource usage with respect to the increase in the cache size. That is, the resource usage of an expression monotonically decreases as it is evaluated under a cache that has more entries. Section 4.2 defines these properties formally and proves the referential transparency of the source expressions.

I now formally describe the extended semantics and subsequently discuss our verification approach.

## 4.1 Semantics with Memoization and Specification Constructs

In this section, I define the semantics of the higher-order specification constructs and memoization constructs. The state of the evaluation is extended by a built-in cache that memoizes the values of functions calls invoking memoized functios, which are functions with @memoize annotation. Let $Mem_P \subseteq Fids$ denote the set of memoized functions in a program $P$. The semantic domains are extended with a cache as described below.

$$\textsc{NonMemoizedCall}\quad \frac{f \in \mathit{Fids} \qquad f \notin \mathit{Mem}_P \qquad \Gamma \vdash (f\ \sigma(x)) \Downarrow_p v, \Gamma'}{\Gamma \vdash f\ x \underset{c_{call} \oplus p}{\Downarrow} v, \Gamma'} \qquad \textsc{MemoCallHit}\quad \frac{f \in \mathit{Mem}_P \qquad ((f\ \sigma(x)), v) \in_H C}{\Gamma : (C, H, \sigma) \vdash f\ x \underset{c_{hit}}{\Downarrow} v, \Gamma}$$

$$\textsc{MemoCallMiss}\quad \frac{f \in \mathit{Mem}_P \qquad u = \sigma(x) \qquad \neg\big((f\ u) \in_H \mathit{dom}(C)\big) \qquad \Gamma \vdash (f\ u) \Downarrow_p v, (C', H', \sigma')}{\Gamma : (C, H, \sigma) \vdash f\ x \underset{c_{miss} \oplus c_{call} \oplus p}{\Downarrow} v, C'[(f\ u) \mapsto v], H', \sigma}$$

$$\textsc{Cached}\quad \frac{v \Leftrightarrow \big((f\ \sigma(x)) \in_H \mathit{dom}(C)\big)}{\Gamma : (C, H, \sigma) \vdash \mathsf{cached}(f\ x) \Downarrow_0 v, \Gamma} \qquad \textsc{in}\quad \frac{C' = \mathit{extract}(\sigma(x)) \qquad (C', H, \sigma) \vdash e \Downarrow_p v, \Gamma'}{\Gamma : (C, H, \sigma) \vdash \mathsf{in}(e, x) \Downarrow_0 v, \Gamma'}$$

$$\textsc{Star}\quad \frac{\Gamma \vdash e \Downarrow_p v, \Gamma' : (C', H', \sigma')}{\Gamma \vdash e^\star \Downarrow_0 v, (C, H', \sigma')}$$

$$\textsc{Fmatch}\quad \frac{H(\sigma(x)) = (\lambda x.f_i\ (x, y), \sigma_1)) \qquad (C, H, \sigma[y_i \mapsto \sigma_1(y)]) \vdash e_i \Downarrow_p v, (C', H', \sigma')}{\Gamma : (C, H, \sigma) \vdash x\ \mathsf{fmatch}\ \left\{\ \lambda x_1.f_i\ (x_i, y_i) \Rightarrow e_i\right\}_{i=1}^{n} \Downarrow_0 v, (C', H', \sigma)}$$

$$\textsc{Contract}\quad \frac{\begin{array}{c}\Gamma \vdash \mathit{pre} \Downarrow_p \mathit{true}, \Gamma_1 \qquad \Gamma \vdash e \Downarrow_q v, \Gamma_2 : (C_2, H_2, \sigma_2) \\ (C_2, H_2, \sigma_2[R \mapsto q, \mathsf{res} \mapsto v, \mathsf{inSt} \mapsto \mathit{convert}(C), \mathsf{outSt} \mapsto \mathit{convert}(C_2)]) \vdash \mathit{post} \Downarrow_r \mathit{true}, \Gamma_3\end{array}}{\Gamma : (C, H, \sigma) \vdash \{\mathit{pre}\}\ e\ \{\mathit{post}\} \Downarrow_q v, \Gamma_2}$$

where $R \in \{\mathsf{steps}, \mathsf{alloc}, \mathsf{stack}, \mathsf{depth}\}$

---

Cost function definition for steps: $c_{miss)} = 2$, $c_{hit} = 1$

Cost function definition for alloc: $c_{miss} = 1$, $c_{hit} = 0$

Figure 4.1 – Operational semantics of higher-order specifications and memoization.

**Semantic domains** The state of an interpreter evaluating expressions of our language is now a quadruple consisting of a cache $C$, a heap $H$, an assignment of variables to values $\sigma$, and a set of function definitions. A cache is a partial function from function calls to their results. The cache $C$ of the environments has the property that every key of the cache, which is a concrete function call in *FVal*, is mapped to the result of the call. This is captured by a domain invariant presented shortly. I denote this new environment with a cache as $\mathit{Env}^c$.

$$
\begin{aligned}
&u, v \in Val = \mathbb{Z} \cup Bool \cup Adr \\
&DVal = Cids \times Val^* \\
&Clo = Lam \times Store \\
&FVal = Fids \times Val \\
&H \in Heap = Adr \rightarrowtail (DVal \cup Clo) \\
&\sigma \in Store = Vars \rightarrowtail Val \\
&C \in Cache = FVal \rightarrowtail Val \\
&\Gamma \in Env^c \subseteq Cache \times Heap \times Store \times Program
\end{aligned}
$$

**Def 3** (**Domain Invariants**). *A quadruple* $(C, H, \sigma, P)$ *is an element of* $Env^c$ *iff the domain invariants of Definition 1 hold for* $H$ *and* $\sigma$*, and the following properties hold for the* $C$*.*

*(a)* $\neg\exists\{k, k'\} \subseteq dom(C).\ k \neq k' \wedge k \underset{H}{\approx} k'$

*(b)* $\forall (k, v) \in C. \exists C', C'', H'$ *s.t.* $C' \sqsubseteq C'' \sqsubseteq C \wedge H' \sqsubseteq H'' \sqsubseteq H$
$\wedge (C', H', \emptyset, P) \vdash k \Downarrow v, (C'', H'', \emptyset, P)$

The first invariant ensures that every key in the cache is unique modulo structural equality. The second invariant ensures the consistency of the cache values. For every key-value pair in the cache, there is a cache and a heap: $C'$ and $H'$ that is smaller than the current cache and heap: $C$ and $H$, respectively, in which the key evaluates to the value it is bound to. (The empty set $\emptyset$ denotes an empty store in the above definition.) This essentially means that the key and the value was added to the cache during a previous evaluation consistent with the current state. Later a couple of more derived domain invariants are established and proved using the operational semantics.

### 4.1.1 Semantic Rules

Figure 4.1 shows the semantic rules for the constructs of the language that use memoization, and new specification constructs that deal with higher-order and memoization features. There are three types of direct calls rules: a call to function that is not memoized NONMEMOIZED-CALL, which is same as a direct call, a call to memoized function that is a hit in the cache: MEMOCALLHIT and that is a miss in the cache: MEMOCALLMISS. The MEMOCALLMISS is the *only* rule that updates the cache. Every semantics rule presented in Figure 2.2 other than rule CONTRACT remain unchanged, except that the input and the output environments now also have a cache, so they omitted from the Figure 4.1. The reachability relation defined in Figure 2.3 extends to the new semantic rules shown here in a straightforward way.

**Memoized Call Semantics** Calling a memoized function involves as a first step querying the cache for the result of the call. In case the result is not found, the callee is invoked, and the cache is updated once (and if) the callee returns a value. Querying the cache involves comparing arguments of the call for equality. For this purpose, the semantics uses a *lookup*

*relation* $\in_H$ that uses structural equivalence to lookup the cache defined as follows:

$$(f\ u) \in_H dom(C) = \exists u' \in Val.(f\ u') \in dom(C) \wedge u' \underset{H}{\approx} u$$

As with other operations, the resource usage of searching and updating the cache is parameterized by the cost functions $c_{hit}$ and $c_{miss}$. To calculate the steps resource, lookup and update are considered as unitary steps, and hence is defined as $c_{miss} = 2$ (as it involves a lookup and an update operation) and $c_{hit} = 1$. In general, $c_{miss}$ and $c_{hit}$ may be changed to depend on the values of the arguments that are looked up.

I would like to remark that the cost of the *lookup* operation depends on several factors such as the implementation of the cache, whether or not datatypes are *hash-consed* etc. and hence may require multiple interpretations. This definition for cost parameters was chosen in the implementation since, in the benchmarks we target, functions that memoize data structures are often methods of the data structure that relied on lazy fields for memoization. The cost of memoization in this case is a small constant. However, in principle, the definition of the cost function could be changed to run the system on a different memory model.

Note that also using structural equality to lookup the cache is not a limitation as one can emulate reference equality by associating unique identifiers with datatypes or closures.

**Specification Constructs**

I now discuss the semantics of the specification constructs. First, let us consider the constructs that deal with the state of the cache. The cost of all the specification constructs described below is zero since they are syntactically excluded from being part of the implementation of functions (see the language syntax shown in Figure 2.1), which renders their resource usage irrelevant. They only serve to specify properties about the implementation.

The construct cached($f\ x$). evaluates to true in an environment $\Gamma$ iff the call $f$ is cached for the value of $x$ in $\Gamma$.

The construct in(e, x) evaluates an expression in a cache state given by $x$. Users of the system may read the cache state at a program point through the constructs inSt and outSt (described shortly). The function *convert* encodes a given cache as a value of the program and the function *extract* decodes the cache from the value of the program. (The actual representation is not important, as these constructs are only meant for specification and would eventually be handled by the verifier and are not meant to be executed at runtime.) This is a powerful specification construct as it allows specifying the value and resource usage of an expression under different caches as necessary, and would essential for defining cache monotonic properties (described shortly).

The construct $e^\star$ computes the result of an expression $e$ without caching the result of $e$ for reuse. This is a side-effect-free operation (even in terms of resource usage) that is to be used

in places where only the result of the expression is relevant (within specifications). This is primarily meant to alleviate some verification overhead that would otherwise result by the use of a expressions that updates the cache.

Now consider the modified CONTRACT rule. The two new entities here are inSt and outSt. The construct inSt is used by expressions in the postcondition to refer to the state of the cache at the beginning of the expression, and outSt to refer to the cache at the end of evaluating the body of the triple. These constructs are bound to their respective value (after conversion to the domain of values) in the postcondition. They can be passed as argument to the in construct described earlier. Note that even though the construct in(e, x) allows evaluating an expression under a cache given by $x$. The cache can only be obtained either through inSt or outSt expressions. Essentially, in(e, x) is used to evaluate an expression under a cache encountered previously during the evaluation. Notice that as in the case of heap, any changes to the state of the cache introduced by the evaluation of pre-or post-condition is ignored by the rule CONTRACT.

**Structural Matching on Closures** Consider now the construct fmatch of the form $x$ fmatch $\{ \lambda x_i.f_i\ (x_i, y_i) \Rightarrow e_i \}_{i=1}^{n}$. It performs structural matching on closures, i.e, matching based on structural equality. For instance, this expression matches $x$ to the first case if $x$ evaluates to a closure of the form: $(\lambda x.f_1\ (x, y), [y \mapsto u])$. It binds the variable $y$ in the match case to the value $u$, and evaluates $e_1$ using the new binding.

The structural matching construct may be seen as a sequence of if-then-else expressions in which the guards are structural-equality comparisons where the captured variables are existentially quantified. That is, the matching construct $x$ fmatch $\{ \lambda x_i.f_i\ (x_i, y_i) \Rightarrow e_i \}_{i=1}^{n}$ is actually equivalent to

$$\textbf{if } (\exists y_1.x \text{ eq } \lambda x_1.f_1\ (x_1, y_1))\ e_1 \textbf{ else if } (\exists y_2.x \text{ eq } \lambda x_2.f_2\ (x_2, y_2))\ e_2 \textbf{ else } \cdots$$

Note that since our language does not support existential quantifiers the matching construct, in fact, makes the specification language more expressive. This construct is useful for specifying the requirements on the captured variables of the closures that are passed into a function. For instance, the following code snippet, shown in the syntax of the core language, shows a function foo that accepts a closure whose target is the function divide and whose captured argument (which is the divisor) is a positive value. The specification function posArgs defined using the fmatch construct returns true if and only if the parameter closure invokes the divide function and its captured argument is positive.

```
def posArgs(cl: Int ⇒ Int): Boolean :=
  cl fmatch {
    λx. divide (x, y) ⇒ y ≥ 1
    _ ⇒ false
  }
```

```
def foo(cl: Int ⇒ Int, x: Int): Int :=
  { posArgs(cl) } cl(x) { true }
```

This fmatch construct was very useful in specifying the invariants of a lazy, bottom-up merge sort algorithm (discussed in section 5), wherein a balanced tree of closures of the merge function are created and forced on demand. It was also useful in expressing the invariants of cyclic streams, such as fibonacci stream and hamming stream detailed in section 5, where it was necessary to state that the arguments to the closures denoting the tail of a stream *s*, is the stream *s* itself. The following code snippet shows this cyclic stream property for a fibonacci stream defined using a zipWith function ([Vasconcelos et al., 2015], [Bird and Wadler, 1988]). The definition of the recursive datatypeSCons is shown in Figure 1.3.

```
def cyclicStream(s: SCons): Boolean =
  let first := s in
   let second := first.tail in
    let third := second.tail in
     third.tfun fmatch {
       λx. zipIWith (f, xs, ys) ⇒ (xs eq first) && (ys eq second)
       _ ⇒ false
     }
```

## 4.2 Referential Transparency and Cache Monotonicity

While the referential transparency or purely functional behavior of the first-order language considered this far was quite evident. The introduction of cache and the specification constructs has made this property more trickier. The language allows expressions to query the state of the cache e.g. using the construct cached. While this is indispensable for specifying properties about the state of the cache, this also makes the expressions of the language *not* referentially transparent. However, as captured by the syntax shown in Fig. 2.1, these constructs are restricted to the specifications.

The source expressions $E_{src}$ of our language only exhibit a weak form of referential transparency with respect to the changes to the cache. The weak referential property guarantees that if a source expression evaluates to a value $u$ at some point in the evaluation, then *if* it evaluates to a value $v$ at a later point in the evaluation (for the same or equivalent argument values) then $u$ and $v$ are equivalent i.e, structurally similar (see section 2.3.3). However, the evaluation at the later point may be undefined due to a violation in the contracts. This is because the expression may have a contract such as !cached($f\ x$) which may hold in a smaller cache but not in a larger cache. (Note that the size of the cache increases monotonically during an evaluation.) This is problematic since it is important to preserve the referential transparency property of the memoized expression to ensure that memoization does not influence the evaluation results. For this purpose, the system relies on the notion of *cache monotonicity*.

**Cache Monotonic Properties** Informally, a boolean-valued expression $pr \in E_{spec}$ is a cache monotonic iff whenever it holds in an environment with a cache $C_1$, it also holds in all environments where the cache has more entries than $C_1$. These properties are interesting because once established they can be assumed to hold at any subsequent point in the evaluation. (These are similar to *heap-monotonic* type states introduced by Fähndrich and Leino [Fähndrich and Leino, 2003]). We find that in almost all cases the properties that are needed to establish resource bounds are (or can be converted to) cache monotonic properties. E.g. the concrUntil property. Intuitively, this phenomenon seems to result from anti-monotonicity of resource usage with respect to the increase in the cache size. That is, the resource usage of an expression monotonically decreases as it is evaluated under a cache that has more entries. Formally, an expression $e$ is cache monotonic iff $\forall\{\Gamma_1, \Gamma_2\} \subseteq Env^c$.

$$(\Gamma_1 \sqsubseteq \Gamma_2 \wedge \Gamma_1 \vdash e \Downarrow true) \implies \Gamma_2 \vdash e \Downarrow true$$

where $(C_1, H_1, \sigma_1, P) \sqsubseteq (C_2, H_2, \sigma_2, P) \triangleq C_1 \sqsubseteq C_2 \wedge H_1 \sqsubseteq H_2 \wedge \sigma_1 \sqsubseteq \sigma_2$.

**Cache Monotonic Programs** In order to guarantee full referential transparency of the source expressions, we impose the restriction that the contracts of memoized functions in the all acceptable input programs should be *cache monotonic*. This property is soundly enforced by translation to a model program, which is discussed in section 4.4. The property guarantees that if a source expression evaluates to a value $u$ at a point in the evaluation, then it *will* evaluate to a value $v$ at a later point in the evaluation (for the same or equivalent argument values) such that $u$ and $v$ are structurally similar. That is, memoization has absolutely no effect on the result of source expressions. In the sequel I formally establish this property.

## 4.3 Proof of Referential Transparency

**Structural Abstraction Relation** Similar to structural simulation relation between two environments, let $\lesssim$ denote a structural abstraction relation between two environments as follows. $\Gamma_1 \lesssim \Gamma_2$ iff $\Gamma_2$ has at least as much cache entries and $\sigma$ entries as $\Gamma_1$ modulo structural equality.

$$(C_1, H_1, \sigma_1, P) \lesssim (C_2, H_2, \sigma_2, P) \triangleq C_1 \underset{H_1,H_2}{\lesssim} C_2 \wedge \sigma_1 \underset{H_1,H_2}{\lesssim} \sigma_2$$

where, $\sigma_1 \underset{H_1,H_2}{\lesssim} \sigma_2$ *iff* $\forall x \in dom(\sigma_1).\sigma_1(x) \underset{H_1,H_2}{\approx} \sigma_2(x)$, and

$$C_1 \lesssim C_2 \text{ iff } \forall k \in dom(C_1).\ \exists k' \in dom(C_2).k \underset{H_1,H_2}{\approx} k' \wedge C_1(k) \underset{H_1,H_2}{\approx} C_2(k')$$

Note that $\sqsubseteq$ is a stronger relation than $\lesssim$. Furthermore, a cache monotonic property with respect to $\sqsubseteq$ is also monotonic with respect to the relation $\lesssim$. Intuitively, this is because, if $\Gamma_1 \lesssim \Gamma_2$, there exists a $\Gamma_3$ such that $\Gamma_1 \approx \Gamma_2$ and $\Gamma_1 \sqsubseteq \Gamma_2$. Hence, a cache monotonic property that evaluates to true under $\Gamma_1$ will evaluate to true under $\Gamma_2$, and hence also under

$\Gamma_3$ (Lemma 1).

The following lemma establishes that a source expression evaluated under two environments related by $\lesssim$ should produce structurally similar values, if the evaluation produces any value at all in the smaller environment. Note that the heaps and caches that may arise during an execution are related by $\sqsubseteq$ (Lemma 7) and hence are also related by the weaker relation $\lesssim$. Hence, the following property established referential transparency of the source expressions of the language.

**Lemma 18.** *Let $P$ be a program where for all* **def** $f\ x := \{p\}\ b\ \{s\}$ *in $P$, $p$ and $s$ are cache monotonic properties. Let $\Gamma_1 : (C_1, H_1, \sigma_1, P)$ in $Env^c$. For all expression $e_s \in (E_{src} \cup FVal)$, if $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ then $\forall \Gamma_2 : (C_2, H_2, \sigma_2, P) \in Env^c$ such that ${\Gamma_1}' \lesssim \Gamma_2$, we have $\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}' \wedge u \underset{H_1', H_2'}{\approx} v$*

*Proof.* I prove the lemma using structural induction on the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$. It suffices to consider the rules corresponding to source expressions and ignore specification expressions. Say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses one of the base cases, namely the rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHIT. Note that the rule CACHED is not a part of the source expressions (see Fig. 2.1) and thus can be excluded from the base cases. Firstly, we know that $\Gamma_1 \sqsubseteq {\Gamma_1}'$. (The store components of $\Gamma_1$ and ${\Gamma_1}'$ are identical.) Therefore, $\Gamma_1 \lesssim \Gamma_2$. Every case other than MEMOCALLHIT uses only the heap and the store (and not the cache). Since $\Gamma_1 \lesssim \Gamma_2$, the free variables in the expressions are bound to structurally similar values in $\Gamma_1$ and $\Gamma_2$. It is easy to see that in each of the cases, the same rule that applied in $\Gamma_1$ will also apply in $\Gamma_2$ and that the resulting values are also structurally similar. Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u$ uses MEMOCALLHIT. Therefore, $e$ is of the form $(f\ x)$ and $\sigma_1(x) \underset{H_1}{\approx} k$ where $k$ is a key in the cache $C_1$. Since $\Gamma_1 \lesssim \Gamma_2$, $\sigma_1(x) \underset{H_1, H_2}{\approx} \sigma_2(x)$ and there exists a $k' \in dom(C_2)$ such that $k \underset{H_1, H_2}{\approx} k'$. By the properties of $\approx$, $\sigma_2(x) \underset{H_2}{\approx} k'$. Hence, the evaluation of $\Gamma_2 \vdash e_s \Downarrow u$ must also use the rule MEMOCALLHIT. In both cases, the value of the expression is looked up from the corresponding caches, and hence are structurally similar (by the definition of $\lesssim$).

Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses one of the following inductive cases: LET, MATCH, CONCRETECALL, NONMEMOIZEDCALL. If $\Gamma_1 \vdash e_s \Downarrow u, {\Gamma_1}'$ uses any of these rule RULE then $\Gamma_2 \vdash e_s \Downarrow v, {\Gamma_2}'$ will also have to use the same rule, which is determined by the syntax of the expression (see Fig. 2.2). Say now $\langle \Gamma_1, e_s \rangle \rightsquigarrow \langle \Gamma_3, e' \rangle$ and $\langle \Gamma_2, e_s \rangle \rightsquigarrow \langle \Gamma_4, e' \rangle$. Firstly, the environment $\Gamma_3$ and $\Gamma_4$ are obtained from a prior big-step evaluation given by an antecedent of RULE, after possible updations to the store component. Let $\Gamma_3 \vdash e' \Downarrow \_, {\Gamma_3}'$. By Lemma 7 and the given facts, $C_1 \sqsubseteq C_3 \sqsubseteq C_3' \sqsubseteq C_1' \underset{H_1', H_2}{\lesssim} C_2 \sqsubseteq C_4$. Consider the store component of $\Gamma_3$, which is identical to ${\Gamma_3}'$, and $\Gamma_4$ namely $\sigma_3$ and $\sigma_4$. Any new mappings added to the store components depend on the prior big-step reductions in the antecedent of RULE, which satisfies the induction hypothesis. Thus, the new entries added are structurally similar. Hence, $\sigma_3' = \sigma_3 \underset{H_3', H_4}{\lesssim} \sigma_4$. Therefore, ${\Gamma_3}' \lesssim \Gamma_4$. By induction hypothesis, $e'$ evaluates to structurally similar values in $\Gamma_3$ and $\Gamma_4$. Since this holds for every antecedent of RULE and since in all inductive cases the

result of the rule is obtained directly from the *result* of an antecedent evaluation involving a source expression or function value (see Fig. 4.1, especially rule CONTRACT), both evaluations $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1{}'$ and $\Gamma_2 \vdash e_s \Downarrow v, \Gamma_2{}'$ produce structurally similar results. That is, $u \underset{H_1', H_2'}{\approx} v$.

Now say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1{}'$ uses the MEMOCALLMISS rule. In this case, since $C_2$ has more entries than $C_1$, $\Gamma_2 \vdash e_s \Downarrow v, \Gamma_2{}'$ will use the rule MEMOCALLHIT, as explained below. In this case, we know that $e_s = (f\ y)$ and $((f\ \sigma_1'(y)), u) \in C_1'$. (Recall that the rule MEMOCALLMISS records the function value and the result of the evaluation in the cache.) Since $C_1' \underset{H_1', H_1}{\lesssim} C_2$, there exists an entry $(k, v) \in C_2$ such that $((f\ \sigma_1'(y)) \underset{H_1', H_2}{\approx} k$ and $u \underset{H_1', H_2}{\approx} v$. Since $\sigma_1' \lesssim \sigma_2$, $\sigma_1'(y) \underset{H_1', H_2}{\approx} \sigma_2(y)$. Thus $(f\ \sigma_1'(y)) \underset{H_1', H_2}{\approx} (f\ \sigma_2(y))$. By the property of $\approx$, $(f\ \sigma_2(y)) \underset{H_2}{\approx} k$. By the definition of MEMOCALLHIT, the result of the evaluation under $\Gamma_2$ is $v$. Since $H_2 \sqsubseteq H_2'$, $u \underset{H_1', H_2'}{\approx} v$ which implies the claim.

Say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1{}'$ uses the rule CONTRACT. In this case, $e_s$ is of the form $\{p\}\ e_b\ \{s\}$. As per the language syntax, this means that $e_s$ is the body of a function definition in $P$. It is given that $p$ is cache monotonic. By the definition of the rule CONTRACT, $\Gamma_1 \vdash p \Downarrow \mathit{true}$. Since $\Gamma_1 \sqsubseteq \Gamma_1{}' \lesssim \Gamma_2$, by the cache monotonicity property, $\Gamma_2 \vdash p \Downarrow \mathit{true}$. Since $e_b \in E_{src}$, by inductive hypothesis, $\Gamma_2 \vdash e_b \Downarrow v, \Gamma_2{}'$ and $u \underset{H_1', H_2'}{\approx} v$. Now $\Gamma_1{}' \lesssim \Gamma_2 \sqsubseteq \Gamma_2{}'$. Since $s$ is also cache monotonic, $\Gamma_1{}' \vdash s \Downarrow \mathit{true}$ , which holds by the definition of CONTRACT, implies that $\Gamma_2{}' \vdash s \Downarrow \mathit{true}$. Hence, all antecedents of the rule CONTRACT are satisfied under $\Gamma_2$ for expression $e_s$. Hence $\Gamma_2 \vdash e_s \Downarrow v, \Gamma_2{}'$ and $u \underset{H_1', H_2'}{\approx} v$. Hence the claim. □

**Strong Cache Correctness** I now define a domain invariant that guarantees that for a program $P$ where function definitions have cache monotonic contracts, for every environment $\Gamma$, every key in the cache of the environment *will* evaluate to a value that it is bound to under $\Gamma$. This invariant, denoted *CacheCorr*, is formally defined below.

$$\mathit{CacheCorr}(\Gamma) \triangleq \forall k \in \mathit{dom}(C).\left(\Gamma \vdash k \Downarrow_p v, (C', H', \sigma', F)\right) \wedge v \underset{H'}{\approx} C(k)$$

The following lemma establishes that *CacheCorr* is an domain invariant.

**Lemma 19.** *Let $P$ a program be such that for all* **def** $f\ x := \{p\}\ b\ \{s\}$ *in $P$, $p$ and $s$ are cache monotonic properties. For all expression $e$, for all $\Gamma_1 : (C_1, H_1, \sigma_1, P)$ in $Env^c$,*

$$\mathit{CacheCorr}(\Gamma_1) \wedge \Gamma_1 \vdash e \Downarrow u, \Gamma_1{}' \implies \mathit{CacheCorr}(\Gamma_1{}')$$

*Proof.* This lemma will be proved using structural induction over the evaluation $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1{}'$. First consider the base cases: rules CST, VAR, PRIM, EQUAL, CONS, LAMBDA, MEMOCALLHIT and CACHED. In each of these cases, either the input and output environments are identical, or the output environment has one new binding in the heap. By Lemma 8, the claim holds in

all the base cases.

Say the evaluation $\Gamma_1 \vdash e_s \Downarrow u, \Gamma_1'$ uses one of the following inductive cases: LET, MATCH, CONCRETECALL, NONMEMOIZEDCALL, and CONTRACT. First, note that introducing new bindings to the store $\sigma$ does not affect the property *CacheCorr*, as the definition of *CacheCorr* does not use $\sigma$. This together with the inductive hypothesis imply that all the environments used in the antecedent of all the rules satisfy *CacheCorr*. In all the above listed rules the heap and cache components of the output environment are obtained directly from an antecedent. Therefore by inductive hypothesis the environment $\Gamma_1'$ satisfies the property *CacheCorr*.

I now use the Lemma 18 to establish that the *CacheCorr* property holds for the output environment of the rule MEMOCALLMISS. Let $k \in FVal \cap dom(C_1)$ be a key in the cache $C_1$. By the domain invariants, there exists a $H_0 \sqsubseteq H_1$ and $C_0 \sqsubseteq C_1$ such that $\Gamma_0 \vdash k \Downarrow u_0, \Gamma_0'$, where $u_0 = C_1(k)$, $\Gamma_0 = (C_0, H_0, \{\})$, $\Gamma_0' = (C_0', H_0', \{\})$ and $C_0' \sqsubseteq C_1$. Since $C_0' \sqsubseteq C_1 \sqsubseteq C_1'$ and $\{\} \sqsubseteq \sigma_1'$, $\Gamma_0' \lesssim \Gamma_1'$. Therefore by Lemma 18,

$$\Gamma_1' \vdash k \Downarrow w, \Gamma_4 \wedge u_0 \underset{H_0', H_4}{\approx} w$$

$$\implies C_1(k) \underset{H_1', H_4}{\approx} w, \quad \text{since } C_1(k) = u_0 \text{ and } H_0' \sqsubseteq H_1 \sqsubseteq H_1'$$

$$\implies C_1(k) = C_1'(k) \underset{H_4}{\approx} w, \text{ since } C_1 \sqsubseteq C_1' \text{ and } H_1' \sqsubseteq H_4$$

Therefore, $CacheCorr(\Gamma_1')$

□

In the rest of the chapter, I assume that every environment in $Env^c$ satisfy the *CacheCorr* invariant, and every expression in the language is referentially transparent.

## 4.4 Generating Model Programs

As in the case of first-order programs, our approach through a series of transformations, reduces the problem of resource bound inference to invariant inference for a functional first-order programs. However, due to the higher-order and memoization features this phase is more involved than the instrumentation phases. This phase is referred to as the *model generation phase*, as is the subject of discussion of this section. In the following section, I described the generation of verification obligations for verifying the first-order program using an extended assume-guarantee reasoning, which then solved using the inference algorithm described in the earlier chapter in section 3.3.

The goal of the model generation phase is to generate a first-order program with recursion that accurately models the resource usage of the input program without any abstraction, only using the first-order features discussed in the previous chapter (Chapter 3) and *sets*. The output of this phase is referred to as the *model program*. In particular, there are three reductions that

$$e_m \in E^\sharp \quad ::= \quad x \mid c \mid pr\ x \mid x \ \mathsf{eq}\ y \mid f\ x \mid C\ \bar{x} \mid Block_s \mid \{e_m\}\ e_m\ \{e_m\} \mid \{x\} \mid x \cup y \mid x \subseteq y \mid \mathsf{error}$$

$$\textsc{SetCons}\ \frac{v = \{\sigma(x)\}}{\Gamma : (H,\sigma) \vdash \{x\} \Downarrow v, \Gamma} \qquad \textsc{Union}\ \frac{v = \sigma(x) \cup \sigma(y)}{\Gamma : (H,\sigma) \vdash x \cup y \Downarrow v, \Gamma} \qquad \textsc{Contains}\ \frac{v \Leftrightarrow \left(\exists u' \in \sigma(y).\sigma(x) \underset{H}{\approx} u'\right)}{\Gamma : (H,\sigma) \vdash x \in y \Downarrow v, \Gamma}$$

$$\textsc{Subset}\ \frac{v \Leftrightarrow \left(\forall u \in \sigma(x).\exists u' \in \sigma(y).u \underset{H}{\approx} u'\right)}{\Gamma : (H,\sigma) \vdash x \subseteq y \Downarrow v, \Gamma}$$

Figure 4.2 – Syntax and semantics of the set operations used by the model programs

are handled by this phase: (a) Defunctionalization of higher-order functions to first-order functions [Reynolds, 1998]. (b) Encoding of cache as an expression that changes during the execution of the program, and (c) Instrumentation of expressions with their resource usage while accounting for the effects of memoization. I formally establish the soundness and completeness of the translation with respect to the operational semantics shown in Fig. 4.1 by establishing a bisimulation between the input program and the model program (Theorem 26). In the sequel, I start by discussing the syntax and semantics of the model programs.

**Syntax and Semantics of Model Programs** Figure 4.2 shows the syntax of the expressions of the model programs. The model programs belong to the first-order language fragment considered in Chapter 3, but it has two new features that were not a part of the core language: (a) set values and *set* primitives such as union $\cup$ and inclusion $\subseteq$, and (b) an error construct that halts the evaluation. Figure 4.2 shows the semantics of the new set operations. The values of the model language $Val^\sharp$ includes the values of the core language $Val$, and also sets of values of the core language ($SetVal = 2^{Val}$). The stores of the model language is a map from $Vars$ to $Val^\sharp$, whereas the closures $Clo$, datatypes $DVal$ and $H$ are constructed as before (see section 2.3) using elements of $Val$. The environments of the model language are triples of the form $(H, \sigma, P)$, which is same as $Env$ (which does not have the cache component). The resource usage of the set expressions are irrelevant as they are not a part of the source language and hence omitted from the Figure 4.2.

### 4.4.1 Model Transformation

**Illustrative Example** I explain the construction of the model program by illustrating it on the constant-time take operation on a stream shown in Fig. 4.3. Later in section 4.6 I use this example to also illustrate the assume-guarantee obligations generated by our system. Fig. 4.3 shows the take operation in the core language syntax In a real language, the function tail would be implemented as a lazy field of the SCons constructor as shown in Figure 1.3

```
1  type Stream := (SCons (BigInt, Unit ⇒ Stream), SNil)
2
3  @memoize
4  def tail s = s match {
5    SNil ⇒ SNil;
6    SCons (x, tfun) ⇒ (tfun Unit);
7  }
8
9  def take (n, s) =
10   { concrUntil(s, n) }
11   if (n ≤ 0) SNil
12   else {
13     s match {
14       SNil ⇒ SNil;
15       SCons (x, tfun) ⇒
16         let t := tail s in
17          let n1 := n − 1 in SCons(x, λa.take (n1, t));
18     }
19   }
20 { steps ≤ ? }
21
22 def concrUntil (s, i) = s match {
23    SNil ⇒ true;
24    SCons (x, tfun) ⇒
25      if (i ≤ 0) true
26      else
27        (Tail s) ∈ st ∧ concrUntil ((tail s).1, i−1))
28 }
```

Figure 4.3 – A constant-time, lazy take operation

of the introduction. But for the purpose of verification, we treat it as a memoized function with a single argument as shown here. The function concrUntil is similar to the Scala function shown in Figure 1.4 that checks if the tail function is memoized for the first n suffixes of a stream. Observe that the lazy take operation (unlike takePrimes) returns a *finite* stream with the first element and a suspension of take, which when accessed constructs the next element. It requires that the input stream is memoized at least until n in order to achieve a constant time bound. Otherwise, the call to tail at line 16 may result in a cascade of calls to take that were suspended previously. The challenge here is to verify that such cascade of calls cannot happen. The take operation with these contracts is in fact used by the Okasaki's persistent *Deque* data structure ([Okasaki, 1998] Page 111) that runs in worst-case constant time. Fig. 4.4 shows the model program that would generated by our approach and is explained in this section.

**Closure encoding** Closures in the source program are represented using algebraic datatypes in the model program in a way that preserves the structural equivalence of closures. This representation is shown by $Clo^{\sharp}{}_{P}$ in Figure 4.5 and is explained below. In Figure 4.5, $types_P$

```
1  type tStream := (Take (BigInt, Stream), Other BigInt)
2  type Stream := (SCons (BigInt, tStream), SNil)
3  type Dcache := (Tail Stream)
4
5  def tail♯ (s, st) = s match {
6    SNil ⇒ SNil;
7    SCons (x, tfun) ⇒ app (tfun, Unit, st);
8  }
9
10 def app (cl,x,st) = cl match{
11   Take (n1,s1) ⇒ take♯ (n1,s1,st);
12 }
13
14 def take♯ (n, s, st) =
15   { concrUntil♯ (s, n, st) }
16   if (n ≤ 0) (SNil, st, 3)
17   else {
18    s match {
19     SNil ⇒ (SNil, st, 5);
20     SCons (x, tfun) ⇒
21       let u := tail♯ (s, st) in
22        let nst := u.2 ∪ { (Tail s) } in
23         let u_cost := if ((Tail s) ∈ st) 1 else u.3 + 3 in
24          let ns := (SCons (x, Take (n − 1, u.1)) in
25           (ns, nst, u_cost + 10);
26    }
27   }
28 { res.3 ≤ ? }
29
30 def concrUntil♯ (s, i, st) = s match {
31    SNil ⇒ true;
32    SCons (x, tfun) ⇒
33      if (i ≤ 0) true
34      else
35        (Tail s) ∈ st ∧ concrUntil♯ ((tail (s, st)).1, i−1, st))
36 }
```

Figure 4.4 – Illustration of the translation on lazy take example

$$
\begin{array}{rcl}
Clo^\sharp{}_P & = & \left\{Clo^\sharp{}_P(\tau) \mid \tau \in types_P\right\} \\
Clo^\sharp{}_P(\tau) & = & \textbf{type}\ d_\tau\ := (C_{\ell_1}\ [\![\tau_1]\!]_P, \cdots, C_{\ell_n}\ [\![\tau_n]\!]_P, C_\tau\ \mathsf{Int}) \\
 & & \text{where, } \left\{e_1^{\ell_1}, \cdots, e_n^{\ell_n}\right\} = \left\{t/_{\cong,P} \mid t \in Lam_P \wedge type_P(t) = \tau\right\} \\
 & & \qquad \forall i \in [1,n].\ \tau_i = type_P(FV(e_i))
\end{array}
$$

$$
\begin{array}{rcl}
FVal^\sharp_P & = & \textbf{type}\ \mathsf{dcache}\ := (C_{f_1}\ [\![\tau_1]\!]_P, \cdots, C_{f_n}\ [\![\tau_n]\!]_P), \\
 & & \text{where, } \forall i \in [1,n].\ f_i \text{ has @memoize annotation in } P, \\
 & & \qquad \forall i \in [1,n].\ \tau_i = type_P(param_P(f_i))
\end{array}
$$

Figure 4.5 – Representation of closure and cache keys

$$
\begin{array}{rcl}
[\![\tau]\!]_P & = & \tau \qquad \text{if } \tau \in \{\mathsf{Unit}, \mathsf{Int}, \mathsf{Bool}\} \\
[\![\tau \Rightarrow \tau]\!]_P & = & d_{\tau \Rightarrow \tau} \\
[\![(\tau_1, \cdots, \tau_n)]\!]_P & = & ([\![\tau_1]\!]_P, \cdots, [\![\tau_n]\!]_P) \\
[\![\textbf{type } d := (C_1\ \tau_1, \cdots, C_n\ \tau_n)]\!]_P & = & \textbf{type } d\ := (C_1\ [\![\tau_1]\!]_P, \cdots, C_n\ [\![\tau_n]\!]_P) \\
Tdef^\sharp_P & = & \{[\![t]\!]_P \mid t \in Tdef_P\} \cup Clo^\sharp{}_P \cup \left\{FVal^\sharp_P\right\}
\end{array}
$$

Figure 4.6 – Translation of types in a program $P$ with a set of type declarations $Tdef_P$

denotes the set of types used in the program $P$.

I will first define a syntactic notion of compatibility between lambda terms in a program. Two lambdas $e_\lambda = \lambda x.f\ (x, y)$ and $e_\lambda{}' = \lambda x.f'\ (x, z)$ are compatible, denoted $e_\lambda \cong e_\lambda{}'$, iff they invoke the same targets i.e, $f = f'$. This relation is interesting because during any evaluation two closures could be structurally equivalent iff their lambdas are compatible i.e, $e_\lambda \cong e_\lambda{}'$ iff $\exists H, \sigma, \sigma'$ s.t. $(e_\lambda, \sigma) \underset{H}{\approx} (e_\lambda{}', \sigma')$. In the generated model it is ensured that the closures of lambdas that are compatible are represented using the same datatype. For each lambda $e_\lambda$ (whether or not it belongs to a program), a representative denoted $e_\lambda / _{\cong, P}$ of the equivalence class with respect to $\cong$ is defined. The representative is required to be another lambda term in the program $P$. It is undefined if $P$ does not have a compatible lambda. For each function type $\tau = A \Rightarrow B$ used in $P$, a datatype $d_\tau$ is added to the model as explained below. Every use of $\tau$ in the input program is replaced by the datatype $d_\tau$.

Let $\{e_{\lambda_i} \mid i \in [1, n]\}$ be the representatives (with respect to $\cong$) of the lambda terms in the program $P$ that are of type $\tau$, and let $\{\ell_i \mid i \in [1, n]\}$ be their labels. The datatype $d_\tau$ has $n+1$ constructors denoted $C_{l_i}$, $i \in [1, n]$ and $C_\tau$. That is, $d_\tau$ is of the form: $\textbf{type } d_\tau := (C_{\ell_1}\ [\![\tau_1]\!]_P, \cdots, C_{\ell_n}\ [\![\tau_n]\!]_P, C_\tau\ \mathsf{Int})$. The $i^{th}$ constructor $C_{\ell_i}$ represents the closure of the $i^{th}$ lambda term: $e_{\lambda_i}$. The parameter of the constructor represents $FV(e_{\lambda_i})$. The type $[\![\tau_i]\!]_P$ is obtained by recursively replacing the function types by their corresponding closure datatypes in $type_P(FV(e_{\lambda_i}))$, which is captured by the type translation shown in Figure 4.6. The $(n+1)^{th}$ constructor $C_\tau$ of $d_\tau$ is a stub for a closure created outside the program under analysis and serves to handle an error case (explained shortly).

In the running example shown in Fig. 4.4, the datatype tStream defined at line 1 represents the closures of lambdas of type Unit ⇒ Stream. The constructor Take of tStream represents the closure of λa.take (n1, t) created at line 17. As shown at line 24, the lambda is replaced by an instance of Take in the model program. The constructor Other represents the stub closure $c_\tau$.

**Cache encoding** The expressions of the input program are instrumented to explicitly track the changes to the cache as the program undergoes evaluation. The instrumentation tracks only the keys of the cache, which are elements of *FVal*, as it fully specifies the state of the cache at every instance. (This is because of the referential transparency of the source expressions established in section 4.2.) The keys of the cache (*FVal*) are represented in the model program by a datatype Dcache defined in Figure 4.5. Below I describe this representation below.

For every memoized function in the program, $FVal^{\sharp}_P$ has a constructor $C_f$ with a field that holds the parameter of the function $f$. Formally, **type** dcache := $(C_{f_1}\ [\![\tau_1]\!]_P, \cdots, C_{f_n}\ [\![\tau_n]\!]_P)$, where $f_i$'s are functions in the program annotated with @memoize, and $[\![\tau_i]\!]_P$ is the type of the parameter of $f_i$ that is translated by replacing all closures by their representatives (shown in Figure 4.6). In the running example shown in Fig. 4.3, the datatype Dcache with one constructor: (Tail Stream) corresponds to this datatype.

**Translation of expressions** Fig. 4.7 formally defines the transformation $[\![\cdot]\!]_P$ that maps expressions of an input program $P$ to a model program $P^{\sharp}$. For every expression $e$, $[\![e]\!]_P$ takes a state expression $st$ representing the keys of the cache before the evaluation of $e$ and returns the translated expression denoted $e^{\sharp}$. The expression $e^{\sharp}$ is a triple where the first element $e^{\sharp}.1$ corresponds to the value of $e$, the second element $e^{\sharp}.2$ corresponds to the keys of the cache after evaluation of $e$, and the last element $e^{\sharp}.3$ corresponds to the resource usage of $e$. The translation is explained in the sequel.

**Cache-state Propagation** The propagation of cache state proceeds top down in a store-passing style following the control flow of the program. To every function definition in the model, a fresh parameter $st$ (of type Set[dcache]) is added. It represents the state of the cache at the beginning of the function (see translation of function definitions). This parameter is propagated through the bodies of the function recording all the calls that are memoized along the way. (E.g. see the translation of **let** expression.) The state parameter is used at two places: (a) by calls to memoized functions, and (b) by the cached construct to check whether the call given as argument is memoized.

Consider the translation of a call to a memoized function shown in Fig. 4.7. It uses the input state parameter $st$ to check whether the call would be a cache hit by testing if $st$ contains $(C_f\ x)$ which represents the key $(f\ x)$. The resource usage in the cache hit case is given by $c_{hit}$, whereas in the miss case it is a combination of $c_{miss}$, the cost of the call $c_{call}$ and the resource usage of the callee $w.3$. Finally, $(C_f\ x)$ is added to the output state to record that the call is memoized. Observe that the call always happens in the model regardless of whether or not it was memoized before. This encodes the referential transparency of memoized functions i.e, the value of the call that is a hit in the cache is equivalent to the result of the invoked function, and avoids having to specify an invariant on the cache. (Recall that we are not interested in the resource usage of the model.)

During the translation of contracts, the precondition is translated using the initial state $st$ and the postcondition using the state resulting after the translation of the body $res.2$, as in the operational semantics (Figure 4.1). Any changes to the state caused by the contracts are discarded at the end of the contracts. The uses of res in the postcondition is replaced by $res.1$, and the uses of a resource $R$ by $res.3$. The uses of inSt and outSt, representing the cache state before and after the triple, is replaced by $st$ and $res.3$ respectively. Note that the construct

***Expression Translation***

$$
\begin{aligned}
[\![x]\!]_P\ st &= (x, st, c_{var}) \\
[\![pr\ x]\!]_P\ st &= (pr\ x, st, c_{pr}) \quad \text{if } pr \in Prim \\
[\![x\ \mathsf{eq}\ y]\!]_P\ st &= (x\ \mathsf{eq}\ y, st, c_{eq}) \\
[\![C\ \bar{x}]\!]_P\ st &= (C\ \bar{x}, st, c_{cons}) \quad \text{if } C \in Cids \\
[\![\mathbf{let}\ x := e_1\ \mathsf{in}\ e_2]\!]_P\ st &= \mathbf{let}\ u := [\![e_1]\!]_P\ st\ \mathsf{in} \\
&\quad \mathbf{let}\ w := [\![e_2[u.1/x]]\!]_P\ u.2\ \mathsf{in} \\
&\quad (w.1, w.2, c_{let} \oplus u.3 \oplus w.3) \\
[\![x\ \mathbf{match}\ \{\ C_i\ \bar{x_i} \Rightarrow e_i\ \}_{i=1}^n]\!]_P\ st &= x\ \mathbf{match}\ \{\ \big(C_i\ \bar{x_i} \Rightarrow \mathbf{let}\ u := [\![e_i]\!]_P\ st\ \mathsf{in} \\
&\qquad\qquad (u.1, u.2, c_{match(i)} \oplus u.3)\big)_{i=1}^n\}
\end{aligned}
$$

***Call and Lambda Translation***

$$
\begin{aligned}
[\![f\ x]\!]_P\ st &= \mathbf{let}\ w := f^\sharp\ (x, st)\ \mathsf{in}\ (w.1, w.2, c_{call} \oplus w.3) \quad \text{if } f \notin Mem_P \\
[\![f\ x]\!]_P\ st &= \mathbf{let}\ w := f^\sharp\ (x, st)\ \mathsf{in} \quad \text{if } f \in Mem_P \\
&\quad \mathbf{let}\ x_{cost} = \mathbf{if}\ (C_f\ x) \in st\ c_{hit}\ \mathbf{else}\ c_{miss} \oplus c_{call} \oplus w.3\ \mathsf{in} \\
&\quad (w.1, w.2 \cup \{(C_f\ x)\}, x_{cost}) \\
[\![e_\lambda]\!]_P\ st &= (C_\ell\ FV(e_\lambda), st, c_\lambda) \quad \text{if } e_\lambda/_{\cong,P} \text{ has label } \ell \\
[\![(x\ y)^\ell]\!]_P\ st &= \mathbf{let}\ w := App_\ell\ (x, y, st)\ \mathsf{in}\ (w.1, w.2, c_{app} \oplus w.3)
\end{aligned}
$$

***Specification Construct Translation***

$$
\begin{aligned}
[\![\mathsf{cached}(f\ x)]\!]_P\ st &= ((C_f\ x) \in st, st, 0) \\
[\![\mathsf{in}(e, x)]\!]_P\ st &= [\![e]\!]_P\ x \\
[\![e^\star]\!]_P\ st &= ([\![e]\!]_P\ st).1 \\
[\![x\ \mathsf{fmatch}\ \{\ \lambda x_i.f_i\ (x_i, y_i) \Rightarrow e_i\}_{i=1}^n]\!]_P\ st &= x\ \mathbf{match}\{C_{\ell_i}\ y_i \Rightarrow [\![e_i]\!]_P\ st\}_{i=1}^n \\
&\qquad \text{where } \ell_i \text{ is the label of } \lambda x_i.f_i\ (x_i, y_i)/_{\cong,P}
\end{aligned}
$$

***Contract Translation***

$$
\begin{aligned}
[\![\{p\}\ e\ \{s\}]\!]_P\ st &= \{([\![p]\!]_P\ st).1\} \quad \text{if } R \in \{\mathsf{steps}, \mathsf{alloc}\} \\
&\quad [\![e]\!]_P\ st \\
&\quad \{\mathbf{let}\ y := [\![s[\mathsf{res}.1/\mathsf{res}][st/\mathsf{inSt}][\mathsf{res}.2/\mathsf{outSt}][\mathsf{res}.3/R]]\!]_P\ \mathsf{res}.2 \\
&\qquad \mathsf{in}\ y.1\}
\end{aligned}
$$

***Function Definition Translation***

$$
[\![\mathbf{def}\ f\ x := e]\!]_P = \mathbf{def}\ f^\sharp\ (x, st) := [\![e]\!]_P\ st
$$

***Dispatch Functions***

For every indirect call $(x\ y)^\ell$ in $P$ where $type_P(x) = \tau$,

$$
\begin{aligned}
\mathbf{def}\ App_\ell\ (cl, w, st)\ := cl\ \mathbf{match}\{\ & C_{\ell_1}\ y_1 \Rightarrow [\![e'_1]\!]_P\ st;\ \cdots \\
& C_{\ell_n}\ y_n \Rightarrow [\![e'_n]\!]_P\ st; \\
& C_\tau\ y \Rightarrow\ \mathsf{error} \\
\}&
\end{aligned}
$$

where, $\forall i \in [1, n]$. $C_{\ell_i}$ are constructors of $d_\tau$ and $e'_i = e_i[y_i/z_i][w/a_i]$ if $(\lambda a_i.e_i)^{\ell_i} \in Lam_P$

Figure 4.7 – Resource and cache-state instrumentation of source expressions

fmatch is translated into a usual **match** construct on the datatypes representing closures.

Fig. 4.4 illustrates the result of propagating the state through the body of take function as outputted by our tool. Our tool eliminates propagation through expressions and functions that are statically inferred as not affecting the state. For instance, concrUntil does not return a state as it was statically determined to not have any effect on the state. (It is a specification function that only queries the cache.) Observe that after the call tail$^\sharp$ (s, st) at line 21, an instance of (Tail s) is added to the output state to record that the call is memoized, and that the computation of steps at line 23 depends on whether or not (Tail s) belongs to the input state st.

**Resource Instrumentation** The expressions of the language are instrumented to track their resource usage in a manner similar to instrumentation of first-order programs discussed in section 3.1. It proceeds bottom-up, first instrumenting the sub-expressions of an expression $e$, and then using the resource usages of the sub-expressions to instrument $e$. However, a call to a memoized function is handled differently. Given a memoized function call $f\ x$, the state reaching the call expression (propagated top down) is looked up to determine whether the key corresponding to call: $C_f\ x$ belongs to the state. If so, it indicates a cache hit and hence the cost of the operation is given by $c_{hit}$. Otherwise, it miss a cache miss and hence the cost of evaluating the function is factored into the resource calculation.

The example model program shown in Fig. 4.4 is obtained after a few straightforward static simplifications performed by our tool. For instance, the constants such as 10 and 5 that appear in the resource expressions are the result of adding up all the constants in the instrumented expressions along the same branch (or match case) in the program.

**Defunctionalization** The model transformation translates an indirect call: $x\ y$ to a guarded disjunction of direct calls through a process known as defunctionalization [Reynolds, 1998]. Every indirect call $x\ y$ with label $\ell$ is replaced by a call to a dispatch function $App_\ell$ constructed as follows. The parameters of the function are (a) a closure $cl$ of type $d_\tau$ where $\tau = type_P(x)$ i.e, $\tau$ is the type of the closure that is invoked, (b) the argument of the call $w$, and (c) a state parameter $st$ denoting the state of the cache at the entry of the function. The dispatch function matches the closure $cl$ to each possible constructor and in each case $C_{\ell_i}$, where $\ell_i$ is the label of the lambda $\lambda a_i.e_i$ represented by the constructor, invokes the expression $[\![e'_i]\!]_P\ st$ where $e'_i$ is the result of replacing in $e_i$ the parameter of the lambda $a_i$ with $w$ and the free variable of the lambda with the argument of $C_{\ell_i}$.

If the closure matches $C_\tau$, the model halts with an error as this case corresponds to the scenario where a function not defined within the program $P$ is applied to an argument. Such a function, being arbitrary, may either not terminate or can have a precondition that is violated by the arguments it is applied to. (In fact, the precondition could even be *false*.) The model program soundly flags this case as an error. We eliminate this case if we can statically infer (based

on type encapsulation) that the targets of the closures are strictly within the program under analysis. Observe that in example model program shown in Figure. 4.4 the call to tfun inside the function tail is translated to a call to the dispatch function app. The case Other is omitted in app as we assume that the call is encapsulated.

Importantly, note that even though the set of possible cases in the function $App_l$ could be large, many of those cases that are not feasible at runtime are not explored by the verification algorithm as it uses *targeted unfolding* described in section 3.7. Recall that in this unfolding strategy only the functions seen along the satisfiable disjuncts in a verification condition that were explored by the inference algorithm (solveUNSAT) will be unfolded. Moreover, this overhead can be further reduced through a static control-flow analysis [Midtgaard, 2012] that narrows down the targets of the call, and through approaches such as DAG inlining [Lal and Qadeer, 2015]

I would like to remark that in contrast to related works [Avanzini et al., 2015], which use defunctionalization as a means to estimate the resource usage of input programs, here it is as a means to precisely encode the control flow of the source program, which is obfuscated in the presence of first-class functions. Also note that only values of the expressions of model programs are of interest (and not resources).

## 4.5 Soundness and Completeness of the Model Programs

In this section, I establish the soundness of the model program for the contract verification problem. That is, I establish that if the contract verification succeeds for the functions of the model program then it will also succeed for the source program. I also establish a form of completeness, which states that if the contract verification fails for a function in the model program then there exists an environment in $Env^c$ such that the contract of the corresponding function in the source program also fails. However, the environment need not be valid i.e, it need not belong $Env^c_{e,P}$. In other words, the completeness does not imply that the contract of the source program will fail as per the definitions of section 2.5. However, it does imply that the contracts of the functions in $Env^c$ are not strong enough to rule out certain environments that could fail the contracts, even though a global program invariant prevents it from arising at runtime. This completeness property despite being weak is nevertheless interesting if one is interested in modular verification. In the sequel I formally detail the proof using simulation relations and also establish several auxiliary lemmas that are necessary for the proof.

To start with I define a relation $\underset{H,H^\sharp,P}{\sim}$ between the semantic domains of the source and the model language as follows: (subscripts omitted below for clarity)

$$\begin{aligned}
\forall a \in \mathbb{Z} \cup Bool.\ & a \sim a \\
\forall c \in Cids, \{\bar{a}, \bar{b}\} \subseteq Val^n.\ & c\ \bar{a} \sim c\ \bar{b}\ \textit{iff}\ \forall i \in [1,n].a_i \sim b_i \\
\forall (e_\lambda, \sigma) \in Clo, v \in Val, \ell \in labels_P.\ & (e_\lambda, \sigma) \sim C_\ell\ v\ \textit{iff}\ \sigma(FV(e_\lambda)) \sim v \\
& \quad \wedge (e_\lambda /_{\cong,P} \text{ is defined and has label } \ell) \\
\forall (e_\lambda, \sigma) \in Clo, v \in Val.\ & (e_\lambda, \sigma) \sim (C_{type_P(e_\lambda)}\ hash((e_\lambda, \sigma)))\ \textit{iff}\ e_\lambda /_{\cong,P} \text{ is undefined} \\
\forall f \in Fids \text{ defined in } P, \{a,b\} \subseteq Val.\ & f\ a \sim C_f\ b\ \textit{iff}\ a \sim b \\
\forall \{a,b\} \subseteq Adr.\ & a \sim b\ \textit{iff}\ H(a) \sim H^\sharp(a) \\
\forall C \in Cache, S \in Set.\ & C \sim S\ \textit{iff}\ |dom_P(C)| = |dom(S)| \wedge \left(\forall x \in dom_P(C).\exists y \in S.x \sim y\right) \\
\forall \{\sigma, \sigma^\sharp\} \subseteq Store.\ & \sigma \sim \sigma^\sharp\ \textit{iff}\ dom(\sigma) \cup \{st\} = dom(\sigma^\sharp) \wedge \forall x \in dom(\sigma).\ \sigma(x) \sim \sigma^\sharp(x)
\end{aligned}$$

The above definition uses the following helper functions. Define a function $hash_\Gamma : Lam \mapsto \mathbb{N}$ that maps two closures in *Clo* (bound in the heap of the environment $\Gamma$) to the same natural number iff they are structurally equivalent. That is,

$$\forall \{c_1, c_2\} \subseteq Clo.\ c_1 \underset{H}{\approx} c_2 \iff hash(c_1) = hash(c_2)$$

Define $dom_P(C)$ as the set of all keys in the cache $C$ that refer to functions in the program $P$. That is, $dom_P(C) = \left\{(f\ u) \in dom(C) \mid f \text{ is defined in } P\right\}$

The relation $\sim$ formally captures that a cache is simulated by a set of instances of dcache, and that a closure of type $\tau$ is simulated by an instance of the datatype $d_\tau$ if the lambda of the closure has a representative in the program $P$ with respect to $\cong$. Otherwise, it is simulated by an instance of $C_\tau$. The instance is chosen in such way that two structurally equivalent closures map to equivalent $C_\tau$ instances. The relation $\sim$ behaves much like a simulation relation between the evaluation of the source and the model programs. However, not all evaluations can be simulated by the model program as will be highlighted later. I now define a simulation relation $\sim_P$ that relates an environment $\Gamma : (C, H, \sigma, P) \in Env^c$ with a model environment $\Gamma^\sharp : (H^\sharp, \sigma^\sharp, P^\sharp) \in Env$, where $P^\sharp$ is the model program corresponding to $P$. But, somewhat unique to our setting, $\Gamma$ is simulated by a pair $(\Gamma^\sharp, S)$ where $S \in 2^{Val}$.

$$\Gamma \sim_P (\Gamma^\sharp, S)\ \ \textit{iff}\ \ C \underset{H,H^\sharp,P}{\sim} S \wedge \sigma \underset{H,H^\sharp,P}{\sim} \sigma^\sharp \wedge P^\sharp = [\![P]\!]$$

Before I establish some auxiliary properties of the relation $\sim$ necessary for the proof. In all the formalism that follow if $\Gamma_i$ (or $\Gamma^i$) is an environment then its individual components $C$ are denoted using $C_i$ (or $C^i$), respectively.

**Lemma 20.** *Let $H_1, H_2$ be two heaps and let $P$ be a program. The relation $\underset{H,H^\sharp,P}{\sim}$ is monotonic with respect to $\sqsubseteq$ on the heaps. That is, if $x \underset{H,H^\sharp,P}{\sim} y$, $H \sqsubseteq H_o$ and $H^\sharp \sqsubseteq H_o^\sharp$ then, $x \underset{H_o,H_o^\sharp,P}{\sim} y$.*

*Proof.* Follow by straightforward structural induction over the definition of $\sim$. □

**Lemma 21.** *Let $P$ be a program. Let $u, v$ be two values and $H, H^\sharp$ be two heaps. The simulation*

*relation* $\underset{H,H^\sharp,P}{\sim}$ *is preserved by the structural equality relations* $\underset{H}{\approx}$ *and* $\underset{H^\sharp}{\approx}$ *and vice versa. That is, if* $u \underset{H,H^\sharp,P}{\sim} v$ *then* $(u \underset{H,H^\sharp,P}{\sim} v' \iff v \underset{H^\sharp}{\approx} v')$ *and* $(u' \underset{H,H^\sharp,P}{\sim} v \iff u \underset{H}{\approx} u')$.

*Proof.* I omit the subscripts of $\sim$ and $\approx$ in the rest of the proof. I show the proof for one part: if $u \sim v$ then $(u' \sim v \Leftrightarrow u \approx u')$. The proof of the other part is symmetric.

Say $u \approx u'$. Now $u' \sim v$ is shown using structural induction on $\approx$. If $u$ is an integer or boolean, the claim follows immediately as $u' = u$. Say $u$ is an address of $(C\ \bar{w})$ i.e, $H(u) = (C\ \bar{w})$. By the definition of $\approx$ and $\sim$, $u'$ and $v$ are also addresses of $(C\ \bar{w'})$ and $(C\ \bar{z})$ such that for all $i \in [1, |u|]$, $w_i \approx w_i'$ and $w_i \sim z_i$, respectively. By inductive hypothesis, $w_i' \sim z_i$. Hence, the claim.

Now say $u$ is an address of a closure $(e_\lambda, \sigma')$. If ${e_\lambda}/_{\cong,P}$ is defined and has label $\ell$, $v = (C_\ell\ t)$ and $\sigma'(FV(e_\lambda)) \sim t$. Since $u \approx u'$, $u' = ({e_\lambda}', \sigma'')$, ${e_\lambda}'/_{\cong,P} = {e_\lambda}/_{\cong,P}$ (by the definition of the $\cong$ relation) and $\sigma''(FV({e_\lambda}')) \approx \sigma'(FV(e_\lambda))$. By induction hypothesis, $\sigma''(FV(e_\lambda)) \sim t$. Hence, $u' \sim v$.

If ${e_\lambda}/_{\cong,P}$ is not defined, $v = (C_{type_P(e_\lambda)}\ hash((e_\lambda, \sigma)))$. Since $u \approx u'$, $u' = ({e_\lambda}', \_)$ and ${e_\lambda}'/_{\cong,P}$ is not defined. By the definition of *hash*, $hash(u) = hash(u')$. Hence, $u' \sim v$.

Say $u' \sim v$. We now show that $u \approx u'$ using structural induction on $\sim$. If $u$ is an integer or boolean the claim immediately follows as in that case $u = v = v' \in \mathbb{N} \cup Bool$. Say $u$ is an address of $(C\ \bar{w})$ i.e, $H(u) = (C\ \bar{w})$. By the definition of $\approx$ and $\sim$, $u'$ and $v$ are also addresses of $(C\ \bar{w'})$ and $(C\ \bar{z})$ such that for all $i \in [1, |u|]$, $w_i \sim z_i$ and $w_i' \sim z_i$, respectively. By inductive hypothesis, $w_i \approx w_i'$. Hence, the claim. The case where $u$ is an address of a closure can be similarly proven. □

**Lemma 22.** *Let $P$ be a program. Let $\Gamma \in Env^c$ , $\Gamma^\sharp \in Env^\sharp$ and $S \in 2^{Val}$ be such that $\Gamma \sim (\Gamma^\sharp, S)$. $\left(\forall x \in dom_P(C).\exists y \in S.\ x \sim y\right)$ and $\left(\forall y \in S.\exists x \in dom_P(C).\ x \sim y\right)$.*

*Proof.* The first part of the claim follows by the definition of $\sim$. That is, $\forall x \in dom_P(C).\exists y \in S.\ x \sim y$. By skolemization, the above implies that there exists a function $g : dom_P(C) \to S$. We know that $|dom_P(C)| = |dom(S)|$ by the definition of $\sim$. If $g$ is injective, it should also be bijective and hence the claim holds. If $g$ is non-injective, there exists an element $s \in S$ such that $x_1 \sim s$ and $x_2 \sim s$ for some $\{x_1, x_2\} \subseteq dom_P(C) \subseteq dom(C)$ and $x_1 \neq x_2$. By Lemma 21, $x_1 \underset{H}{\approx} x_2$. But by the domain invariants, every key in the cache is unique with respect to structural equality. Therefore, this case is not possible. □

**Lemma 23.** *Let $P$ be a program. Let $\Gamma \in Env^c$ , $\Gamma^\sharp \in Env^\sharp$ and $S \in 2^{Val}$ be such that $\Gamma \sim (\Gamma^\sharp, S)$. Let $x \in dom(\sigma)$ and $f \in Fids$ be a function defined in $P$. $\left(\exists u.(f\ u) \in dom_P(C) \wedge u \underset{H}{\approx} \sigma(x)\right)$ iff $\left(\exists u'.(C_f\ u') \in S \wedge u' \underset{H^\sharp}{\approx} \sigma^\sharp(x)\right)$*

*Proof.* Consider the only if direction. Say $(f\ u) \in dom_P(C)$ and $u \underset{H}{\approx} \sigma(x)$. By the definition of $\sim$, $\exists y \in S.(f\ u) \sim y$. In other words, $\exists u'.(C_f\ u') \in S \wedge u \sim u'$. We are given that $\sigma(x) \sim \sigma^\sharp(x)$ and

$\sigma(x) \underset{H}{\approx} u$. By Lemma 21, $u \sim \sigma^\sharp(x)$. This together with the fact that $u \sim u'$ imply that $u' \underset{H}{\approx} \sigma^\sharp(x)$. Hence, the claim. The other direction is symmetric (by Lemma 22). □

### 4.5.1 Correctness of Model Transformation

The following lemma establishes that if $\Gamma \sim (\Gamma^\sharp, S)$, evaluating an expression $e$ under $\Gamma$ results in fewer crashes than evaluating the translation of $e$ under $\Gamma^\sharp$. That is, $\Gamma$ progresses as long as $\Gamma^\sharp$ progresses on the translation of $e$.

**Lemma 24.** *Let $P$ be a program. Let st be an expression of the model language. Let $\Gamma \in Env^c$ and $\Gamma^\sharp \in Env$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim (\Gamma^\sharp, S)$. Let e be any expression. If $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$ then $\exists \Gamma_o \in Env, v \in Val, p \in \mathbb{N}$ such that $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ and*
• $\Gamma_o \sim (\Gamma^\sharp_o, u.2)$ • $v \underset{H_o, H^\sharp_o, P}{\sim} u.1$ • $p = u.3$

*Proof.* Using structural induction on the evaluation $\Gamma^\sharp \vdash (\llbracket e \rrbracket_P\ st) \Downarrow u, \Gamma^\sharp_o$. However, instead of considering the semantic rules one by one, I consider here the semantic rules for the model expressions that correspond to the possible outermost operation in $e$. (Note that $e$ is a source expression and the induction is performed on the semantic rules of the model expressions.) In the rest of the proof I omit the subscript $P$ of $\sim$ relations that refers to the program. Let $e' = \llbracket e \rrbracket_P\ st$.

Say the expression $e$ belongs to one of the following cases: a constant $c$, a variable $x$, $pr\ x$, $x\ \mathsf{eq}\ y$, $cons\ \bar{x}$, $\lambda x.f\ (x, y)$, $\mathsf{cached}(f\ x)$. The free variables of $e'$ and $e$ are identical, and by the definition of $\sim$, $FV(e) \subseteq dom(\sigma^\sharp) = dom(\sigma) \cup \{st\}$. Hence, there is a value defined for all free variables in $\sigma$. Since $\Gamma$ satisfies all the domain invariants, the antecedent of every base case rule is defined. Therefore, $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ for some $v$, $p$ and $\Gamma_o$.

In all these cases, the cost of the operation $c_{op}$ is a constant as per the semantics, and is exactly same as $u.3$ as per the translation $\llbracket \cdot \rrbracket_P$ . Therefore, $p = u.3$ holds trivially.

Consider now the claim: $\Gamma_o \sim (\Gamma^\sharp_o, u.2)$. Recall that the relation $\sim$ is monotonic with respect to the ordering $\sqsubseteq$ between the heaps. In all these base cases, the cache and store components of the input and the output environments $\Gamma$ and $\Gamma_o$ are identical. The heaps of $\Gamma$ and $\Gamma^\sharp$ are contained in the heaps of $\Gamma_o$ and $\Gamma^\sharp_o$. Moreover, as per the translation, $st$ and $u.2$ are also identical in the base cases. Therefore, by Lemmas 7 and 20, $\Gamma \sim (\Gamma^\sharp, S)$ directly implies $\Gamma_o \sim (\Gamma^\sharp_o, u.2)$.

Consider now the claim: $v \underset{H_o, H^\sharp_o}{\sim} u.1$. In the case of a constant it is easy to see that the values returned by $e$ are identical primitive values (in $\mathbb{N} \cup Bool$) in both evaluations under $\Gamma$ and $\Gamma^\sharp$. In the case of $pr\ x$, the arguments of the operations are integer or boolean. By the definition of $\sim$, the arguments are equal in both $\sigma$ and $\sigma^\sharp$. Hence the output of PRIM is also equal under both environments. (We allow only deterministic primitive operations.) Therefore, $v \underset{H_o, H^\sharp_o}{\sim} u.1$

in both cases.

Consider the case of $e$ being a variable. Say $\sigma(x) = a$ and $\sigma^\sharp(x) = a'$. It is given that $a \underset{H,H^\sharp}{\sim} a'$. By definition, $v = a$ and $u.1 = a'$. Hence, the claim holds by Lemmas 7 and 20. In the case of CONS, $(a \mapsto cons\, \hat{\sigma}(\bar{x}))$ is added to $H$ and $(a' \mapsto cons\, \hat{\sigma^\sharp}(\bar{x}))$ is added to $H^\sharp$, for some fresh $a$ and $a'$ that are not bounded in $H$ and $H^\sharp$, respectively. It is given that $\sigma \underset{H,H^\sharp}{\sim} \sigma^\sharp$. Therefore, $a \underset{H,H^\sharp}{\sim} a'$ by the definition of $\sim$, which by Lemma 20 implies $a \underset{H_o,H_o^\sharp}{\sim} a'$. Therefore, $v \underset{H_o,H_o^\sharp}{\sim} u.1$. The LAMBDA case can be similarly proved.

Consider now the case where $e$ is cached$(f\ x)$ (for some $f$ and $x$). We are given that $\sigma(x) \underset{H,H^\sharp}{\sim} \sigma^\sharp(x)$. By the definition of $\underset{H,H^\sharp}{\sim}$, $(f\ \sigma(x)) \underset{H,H^\sharp}{\sim} (C_f\ \sigma^\sharp(x))$, provided $f$ is define in the program $P$, which holds because we require that every named function used in the program are defined in the program. By Lemma 23, $\exists u'.(C_f\ u') \in S \wedge u' \underset{H^\sharp}{\approx} \sigma^\sharp(x)$, where $\Gamma^\sharp \vdash st \Downarrow S$, if and only if $\exists u.(f\ u) \in dom(C) \wedge u \underset{H}{\approx} \sigma(x)$. By the semantics of set inclusion shown in Fig. 4.2 and $\in_H$, $(C_f\ x) \in st$ evaluates to true under $\Gamma^\sharp$ iff $C(f\ x)$ evaluates to true under $\Gamma$.

Consider now the case where $e$ is of the form $x$ eq $y$. This evaluates to true under $\Gamma$ iff $\sigma(x) \underset{H}{\approx} \sigma(y)$. It is given that $\sigma(x) \underset{H,H^\sharp}{\sim} \sigma^\sharp(x)$ and $\sigma(y) \underset{H,H^\sharp}{\sim} \sigma^\sharp(y)$. By Lemma 21, if $\sigma(x) \underset{H}{\approx} \sigma(y)$ is true then $\sigma(y) \underset{H,H^\sharp}{\sim} \sigma^\sharp(x)$, which in turn by the same lemma implies that $\sigma^\sharp(x) \underset{H^\sharp}{\approx} \sigma^\sharp(y)$. Similarly, if $\sigma(x) \underset{H}{\approx} \sigma(y)$ is false then by Lemma 21, $\neg(\sigma(y) \underset{H,H^\sharp}{\sim} \sigma^\sharp(x))$ which in turn implies that $\neg(\sigma^\sharp(x) \underset{H^\sharp}{\approx} \sigma^\sharp(y))$. Hence, the claim.

Now consider the case where $e$ is a direct call to a memoized function. There are two cases to consider based on whether or not the evaluation of $[\![e]\!]_P\ st$ goes through the cache hit branch of the if-condition (see Figure 4.7). Say we have a cache hit. In this case $p = u.3 = c_{hit}$. Also, $\Gamma_o \sim (\Gamma_o^\sharp, u.2)$, since the output caches, state expressions and stores are identical to the input in both evaluations $\Gamma$ and $\Gamma^\sharp$, and the output heaps are only larger. Consider now the claim: $v \underset{H_o,H_o^\sharp}{\sim} u.1$. Here, $v$ is the result of looking up $(f\ \sigma(x))$ in the cache $C$, whereas $u.1$ is the result of the evaluation $(f\ \sigma^\sharp(x))$ under $\Gamma^\sharp$. By inductive hypothesis, $\Gamma^\sharp \vdash (f\ \sigma^\sharp(x)) \Downarrow u, \Gamma_o^\sharp$ implies that $\Gamma \vdash (f\ \sigma(x)) \Downarrow w, \Gamma'$ and $w \underset{H',H_o^\sharp}{\sim} u.1$. By the property *CacheCorr*, $v = C((f\ \sigma(x)) \underset{H'}{\approx} w$. Therefore, $v \underset{H',H_o^\sharp}{\sim} u.1$ (by Lemma 21). Since $v \in dom(H)$ and $H \sqsubseteq H'$, $v \underset{H,H_o^\sharp}{\sim} u.1$. But $H$ and $H_o$ are identical for the MEMOCALLHIT rule. Hence, $v \underset{H_o,H_o^\sharp}{\sim} u.1$.

Now say we have a cache miss. Consider now the claim: $v \underset{H,H^\sharp}{\sim} u.1$. Here, $v$ and $u.1$ are the result of the evaluation $(f\ \sigma(x))$ and $(f\ \sigma^\sharp(x))$ under $\Gamma$ and $\Gamma^\sharp$, respectively. Thus, by inductive hypothesis, $v \underset{H_o,H_o^\sharp}{\sim} u.1$. Similarly, the fact that the resource usage of both evaluations are identical follow from the hypothesis. Consider now $\Gamma_o \sim (\Gamma_o^\sharp, u.2)$. Clearly, $\sigma_o \sim \sigma_o^\sharp$. However, $C_o^\sharp$ is added a new entry $C_f\ \sigma^\sharp(x)$ (by the semantics of set union). However, $C_o$ is also added a

new entry $(f\ \sigma(x)) \mapsto v$. Since $(f\ \sigma(x)) \sim C_f\ \sigma^\sharp(x)$ by definition, the claim that $\Gamma_o \sim (\Gamma_o^\sharp, u.2)$ holds.

If the expression $e$ is one of let expression, match expression, concrete call pr contract the claim follows by inductive hypothesis. Consider now the case where $e$ is an indirect call i.e, $e = (x\ y)^\ell$. The translated expression $[\![e]\!]_P\ st$ invokes the function $App_\ell$ defined in Fig. 4.7. Let $\sigma(x) = (e_\lambda, \sigma')$.

Now say $e_\lambda /_{\cong,P}$ is not defined. By definition of $\sim$, $\sigma^\sharp(x) = (C_{type_P(e_\lambda)}\ hash((e_\lambda, \sigma')))$. Therefore $App_\ell$ with execute the error expression, and thus will crash. That is, $\neg\exists\Gamma_o^\sharp, u.\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma_o^\sharp$. Hence the claim trivially holds.

Now say $e_\lambda /_{\cong,P} = (\lambda x.f\ (x,z), \sigma')^{\ell'}$, where $dom(\sigma') = \{z\}$. By definition of $\cong$, $target(e_\lambda) = f$. By the definition of $\sim$, $\sigma^\sharp(x) = (C_{\ell'}\ t)$ where $\sigma'(z) \sim t$. By the definition of $App_\ell$ (Fig. 4.7) and the match construct, $\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma_o^\sharp$ reduces to $\Gamma^{\sharp\prime} \vdash ([\![f\ (y, y_i)]\!]_P\ st) \Downarrow u, \Gamma_o^\sharp$, where $\Gamma^{\sharp\prime} = (H^\sharp, \sigma^\sharp \uplus (y_i \mapsto t))$. Now consider $\Gamma' = (C, H, \sigma \uplus \sigma')$. Clearly, $\Gamma' \sim (\Gamma^{\sharp\prime}, \sigma^\sharp(st))$. Therefore, by induction hypothesis, $\Gamma' \vdash f\ (y,z) \Downarrow_p v, \Gamma_o$, $\Gamma_o \sim (\Gamma_o^\sharp, S)$, $p = u.3$ and $u \sim v$. (Note that the variables $y_i$ and $z$ can be renamed to a variable say $r \notin dom(\sigma)$ so that the calls are syntactically identical and the induction hypothesis can be applied.) By the definition of the rule IndirectCall, the above implies that $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ and hence the claim holds. □

The above lemma shows that if an expression $[\![e]\!]_P\ st$ evaluates to a value in the model program under an environment then $e$ evaluates to a similar value in source program under a similar environment. Below I show the converse. However, the converse holds only if the indirect calls encountered during the evaluation of the source expression have their definition in $P$. That is, they have to be encapsulated calls (section 2.7).

**Lemma 25.** *Let $P$ be a program. Let $st$ be an expression of the model language. Let $\Gamma \in Env^c$ and $\Gamma^\sharp \in Env$ be such that $\Gamma^\sharp \vdash st \Downarrow S$ and $\Gamma \sim (\Gamma^\sharp, S)$. Let $e$ be any expression such that if $\langle \Gamma, e\rangle \rightsquigarrow^* \langle \Gamma', x\ y\rangle$, $H'(\sigma'(x)) = ({e_\lambda}^\ell, \sigma'')$ and $\ell \in labels_P$. If $\Gamma \vdash e \Downarrow_p v, \Gamma_o$ then $\exists \Gamma_o^\sharp \in Env, u \in DVal$ such that $\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma_o^\sharp$ and*

- $\Gamma_o \sim (\Gamma_o^\sharp, u.2)$
- $v \underset{H_o, H_o^\sharp, P}{\sim} u.1$
- $p = u.3$

*Proof.* The proof of this lemma is very similar to the proof of Lemma 24, expect for a minor difference in the handling of the case where $e$ is an indirect call. It is given that every indirect call encountered during the evaluation of $e$ is an encapsulated call. As a result, when the expression $e$ is an indirect call $(x\ y)^{\ell'}$ (the rule IndirectCall), we are guaranteed that $\sigma(x) = ({e_\lambda}^\ell, \sigma')$ and $e_\lambda /_{\cong,P}$ is defined, since $e_\lambda$ itself belongs to the program $P$. Thus, the evaluation of $([\![e]\!]_P\ st)$ under $\Gamma^\sharp$ cannot go through the error case of $App_{\ell'}$ function, which implies that $\Gamma^\sharp \vdash ([\![e]\!]_P\ st) \Downarrow u, \Gamma_o^\sharp$ will be defined for the rule IndirectCall. It is easy to see that it will satisfy the properties of the claim as detailed in the proof of Lemma 24. □

**Relevant Model Environments**

Before I establish the sufficiency of the model programs for contract verification, I first define a subset of the valid environments of the model programs that are of interest here. It suffices to ensure validity of the contracts of the model programs with respect to this subset instead of considering all valid environments. This property is exploited by the assume-guarantee reasoning I will later present in section 4.6.

Let $P^\sharp$ be a model program corresponding to a source program $P$. Recall that the valid environments $\Gamma_{e,P^\sharp}$ of an expression $e$ are defined as the environments that reach $e$ during the evaluation of some closed program $P' \| P^\sharp$ for some client $P'$ of $P^\sharp$. However, since the cache in the model program is an expression of the model program, considering all possible clients of $P^\sharp$ may include clients that do not update the expression denoting the cache in accordance with the operational semantics of the input language. For instance, a client of the model program may update the set denoting the cache state non-monotonically. In other words, all environments $\Gamma_{e,P^\sharp}$ do not have a corresponding image in $Env^c_{e,P}$. Therefore, I define a subset of a the valid environments of the model program referred to as the *relevant model environments*, denoted ${Env^\sharp}_{e,P^\sharp}$, that has a one-to-one correspondence with the valid environments of the source program $Env^c_{e,P}$. Let **def** $f^\sharp\ (x, st)\ :=\ \tilde{e}'$ be a function definition in $P^\sharp$ that is a translation of the definition **def** $f\ x := \tilde{e}$ in $P$.

$$Env^\sharp_{\tilde{e}',P^\sharp} = \left\{\Gamma^\sharp \in Env \mid \exists \Gamma \in Env_{\tilde{e},P}.\Gamma \sim_P (\Gamma^\sharp, \sigma^\sharp(st))\right\}$$

The relevant environments ${Env^\sharp}_{e,P^\sharp}$ are defined only for the body of the functions in the model. The theorems that follow would need only these.

**Theorem 26** (***Model Soundness***)**.** *Let $P$ be a program and $P^\sharp$ the model program. Let $\tilde{e} = \{p\}\ e\ \{s\}$ and $\tilde{e}' = \{p'\}\ e'\ \{s'\}$. Let* **def** $f\ x := \tilde{e}$ *be a function definition in $P$, and let* **def** $f^\sharp\ (x, st) := \tilde{e}'$ *be the translation of $f$, where $st$ is the state parameter added by the translation.*

$$\forall \Gamma^\sharp \in {Env^\sharp}_{\tilde{e}',P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p' \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e}' \Downarrow u \implies \forall \Gamma \in Env_{\tilde{e},P}.\exists v.\ \Gamma \vdash p \Downarrow \mathit{false} \vee \Gamma \vdash \tilde{e} \Downarrow v$$

*Proof.* By Lemma 24 if $e'$ evaluates under $\Gamma^\sharp$ to a value then it does evaluated to a similar value under an environment $\Gamma \in Env^c$ that has is related to $\Gamma^\sharp$ by $\underset{P}{\sim}$. Therefore, the proof directly follows from the Lemma 24 if for every $\Gamma \in Env^c_{e,P}$ there exists a $\Gamma^\sharp \in {Env^\sharp}_{e,P}$ such that $\Gamma \underset{P}{\sim} (\Gamma^\sharp, \sigma^\sharp(st))$. Below I construct such an environment, which completes the proof. For every $\Gamma \in Env^c_{e,P}$ construct a $\Gamma^\sharp \in {Env^\sharp}_{e,P}$ as follows:
(a) $\sigma^\sharp = \sigma \cup (st \mapsto S)$, where $S = \{(C_f\ u) \mid (f\ u) \in dom_P(C)\}$
(b) $H^\sharp = \{(a, map(v)) \mid (a, v) \in H\}$,
where $map((e_\lambda, \sigma'))$ is $(C_l\ \sigma'(FV(e_\lambda)))$ if $e_\lambda /_{\cong,P}$ is defined and has label $\ell$, $map((e_\lambda, \sigma'))$ is $C_{type_{e_P}(e_\lambda)}\ hash((e_\lambda, \sigma'))$ if $e_\lambda /_{\cong,P}$ is not defined, and $map(v) = v$ otherwise. □

Note that the above theorem evidently implies that if the contracts of functions in the model programs hold then the contracts of functions in the source program also hold, since $Env^{\sharp}_{\tilde{e},P^{\sharp}} \subseteq Env_{\tilde{e},P^{\sharp}}$ by definition. Below I discuss the completeness property of the model program.

### 4.5.2 Completeness of Model Transformation

A tricky aspect here is that for an expression with contracts $\tilde{e}$ belonging to a program $P$ there may exist valid environments of $\Gamma \in Env^{c}_{\tilde{e},P}$ that binds addresses to lambdas not in the scope of the program $P$ under which the expression $\tilde{e}$ evaluates to a value. (This may happen e.g. if the expression invokes a closure passed as an argument to the function that contains the expression.) Such environments correspond to environments in the model program that bind the lambdas to the stub closure $C_{\tau}$. The model program crashes unconditionally if such a closure is invoked. Therefore, a general completeness guarantee does not hold . However if such lambdas external to $P$ are invoked by an expression belonging to $P$, the contracts of $\tilde{e}$ do not hold for all environments in $Env^{c}$ as there exists an environment in $Env^{c}$ that results in a contract violation in $\tilde{e}$. However, such an environment may not belong to $Env^{c}_{\tilde{e},P}$. Below I formalize this weaker completeness property. (Notice that the universal quantification over all environments of $Env^{c}$ that have a binding for the argument $x$ and not on the valid environments. This is why the completeness guarantee is weaker than ideal.)

**Theorem 27** (***Model Completeness***)**.** *Let $P$ be a program and $P^{\sharp}$ the model program. Let $\tilde{e} = \{p\}\ e\ \{s\}$ and $\tilde{e}' = \{p'\}\ e'\ \{s'\}$. Let* **def** $f\ x := \tilde{e}$ *be a function definition in $P$, and let* **def** $f^{\sharp}\ (x, st)\ := \tilde{e}'$ *be the translation of $f$, where $st$ is the state parameter added by the translation.*

$$\forall \Gamma : (C, H, \sigma, P) \in Env^{c}\ s.t.\ x \in dom(\sigma).\exists v.\ \Gamma \vdash p \Downarrow false \vee \Gamma \vdash \tilde{e} \Downarrow v \implies \\ \forall \Gamma^{\sharp} \in Env_{\tilde{e}',P^{\sharp}}.\exists u.\ \Gamma^{\sharp} \vdash p' \Downarrow false \vee \Gamma^{\sharp} \vdash \tilde{e}' \Downarrow u$$

*Proof.* There are two cases to consider here. In the first case say $P$ has only encapsulated calls. That is, for all $\Gamma \in Env^{c}$ , if $\langle \Gamma, e\rangle \rightsquigarrow^{*} \langle \Gamma', c\ a\rangle$ implies $H'(\sigma'(c)) = ({e_{\lambda}}^{\ell}, \sigma'') \wedge l \in labels_P$. In this case the claim holes by Lemma 25 since every evaluation in the model program is bisimulated by one evaluation in $\Gamma \in Env^{c}$.

Therefore, say $\langle \Gamma, e\rangle \rightsquigarrow^{n} \langle \Gamma', c\ a\rangle$, for some $n \in \mathbb{N}$, and $H'(\sigma'(x)) = ({e_{\lambda}}^{\ell}, \sigma'') \wedge l \notin labels_P$. That is, the evaluation of $e$ under $\Gamma$ invokes a lambda created outside the program. Without loss of generality assume that $c\ a$ is the first such call. That is, every call reached before $n$ steps is an encapsulated call. Now, it is easy to see that $H'(\sigma'(c)) = H(\sigma'(c))$. This is because if $\sigma'(c)$ is not bound in the input heap, it has to be bound subsequently. But we know that every expression that executes until encountering the call $c\ a$ belongs to the program $P$ since we assume that $c\ a$ is the first *call back* that executes code outside $P$. Thus, any closure created during the evaluation of $e$ until $c\ a$ belongs to $P$. Therefore, $\sigma'(c)$ should be bound in the input heap. Let $\sigma'(c) = a$ and $H(a) = (\lambda r.h\ (r, s), \sigma'')$.

Now, consider a new environment $\Gamma_{err} \in Env^{c}$ defined as follows: $\Gamma_{err} = (C, H[a \mapsto map(v)], \sigma, P \cup$

{**def** $g\ t = \{false\}\ h\ t\ \{true\}\}$), where $map((\lambda r.h\ (r,s),\sigma'')) = (\lambda r.g\ (r,s),\sigma'')$, for some $r$, $s$ and $\sigma''$, and $map(v) = v$ otherwise. That is, the new environment wraps the body of the lambdas *compatible* with $H(a)$ by a contract whose precondition is *false*. By the totality of $\sim$, $\exists \Gamma^\sharp \in Env_{e,Psharp}$ such that $\Gamma \sim (\Gamma^\sharp, \sigma^\sharp(st))$. Note that firstly (a) $\lambda r.h\ (r,s)/_{\cong,P}$ will not be defined as it is external to the program $P$. Consider now the following definition of the hash function for the newly introduced lambdas: define $hash((\lambda r.g\ (r,s),\sigma''))$ as equal to $hash((\lambda r.h\ (r,s),\sigma''))$. Clearly, this *hash* function preserves structural equality, i.e, $\forall\{e_\lambda, e_\lambda{}'\} \subseteq range(H_{err})$. $e_\lambda \underset{H_{err}}{\approx} e_\lambda{}' \iff hash(e_\lambda) = hash(e_\lambda{}')$ and hence is well-defined. Therefore, it is easy to see that $\Gamma_{err} \sim (\Gamma^\sharp, \sigma^\sharp(st))$ by our construction. Hence, $\langle \Gamma_{err}, e\rangle \rightsquigarrow^n \langle \Gamma'_{err}, c\ a\rangle$ and $\exists S.\Gamma' \sim (\Gamma'_{err}, S)$. Clearly, evaluating $(c\ a)$ under $\Gamma'_{err}$ results in a contract violation as the precondition of $g$ will not hold. Hence, the contact of $f$ cannot hold in all environments in $Env^c$.

□

## 4.6 Model Verification and Inference

In this section, we discuss the approach for verifying contracts of the model programs generated as described in the previous chapter. In principle, since the model program uses only first-order features the techniques described in Chapter 3 can be used to verify the model program. However, as I will describe in this section, applying the function-level, modular assume-guarantee reasoning will result in obligations which to be established require dramatically more specifications (which are provided by the user). In other words, the simple function-level, modular reasoning increases the contact annotation overhead in the programs dramatically. To address this difficulty I introduce an extension to the assume-guarantee reasoning: *creation-dispatch reasoning*, which propagates cache-monotonic properties that hold at creation point of closure to the invocation points.

I now explain this difficulty in applying the traditional, function-level modular reasoning for verifying model programs using the example shown in Figure 4.4, which was generated for the lazy take function shown in Figure 4.3.

**Challenges in Modular Reasoning for Model Programs.** As quick recap consider again the function-level assumed guarantee obligations described in section 3.2.

For each function definition **def** $f\ x := \{pre\}\ e\ \{post\}$ in a program $P$,
(2.I) $\models_P pre \rightarrow post[e/\mathsf{res}]$
(2.II) For each call site $(f\ y)^\ell$ in $P$, $\models_P path((f\ y)^\ell) \rightarrow pre(f\ y)$

Here $e_1 \rightarrow e_2$ denotes a semantic implication that in all environments having a binding for the free variables of $e_1$ and $e_2$, whenever $e_1$ does not evaluate to *false*, $e_2$ evaluates to *true*. The notation $\models_P e_1 \rightarrow e_2$ denotes that under the *assumption* that all functions invoked by $e_1$ and $e_2$ terminate in all environments that reach them, and their pre-and post-conditions hold,

$e_1 \rightarrow e_2$ is *guaranteed*. The path condition $path(c)$ denotes the static path to $c$ from the entry of the function containing $c$ and is defined in Figure 3.6. $pre(c)$ denotes the precondition of $f$ after translation to the argument of the call $c$. As noted earlier, this modular reasoning requires that the assume/guarantee assertions hold for all environments $\Gamma \in Env^c$, even though for contract verification it suffices to consider only valid environments that reach the function bodies. In fact, in the case of model programs it suffices to consider the environments in $Env^\sharp{}_{e,P}$ defined in section 4.5.1. This obligation dramatically increases the specification/verification overhead when applied as such to the model programs.

For example, consider the call to take$^\sharp$ (n1,s1,st) within app at line 11 in the program shown in Fig. 4.4. The path condition to the call is cl = Take (n1,s1). Obviously, this is not strong enough to imply the precondition of the call namely concrUntil$^\sharp$ (s1,n1,st). To make this example verify, it would in fact require concrUntil$^\sharp$ to hold on the arguments of every instance of Take reachable from the recursive datatype Stream, due to the mutual recursion between app, take$^\sharp$ and tail$^\sharp$. That is, the precondition of app would need a function pre (cl,st) defined as follows:

```
def pre (cl,st) = cl match{
  Take (n1,s1) ⇒ concrUntil♯ (s1,n1,st) ∧
                        (s1 match {
                           SCons(x, t) ⇒ pre (t, st);
                           SNil ⇒ true
                        });
}
```

What complicates this further is that to ensure this precondition at the call to app at line 7, the precondition of the function tail$^\sharp$ and all its transitive callers (including take$^\sharp$) should be modified similarly. This scenario happens very often when dealing with recursive, lazy data structures [Okasaki, 1998]. Our initial attempts to synthesize a precondition such as the above for *App* functions resulted in formulas too complicated for the state-of-the-art SMT solvers to solve.

In the sequel, I discuss an approach to alleviate this specification overhead based on the observation that the property concrUntil$^\sharp$ actually holds at the points where the closure Take is created and is monotonic with respect to the changes to the cache.

**Asserting Cache Monotonic Properties** Recall that the contracts of all functions in source program are required to be cache monotonic (section 4.2). This ensures that the source expressions of the language remain referentially transparent. To check if a property *pr* is cache monotonic it suffices to check the following property on the translation of *pr* with respect to $[\![\cdot]\!]_P$ defined in Figure 4.7:

$$st_1 \subseteq st_2 \wedge [\![pr]\!]_P\ st_1) \rightarrow [\![pr]\!]_P\ st_2$$

With this observation I now present the creation-dispatch reasoning for verifying model programs.

### 4.6.1 Creation-Dispatch Reasoning

Let $P$ be a program and $P^\sharp$ be the model program generated for $P$. Recall that each indirect call $x\ y$ has a set of target lambdas that are estimated at the time of model construction based on $type_P(x)$. Let $\Lambda = \{e_i \mid i \in [1,n]\}$, where $e_i = \lambda x.f_i\ (x, y_i)$, be the lambdas in the program that are the possible targets of encapsulated calls in the program $P$. (Recall the definition of encapsulated calls defined in section 2.3.) Let $CloCons = \{C_i\ w_i \mid i \in [1,n]\}$ be the closure constructions in the model program generated for $P$ representing the lambdas $\Lambda$. In the model program, the dispatch functions $App_\ell$ corresponding to the encapsulated calls invoke the function $f_i^\sharp$ (the translation of $f_i$) in each case $C_i\ w_i$ (see Fig. 4.7 and the illustration Fig 4.4). Let $DispCalls = \{f_i^\sharp\ (x, z_i, st) \mid i \in [1,n]\}$ be the calls invoked by such dispatch functions $App_\ell$. Note that $st$ is the state parameter added by the translation.

For instance, for the program shown in Figure 4.3 and its model shown in Fig. 4.4 the above sets would be defined as follows:

| | |
|---|---|
| $\Lambda$ | $= \{\lambda a.\ \mathsf{takeLazy}\ (\mathsf{n1},\mathsf{t})\}$ |
| *CloCons* | = {TakeLazy (n1,r_.1)} constructed at line 24 |
| *DispCalls* | = {takeLazy (n1,s1,st)} called at line 11 |

Let $Props = \{\rho_i \mid i \in [1,n]\}$ be a set of boolean-valued expressions (predicates) in $E_{spec}$ defined on the captured argument $y_i$ of the lambda $e_i \in \Lambda$ (i.e, $\rho_i$ has only $y_i$ as free variable).

The creation *creation-dispatch* reasoning allows augmenting the function-level assume/guarantee rules presented earlier with the following condition: if each property $\rho_i$ is cache monotonic (rule (3.III)), and hold at the point of creation of the lambda $e_i$ for the state of the cache at that point (rule (3.IV)), then it can be assumed to hold at the point of dispatch (rule (3.V)). Formally,

***Obligations for creation-dispatch reasoning***

(3.I) For each **def** $f\ x := \{pre\}\ e\ \{post\} \in P^\sharp$, $\models_{P^\sharp} pre \rightarrow post[e/\mathsf{res}]$

(3.II) For each call site $c \notin DispCalls$, $\models_{P^\sharp} path(c) \rightarrow pre(c)$

(3.III) *(Cache monotonicity)* For each $\rho_i \in Props$
$\models_{P^\sharp} (st_1 \subseteq st_2 \wedge [\![\rho_i]\!]_P\ st_1) \rightarrow [\![\rho_i]\!]_P\ st_2$

(3.IV) For each closure construction site $c = C_i\ w_i$ in *CloCons*
$\models_{P^\sharp} path(c) \rightarrow ([\![\rho_i]\!]_P\ st(c))$

(3.V) For each call site $c = f_i^\sharp\ (x, z_i, st)$ in *DispCalls*
$\models_{P^\sharp} (path(c) \wedge [\![\rho_i[z_i/y_i]]\!]_P\ st) \rightarrow pre(c)$

It may appear at first glance that the above reasoning in facts increases the number of proof obligations. The main advantage of the above obligations is the rule (3.V) which permits assuming the property $\rho_i$ for establishing the preconditions of a call in *DispCalls*, which corre-

spond to indirect calls. In other words, verifying preconditions of indirect calls can assume properties that hold at the point of creation of the invoked closure, provided the property is cache monotonic. Note that, by the definition of $\models_{P^\sharp}$, each obligation can assume that function all functions invoked by the expressions being checked terminate in all environments that reach them, and their pre-and post-conditions hold

In the above rules, $st(c)$ denotes the cache-state expression propagated by the translation function $[\![\cdot]\!]_P$ to an expression $c$ in the model program. Note that there is exactly one cache-state expression reaching every point in the model program by the definition of the translation shown in Fig. 4.7. For instance, the state expression reaching the line 11 of Fig. 4.4 is st, whereas the state expression reaching the line 24 is nst. This can be determined by a simple syntactic analysis of the model programs.

While the above reasoning holds irrespective of the how the properties $\rho_i$ are chosen for each lambda $e_i$, a particular strategy is implemented in our implementation. For each $e_i = \lambda x.f_i\ (x, y_i)$, $\rho_i$ is chosen to be the disjuncts of the precondition of the call $f_i\ (x, y_i)$ that only refer to the captured variable $y_i$. For example, for the model shown in Figure 4.4, our approach would verify the following:
(a) concrUntil is a cache monotonic property: $\models_{P^\sharp} (st_1 \subseteq st_2 \wedge \mathsf{concrUntil}(s, i, st_1)) \rightarrow \mathsf{concrUntil}(s, i, st_2)$
(b) The property concrUntil(u.$_1$, n−1,nst) is implied by the path condition at the point of creation of the closure Take(n−1,u.$_1$) at line 24. This is encoded by the following obligation (after a few straightforward simplifications):

$$\models_{P^\sharp} \quad \big(\mathsf{concrUntil}\ (\mathsf{s,n,st}) \wedge \mathsf{s} = \mathsf{SCons}\ (\mathsf{x,tfun}) \wedge \mathsf{u} = \mathsf{tail}^\sharp\ (\mathsf{s,st}) \wedge \mathsf{nst} = \mathsf{u}._2 \cup \{(\mathsf{Tail\ s})\}\big) \rightarrow \mathsf{concrUntil}\ (\mathsf{u}._1, \mathsf{n}-1, \mathsf{nst})$$

The above obligation follow by the definition of the function concrUntil shown in Figure 4.4. The property concrUntil (s1,n1,st) is assumed to hold while checking the precondition of call to take$^\sharp$ at line 11. With this extension we do not need any more preconditions than what is stated in the program to verify the model program.

If the functions in the input program have holes then the assume-guarantee obligations generated as above will also have holes. The problem then is infer values for the holes that will make the creation-dispatch obligation hold. These creation-dispatch obligations are solved using the inference algorithm presented in section 3.3. For the lazy take function shown in Figure 4.3, our algorithm inferred that it completed in at most 10 steps.

## 4.7 Correctness of Creation-Dispatch Reasoning

Similar to Lemma 10 of section 3.2, I now establish that the above creation-dispatch rules are essentially a part of an inductive reasoning. The induction order is the steps in the evaluation of an expression, or equivalently the depth of the big-step evaluation tree. This order is well-founded only for environments in which a function terminates and thus only entails partial

correctness. Our independently verifies the termination of programs using the termination checker of the LEON verification system [Nicolas Voirol and Kuncak, 2017].

However, a difference compared to function-level, modular reasoning is that, under this reasoning, soundness of verification of the contracts of one function say $f$ may be incumbent on the termination of *other* functions in the program, namely the ones that create the closures invoked by $f$. This is somewhat obvious since the facts are propagated from the creators to the dispatchers. For this reason, this creation-dispatch reasoning could be thought of as module-level (or class-level) modular reasoning as opposed to function-level reasoning.

**Encoding Model Language Extensions** Recall that the model programs use an error construct in the bodies of $App_\ell$ functions to handle (non-encapsulated) indirect calls. Let **def** $App_\ell\ (cl, x, st)$ be one such function corresponding to an indirect call $(y\ z)$. The error construct will be encountered during the evaluation of $App_\ell$ if and only if $cl = C_{type_P(y)}$. In this case, the result of the evaluation is undefined. The same effect can be achieved if we add a precondition to $App_\ell$ namely $cl \neq C_{type_P(y)}$. It is obvious that the $App_\ell$ with the precondition is equivalent to the $App_\ell$ function with the error construct. For simplicity, in the rest of section, we assume that the model programs are free of error constructs, which have been lifted to the preconditions of $App_\ell$ functions. This provides us the property that the Lemma 4, which states that undefined evaluation are possible only due to non-termination or contract failures, applies to the model programs as well.

Also for the simplicity of the proof it is assumed that $([\![\rho_i]\!]_P\ st(c))$ is invoked just before the construction site $c = (C_i\ w_i)$, and that the result of $\rho_i$ is ignored by model program. That is, the closure construction $(C_i\ w_i)$ is replaced by **let** _ := $([\![\rho_i]\!]_P\ st(c))$ in $(C_i\ w_i)$. It is obvious that this transformation is semantics preserving. But the benefit from the perspective of the proof is that it simplifies the statement of the following theorems, which now only have to reason about functions defined in the program, and not an additional property not in the program.

### 4.7.1 Partial Correctness of Creation-Dispatch Obligations

The following lemma establishes that whenever a model program $P^\sharp$ corresponding to a input program $P$ satisfies the assume-guarantee obligations, then one of the following properties hold for all natural number $n$, for all clients $P'$ that close the program $P$ and environment $\Gamma \in Env^c$: (a) either there exists a function $f$ in the input program $P$ such that the closed evaluation under $P' \| P$ reaches the body of the function with the environment $\Gamma$ and the evaluation of the model function $f^\sharp$ under the evaluation corresponding environment $\Gamma^\sharp$ takes more than $n$-steps (with respect to $\rightsquigarrow$ relation). (b) Otherwise, whenever the closed evaluation under $P' \| P$ reaches the body of the function with the environment $\Gamma$, the contracts of the model function $f^\sharp$ are satisfied under the corresponding environment $\Gamma^\sharp$.

I would like to remark that this intellectual complexity in the formulation of the lemma statement is the result of the fact that the valid (or relevant) environments of the model programs are constrained not by the clients of the model programs that closes it, but rather by clients of the original program. This is because the clients of the model programs are more unconstrained than the clients of the source programs (see section 4.5.1). A more intuitive abstraction of the following lemma would be that if the creation-dispatch obligations hold for the model program then either (a) the contracts of all functions in the model program hold for all the *relevant* environments, which are environments that that have an one-to-one correspondence with the valid environments of the source programs as defined in section 4.5.1, or (b) at least one function in the model program does not terminate under a relevant environment. Note that as established by Theorem 26 it suffices to consider all *relevant* environments of the model program to verify the contracts of the original program.

In the proof shown below I use the following convention. If $\Gamma^\sharp \in Env$ is an environment of the model program then I denote by $\Gamma$ corresponding environment in $Env^c$ with respect to the relation $\sim$ i.e, $\exists S.\Gamma \sim (\Gamma^\sharp, S)$.

**Lemma 28.** *Let $P$ be a program and $P^\sharp$ the model program. If every function defined in $P^\sharp$ satisfy the assume-guarantee obligations 3.(I) to 3.(V) defined above, the following property holds for all $n \in \mathbb{N}$*

$$
\begin{aligned}
&\forall\ \textit{program}\ P'.\forall \Gamma \in Env^c.\\
&\quad \big(\exists (\textbf{\textit{def}}\ f\ x := \tilde{e}) \in P, \Gamma^\sharp \in Env^\sharp\ \textit{s.t.}\ \langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle \wedge \Gamma \sim (\Gamma^\sharp, \sigma^\sharp(st)) \wedge\\
&\qquad \exists k > n, e', \Gamma'. \langle \Gamma^\sharp, [\![\tilde{e}]\!]_P\ st\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle\big) \vee\\
&\quad \forall\, \textbf{\textit{def}}\ f\ x := \tilde{e} \in P, \tilde{e} = \{p\}\ e\ \{s\}, \Gamma^\sharp \in Env.\\
&\qquad \textit{if}\ \big(\langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle\big) \wedge \Gamma \sim (\Gamma^\sharp, \sigma^\sharp(st))\ \textit{then}\\
&\qquad \Big(\exists v. \Gamma^\sharp \vdash [\![p]\!]_P\ st \Downarrow \textit{false} \vee \Gamma^\sharp \vdash [\![\tilde{e}]\!]_P\ st \Downarrow v\Big)
\end{aligned}
$$

*Proof.* We prove this by induction on $n$. Intuitively, $n$ limits the depth of evaluation of any expression in the model program $P^\sharp$ during a run starting from the entry expression $e_{entry}$ with respect to a client $P'$. The base case is when $n = 1$. Consider a function definition **def** $f\ x := \tilde{e}$. Let $e' = [\![\tilde{e}]\!]_P\ st$, and let $e' = \{p'\}\ b'\ \{s'\}$. Let $\langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle$ and $\Gamma \sim (\Gamma^\sharp, \sigma^\sharp(st))$.

Now, if the evaluation of $e'$ under $\Gamma^\sharp$ has depth more than 1 then the claim trivially holds. Therefore, say the evaluation of $e'$ under $\Gamma^\sharp$ has depth at most 1. Hence, it cannot make any function calls. (Note that there are only direct calls in the model program.)

$$
\begin{aligned}
&\mathit{Calls}(\Gamma^\sharp, p') \cup \mathit{Calls}(\Gamma^\sharp, e') = \emptyset\\
&\text{By 3.1 – 3.7 discussed in section 3.2 }, \exists v. \Gamma \vdash p' \Downarrow \textit{false} \vee \Gamma \vdash e' \Downarrow v
\end{aligned}
$$

Hence the claim holds in the base case.

Now, consider the inductive case and say the claim holds upto some number $m$. Now, if the

evaluation of $e'$ under $\Gamma^\sharp$ has depth more than $m$ then the claim trivially holds. Therefore, say the evaluation of $e'$ under $\Gamma^\sharp$ has depth at most $m+1$.

(a) Say $\neg\exists c \in \mathit{DispCalls}, \Gamma'.\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle c, \Gamma'\rangle$. In this case, for all $(c, \_) \in \mathit{Calls}(\Gamma^\sharp, e')$, $\models_P \mathit{path}(c) \rightarrow \mathit{pre}(c)$ holds. Hence, by the argument 3.8 – 3.20 of section 3.2 , $\exists v.\Gamma \vdash p' \Downarrow \mathit{false} \vee \Gamma \vdash e' \Downarrow v$.

(b) Now say there exists a $c = g^\sharp\ (x, z, st)$ belonging to *DispCalls* and $\Gamma^\sharp_3 \in \mathit{Env}$ such that $\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle c, \Gamma^\sharp_3\rangle$. Let $w = \sigma^\sharp_3(z)$. By the definition of *DispCalls* and the model translation,

$$\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle \mathit{App}_\ell\ (y, a, st'), \Gamma^\sharp_1\rangle \rightsquigarrow^* \langle c, \Gamma^\sharp_3\rangle$$

where $\sigma^\sharp_1(H^\sharp_1(y)) = (C_g\ w)$. That is, $\Gamma^\sharp_1$ is the environment that reaches the *App* function. By Lemma 24, $\langle \tilde{e}, \Gamma\rangle \rightsquigarrow^* \langle (y\ q), \Gamma_1\rangle$ (for some $q$) and $\Gamma_1 \sim (\Gamma^\sharp_1, \sigma^\sharp_1(st'))$. Therefore, there exists an address $a$ such that $\sigma_1(y) = a$, $H_1(a)$ is a closure $((\lambda x.g\ (x, p))^\ell, [p \mapsto v])$ and $v \sim w$.

By definition of *DispCalls*, the call $(y\ q)$ is an encapsulated call. Therefore, $(\lambda x.g\ (x, p))^\ell$ belongs to the program $P$ i.e, $l \in \mathit{labels}_P$. Let "**def** $cr\ x = \{\ e_b\ \}$" be the function in $P$ that contains the lambda with label $\ell$. The closure $((\lambda x.g\ (x, p))^\ell, [p \mapsto v])$ should have been created at some point during the run starting from $e_{entry}$. Therefore, there exists a sequence:

$$\langle \Gamma_{P' \| P}, e_{entry}\rangle \rightsquigarrow^* \langle \_, cr\ x\rangle \rightsquigarrow \langle \Gamma_{in}, e_b\rangle \rightsquigarrow^* \langle \Gamma_0, \lambda x.g\ (x, p)\rangle \tag{4.1}$$

such that

$$\Gamma_0 \vdash \lambda x.g\ (x, p) \Downarrow a, (C_0, H_0[a \mapsto (\lambda x.g\ (x, p), [p \mapsto v])], \sigma_0)$$

where $H_0 \sqsubseteq H_1$ and $C_0 \sqsubseteq C_1$. Let ${\Gamma^\sharp}_{in}$ be such that $\Gamma_{in} \sim ({\Gamma^\sharp}_{in}, \sigma^\sharp_{in}(st))$ and let $e'_b = [\![e_b]\!]_P\ st$.

I now establish the following sub-property which shows that if there exists a environment $\Gamma_{in}$ under which $\langle \Gamma_{in}, e_b\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle$ in the source program, either (a) an evaluation starting form $\langle {\Gamma^\sharp}_{in}, e'_b\rangle$, where $e'_b$ is the translation of $e_b$, takes more than $m+1$ steps or (b) an environment similar to $\Gamma'$ would reach the translation of $e'$.

**Property 5.** *For all $k \in \mathbb{N}$, for all $\Gamma_{in} \in {\mathit{Env}^\sharp}_{e_b,P}$ and for all ${\Gamma^\sharp}_{in}$ such that $\Gamma_{in} \sim ({\Gamma^\sharp}_{in}, \sigma^\sharp_{in}(st))$.*
*If $\langle \Gamma_{in}, e_b\rangle \rightsquigarrow^k \langle \Gamma', e'\rangle$ then*
*(a) there exists an chain $\langle {\Gamma^\sharp}_{in}, e'_b\rangle \rightsquigarrow^r \_$ and $r > m+1$, or*
*(b) $\exists s.\ \langle {\Gamma^\sharp}_{in}, e'_b\rangle \rightsquigarrow^* \langle {\Gamma^\sharp}', [\![e']\!]_P\ s\rangle$ and $\Gamma' \sim ({\Gamma^\sharp}', S)$ and ${\Gamma^\sharp}' \vdash s \Downarrow S$.*

*Proof.* This property is proved by induction on $k$. The inductive and base cases are very similar and so are proven together as shown below. Let *is* and $e$ be expressions and let $e' = ([\![e]\!]_P\ is)$ be the translation of $e$ with respect to *is*. Let $\Gamma^\sharp$ be some expression such that $\langle \Gamma^\sharp, e'\rangle$ is reachable from $\langle {\Gamma^\sharp}_{in}, e'_b\rangle$ and $\Gamma \sim (\Gamma^\sharp, S)$ and $\Gamma^\sharp \vdash is \Downarrow S$. Say $\langle \Gamma, e\rangle \rightsquigarrow \langle \Gamma_o, e_o\rangle$. I now establish that an environment similar to $\Gamma_o$ reaches the translation of $e_o$, or the property (a) above holds. The

claim follows from this by induction.

In the $\rightsquigarrow$ relations (shown in Fig. 2.3) introduced by all rules except LET and CONTRACT, the environments $\Gamma$ and $\Gamma_o$ (the input and the output environments) differ only by the store component. By the definition of the translation and the operational semantics it is easy to see that there exists an ${\Gamma^\sharp}_o$ such that $\langle \Gamma^\sharp, e' \rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_o]\!]_P\ is \rangle$ and $\Gamma_o \sim ({\Gamma^\sharp}_o, S)$ and ${\Gamma^\sharp}_o \vdash is \Downarrow S$.

Consider now the rule LET. Let $e = \textbf{let}\ x\ :=\ e_1\ \textsf{in}\ e_2$. By the definition of the translation: $e' = \textbf{let}\ x\ :=\ [\![e_1]\!]_P\ is\ \textsf{in}\ [\![e_2]\!]_P\ x.2$. There are two $\rightsquigarrow$ relations introduced by the rule. Consider the first relation introduced by LET: $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma, e_1 \rangle$. A similar relation will be introduced in the translated expression: $\langle \Gamma^\sharp, e' \rangle \rightsquigarrow \langle \Gamma^\sharp, [\![e_1]\!]_P\ is \rangle$, which clearly satisfies the claim.

Consider the other relation introduced by the LET rule defined as follows: If $\Gamma \vdash e_1 \Downarrow u_1, \Gamma_1$ then $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma_o, e_2 \rangle$, where $\Gamma_o = (C_1, H_1, \sigma_1[x \mapsto u_1])$. We also have similar relation for the translated expression. If $\Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v, {\Gamma^\sharp}_1$ then $\langle \Gamma^\sharp, e' \rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_2]\!]_P\ x.2 \rangle$, where ${\Gamma^\sharp}_o = (H_1^\sharp, \sigma_1^\sharp[x \mapsto v_1])$. Now there are two cases to consider

(a) There exists a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st \rangle \rightsquigarrow^r \_$ and $r > m$. In this case, there exists an chain $\langle {\Gamma^\sharp}_{in}, e'_b \rangle \rightsquigarrow^r \_$ and $r > m+1$, since are given that $\langle \Gamma^\sharp, e' \rangle$ is reachable from $\langle {\Gamma^\sharp}_{in}, e'_b \rangle$ (in one or more steps). Hence the claim holds.

(b) There does not exist a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st \rangle \rightsquigarrow^r \_$ and $r > m$.
I now claim that $\exists v_1.\ \Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v_1, {\Gamma^\sharp}_1$. This is because, by Lemma 4, the evaluation could be undefined only if either the evaluation does not terminate or because there is a contract violation during the evaluation. The former case is not possible since there does not exist a chain $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st \rangle \rightsquigarrow^r \_$ and $r > m$. The latter case is not possible because every call $(h\ g)$ reached during the evaluation (with an environment ${\Gamma^\sharp}''$) cannot have a chain $\langle {\Gamma^\sharp}'', (h\ g) \rangle \rightsquigarrow^r \_$ and $r > m$ (otherwise $\langle \Gamma^\sharp, [\![e_1]\!]_P\ st \rangle \rightsquigarrow^r \_$ and $r > m$, which contradicts the given fact). Therefore, by the (outer) induction hypothesis the call should produce a value. That is, there can be no contract violation.

It has now been shown that $\Gamma^\sharp \vdash [\![e_1]\!]_P\ st \Downarrow v_1, {\Gamma^\sharp}_1$ is defined. Therefore, $\langle \Gamma^\sharp, e' \rangle \rightsquigarrow \langle {\Gamma^\sharp}_o, [\![e_2]\!]_P\ x.2 \rangle$ is defined. By Lemma 24, $\Gamma \vdash e_1 \Downarrow u_1, \Gamma_1$ is defined. Thus, $\langle \Gamma, e \rangle \rightsquigarrow \langle \Gamma_o, e_2 \rangle$ is also defined and $\Gamma_o \sim ({\Gamma^\sharp}_o, S')$ and ${\Gamma^\sharp}_o \vdash x.2 \Downarrow S$. Hence, the claim holds. The rule contract can be similarly proven. □

With this above established claim let us again revisit the evaluation sequence given by (4.1). Due to the above property, we know that one of the following cases hold: (a) either there exists $\langle {\Gamma^\sharp}_{in}, e'_b \rangle \rightsquigarrow^r \_$ and $r > m+1$. or (b) $\langle {\Gamma^\sharp}_{in}, e'_b \rangle \rightsquigarrow^* \langle {\Gamma^\sharp}_0, (C_\ell\ p) \rangle$ and $\exists s.\Gamma_0 \sim ({\Gamma^\sharp}_0, S)$ and ${\Gamma^\sharp}_0 \vdash s \Downarrow S$. In the former case the lemma holds as the first disjunct of the lemma is satisfied as ${\Gamma^\sharp}_0$ belongs to $Env^\sharp_{e', P^\sharp}$ (which is the set of relevant environments reaching $e'$ as defined in section 4.5.1). Therefore consider the latter case.

Let $cc = (C_\ell\ p)$. By definition, $st(cc)$ is the state expression reaching the construction site $cc$.

Therefore $s = st(cc)$. By the definition of *path*, for any function definition **def** $f\ x := e'_b$ and closure construction site $cc$ in $f$.

$$\forall \Gamma^\sharp \in Env^\sharp. \langle \Gamma^\sharp, e'_b \rangle \rightsquigarrow^* \langle \Gamma^{\sharp\prime}, cc \rangle \Rightarrow \Gamma^{\sharp\prime} \vdash path(cc) \Downarrow true \tag{4.2}$$

$$\text{Therefore, } \Gamma^\sharp{}_0 \vdash path(cc) \Downarrow true \tag{4.3}$$

$$\Rightarrow \mathscr{A}(\Gamma^\sharp{}_0, path(cc)) \tag{4.4}$$

The assumption $\mathscr{A}$ is defined in sectoin 3.2. Let $\rho' = (\llbracket \rho_i \rrbracket_P\ st(c))$. We know that the $FV(\rho_i) \subseteq \{p\}$, where $p$ is argument of the constructor. Now, by the assume-guarantee reasoning obligation 3.(IV), we are given that

$$\models_P path(cc) \rightarrow \rho' \tag{4.5}$$

$$\Rightarrow \forall \Gamma^{\sharp\prime}. \neg \mathscr{A}(\Gamma^{\sharp\prime}, path(cc)) \vee \Gamma^{\sharp\prime} \vdash path(cc) \Downarrow false$$

$$\vee \neg \mathscr{A}(\Gamma^{\sharp\prime}, \rho') \vee \Gamma^{\sharp\prime} \vdash \rho' \Downarrow true \tag{4.6}$$

$$\Rightarrow \neg \mathscr{A}(\Gamma^\sharp{}_0, \rho') \vee \Gamma^\sharp{}_0 \vdash \rho' \Downarrow true, \quad \text{by 4.3, 4.4} \tag{4.7}$$

Recall that we have assumed that $\rho'$ is invoked just before the closure construction $cc$. Therefore, $\exists \Gamma^\sharp \sqsubseteq \Gamma^\sharp{}_0$ such that $\exists v. \Gamma^\sharp \vdash \rho' \Downarrow v$, since we are given that $\langle \Gamma^\sharp{}_0, e'_b \rangle \rightsquigarrow^* \langle \Gamma^\sharp{}_0, cc \rangle$. Hence, $\mathscr{A}(\Gamma^\sharp, \rho')$ holds. It is easy to see that, $Calls(\Gamma^\sharp, \rho') = Calls(\Gamma^\sharp{}_0, \rho')$ since $\Gamma^\sharp \sqsubseteq \Gamma^\sharp{}_0$. (Note that the model program does not have memoization and is purely functional.) Therefore, $\mathscr{A}(\Gamma^\sharp{}_0, \rho')$ also holds. Substituting this in 4.7 we get, $\Gamma^\sharp{}_0{}' \vdash \rho' \Downarrow true$. By Lemma 24, $\Gamma_0 \vdash \rho_i \Downarrow true$.

Now, we know that $H_0 \sqsubseteq H_1$ and $C_0 \sqsubseteq C_1$, where $H_0$ and $C_0$ are the environments at the creation point of the closure (see the definition of (4.1)). It is also given by the assume-guarantee obligation 3.(III) that

$$\models_P st_1 \subseteq st_2 \wedge \llbracket \rho_i \rrbracket_P\ st_1 \rightarrow \llbracket \rho_i \rrbracket_P\ st_2 \tag{4.8}$$

$$\Rightarrow \forall \Gamma^\sharp \text{ s.t. } dom(\sigma^\sharp) \subseteq FV(\rho_i) \cup \{st_1, st_2\}.\ \neg \mathscr{A}(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_1) \vee \mathscr{A}(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2)$$

$$\vee \Gamma^\sharp \vdash (st_1 \subseteq st_2) \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_1 \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true \tag{4.9}$$

$$\Rightarrow \forall \Gamma^\sharp \text{ s.t. } \Gamma^\sharp{}_0 \sqsubseteq \Gamma^\sharp \wedge st_2 \in dom(\sigma^\sharp).\ \Gamma^\sharp \vdash (st(cc) \subseteq st_2) \Downarrow false \vee$$

$$\neg \mathscr{A}(\Gamma^\sharp, \rho') \vee \mathscr{A}(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2) \vee \Gamma^\sharp \vdash \rho' \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true$$

By the definition of the model programs, the depths of the evaluations of the expressions of the model program are independent of the state parameter. (This is because the translation always invokes a function even if it a hit in the cache.) Recall that as shown by Fig. 4.4 the state parameter only influences the value of the last element of the tuple, namely the resource usage component. Therefore, $Calls(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2) = Calls(\Gamma^\sharp, \rho')$, where $\rho' = (\llbracket \rho_i \rrbracket_P\ st(c))$ as defined before. We are given that $\mathscr{A}(\Gamma^\sharp{}_0, \rho')$ holds. Therefore as $\Gamma^\sharp{}_0 \sqsubseteq \Gamma^\sharp$, $\mathscr{A}(\Gamma^\sharp, \rho')$ and $\mathscr{A}(\Gamma^\sharp, \llbracket \rho_i \rrbracket_P\ st_2)$ also holds. Substituting this and the fact that $\Gamma_0 \vdash \rho_i \Downarrow true$ in 4.10 we get,

$$\forall \Gamma^\sharp \text{ s.t. } \Gamma^\sharp{}_0 \sqsubseteq \Gamma^\sharp \wedge st_2 \in dom(\sigma^\sharp).\ \Gamma^\sharp \vdash (st(cc) \subseteq st_2) \Downarrow false \vee \Gamma^\sharp \vdash \llbracket \rho_i \rrbracket_P\ st_2 \Downarrow true$$

By Lemma 24 that relates the state $st(cc)$ in the model program to the cache reaching the lambda corresponding to $cc$ in the source program, and the totality of $\sim$ relation, the above implies that

$$\forall C_1 \in Cache.\ \neg(C_0 \sqsubseteq C_1) \vee (C_1, H_0, \sigma_0) \vdash \rho_i \Downarrow true \tag{4.10}$$
$$\Rightarrow \forall \Gamma' \in Env.\neg(\Gamma_0 \sqsubseteq \Gamma') \vee \Gamma' \vdash \rho_i \Downarrow true \tag{4.11}$$

Since we know $C_0 \sqsubseteq C_1$ and $H_0 \sqsubseteq H_1$ (see Definition 4.1). The above implies that

$$(C_1, H_1, \sigma_0) \vdash \rho_i \Downarrow true \tag{4.12}$$
$$(C_1, H_1, [p \mapsto \sigma_0(p)]) \vdash \rho_i \Downarrow true, \quad \text{since } FV(\rho_i) \subseteq \{p\} \tag{4.13}$$

We are given that $\sigma_0(p) = v$, $v \underset{H_1,H_3^\sharp}{\sim} w$, $\sigma_3^\sharp(z) = w$, $C_1 \underset{H_1,H_1^\sharp}{\sim} \sigma_1^\sharp(st')$, $\sigma_1^\sharp(st') = \sigma_3^\sharp(st)$ and $H_1^\sharp \sqsubseteq H_3^\sharp$. The last three facts imply that $C_1 \underset{H_1,H_3^\sharp}{\sim} \sigma_3^\sharp(st)$. Hence,

$$(C_1, H_1, [p \mapsto \sigma_0(p)]) \underset{H_1,H_3^\sharp}{\sim} ((H_3^\sharp, [p \mapsto \sigma_3^\sharp(z)]), \sigma_3^\sharp(st))$$

Therefore, by Lemma 24 and 4.13,

$$(H_3^\sharp, [p \mapsto \sigma_3^\sharp(z)]) \vdash [\![\rho_i[z/p]]\!]_P\ st \Downarrow true \tag{4.14}$$
$$\Gamma^\sharp{}_3 \vdash [\![\rho_i[z/p]]\!]_P\ st \Downarrow true \tag{4.15}$$

We are given that $\langle e', \Gamma^\sharp\rangle \rightsquigarrow^* \langle c, \Gamma_3^\sharp\rangle$, where $e' = [\![\tilde{e}]\!]_P\ st$. By the assume-guarantee obligation 3.(V),

$$\models_P (path(c) \wedge [\![\rho_i[z_i/y_i]]\!]_P\ st) \rightarrow pre(c) \tag{4.16}$$
$$\Rightarrow \Gamma^\sharp{}_3 \vdash pre(c) \Downarrow true, \tag{4.17}$$
by the reasoning shown in 3.15 – 3.17 and 4.15

Therefore, for every $(\Gamma^{\sharp\prime}, c) \in Calls(\Gamma^\sharp, e')$ where $c$ is an instance of *DispCalls*, $\Gamma^{\sharp\prime} \vdash pre(c) \Downarrow true$. By the reasoning shown in 3.17 and 4.15 of section 3.2, for every $(\Gamma^{\sharp\prime}, c) \in Calls(\Gamma^\sharp, e')$ where $c$ is not an instance of *DispCalls*, $\Gamma^{\sharp\prime} \vdash pre(c) \Downarrow true$. Therefore, as shown by 3.18 – 3.20 of section 3.2, $\mathscr{A}(\Gamma^\sharp, e')$ and $\mathscr{A}(\Gamma^\sharp, p')$. Hence, by the assume-guarantee obligation 3.(I), $\exists v.\Gamma^\sharp \vdash [\![p]\!]_P\ st \Downarrow false \vee \Gamma^\sharp \vdash [\![\tilde{e}]\!]_P\ st \Downarrow v$. □

Below I establish the main theorem of this section.

**Theorem 29** (***Partial-correctness of creation-dispatch reasoning***)**.** *Let $P$ be a program and $P^\sharp$ the model program. Let* ***def*** $f^\sharp\ x := \tilde{e}$ *where* $\tilde{e} = \{p\}\ e\ \{s\}$ *be a function definition in $P^\sharp$. If every function defined in $P$ terminate and the assume/guarantee assertions 3.(I) to 3.(V) defined above*

*hold, the contracts of $f^\sharp$ holds for all relevant environments $Env^\sharp_{\tilde{e},P^\sharp}$ i.e,*

$$\forall \Gamma^\sharp \in Env^\sharp_{\tilde{e},P^\sharp}.\exists u.\ \Gamma^\sharp \vdash p \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$$

*Proof.* Let **def** $g^\sharp\ x := \tilde{e}'$ be a function definition in $P^\sharp$, where $\tilde{e}' = [\![e]\!]_P\ st$. Let $\Gamma^\sharp \in Env^\sharp_{\tilde{e}',P^\sharp}$. By definition, there exists a $\Gamma \in Env^c$ and a program $P'$ such that $\big(\langle \Gamma_{P'\|\mathbb{P}}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma, \tilde{e}\rangle\big) \wedge \Gamma \sim (\Gamma^\sharp, \sigma^\sharp(st))$. Now say there exists an infinite chain of the form $\langle \Gamma^\sharp, [\![\tilde{e}]\!]_P\ st\rangle \rightsquigarrow \langle \Gamma^\sharp{}_1, [\![e_1]\!]_P\ s_1\rangle \rightsquigarrow \cdots$. By Lemma 24 and the definition of $\rightsquigarrow$ (Fig. 2.3), there exists an infinite chain: $\langle \Gamma, \tilde{e}\rangle \rightsquigarrow \langle \Gamma_1, e_1\rangle \rightsquigarrow \cdots$, which is a contradiction to the given fact that every function defined in $P$ is terminating. Therefore, there cannot be any infinite chains of the form: $\langle \Gamma^\sharp, [\![e]\!]_P\ st\rangle \rightsquigarrow \langle \Gamma^\sharp{}_1, [\![e_1]\!]_P\ s_1\rangle \rightsquigarrow \cdots$. Hence, the evaluation of $\langle \Gamma^\sharp, [\![e]\!]_P\ st\rangle$ is terminating

By the same argument, for every other function **def** $h^\sharp\ x := [\![\tilde{e}_h]\!]_P\ st \in P^\sharp$ there does not exist a $\Gamma^\sharp{}_h$ such that $\langle \Gamma_{P'\|\mathbb{P}}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma_h, \tilde{e}_h\rangle$ and $\Gamma_h \sim (\Gamma^\sharp{}_h, \sigma^\sharp_h(st))$ and $\langle \Gamma^\sharp{}_h, [\![\tilde{e}_h]\!]_P\ st\rangle \rightsquigarrow \cdots$ is infinite. Thus, there exists a $n \in \mathbb{N}$ such that

$$\neg\big(\exists \textbf{def}\ h\ x := \tilde{e}_h \in P, \Gamma^\sharp{}_h \in Env^\sharp \text{ s.t. } \langle \Gamma_{P'\|\mathbb{P}}, e_{entry}\rangle \rightsquigarrow^* \langle \Gamma_h, \tilde{e}\rangle \wedge \Gamma \sim (\Gamma^\sharp{}_h, \sigma^\sharp_h(st)) \wedge \exists k > n, e', \Gamma'. \langle \Gamma^\sharp{}_h, [\![\tilde{e}]\!]_P\ st\rangle \rightsquigarrow^k \langle \Gamma', e\rangle\big)$$

Hence, by Lemma 28, $\exists u.\ \Gamma^\sharp \vdash p \Downarrow \mathit{false} \vee \Gamma^\sharp \vdash \tilde{e} \Downarrow u$ for every function definition in $P^\sharp$. Hence, the contracts of the function $f^\sharp$ holds. □

## 4.8 Encoding Runtime Invariants and Optimizations

In this section, I discuss some of the optimizations and features supported by our system for improving automation or performance in verifying programs with higher-order features and memoization.

**Encoding Referential Transparency** Our system encodes certain invariants ensured by the runtime (i.e, operational semantics) that are not explicit in the model either as contracts or using *assume* constructs, which are treated by the verifier as facts that can be assumed without verification. For instance, the referential transparency of the *source* functions of the input program, namely that the result of the function is independent of the cache state, is encoded in model program in the following way. In principle, this corresponds to the axiom $\forall x, st_1, st_2.(f^\sharp\ (x, st_1)).{}_1 = (f^\sharp\ (x, st_2)).{}_1$ for every function $f^\sharp$ in the model. However this axiom is a binary axiom and instantiating it at the level of VCs may increase the sizes of the VC quadratically. Instead, these can be encoded by adding a predicate $\mathsf{res}.{}_1 = UF_f(x)$ in the postcondition of every non-specification function $f^\sharp$ of the model program, where $UF_f$ is a unique *uninterpreted function* for $f^\sharp$. This helps achieve a completely functional reasoning for correctness properties that only rely on the result of the evaluation.

**Encoding Cache Monotonicity** Our system encodes the monotonic evolution of the cache by adding the predicate: $st \subseteq \mathsf{res}.{}_2$ in the postcondition of every function $f^\sharp$ in the model program.

These facts are assumed in the postcondition every time the functions in the model program are unfolded.

**Encoding Reference Equality** Notice that thus far the core language only supports structural equality. However, to be consistent with the semantics of Scala for equality of closures, our system also supports reference equality for equating closures. This can be accomplished by associating a unique identifier with the datatype constructors that represents closures in the model program. The unique identifier is much like a global state which is passed through expression in a *store-passing style*. This instrumentation in automatically performed by the tool when this feature is enable through an option.

**Verifying Programs with Unknown Implementation** One of the main limitations of the technique presented here is that it is not possible to directly specify contracts and resource bounds of first-class functions, without specifying their implementation. This limits the approach in certifying open higher-order libraries that can accept arbitrary implementations. One way to encode contracts of such first-class functions without known implementations is by creating an *uninterpreted named function* in the program that could serve as a stub for the target of the indirect call. Such uninterpreted function without bodies can be specified using the an annotation @extern. These function can also take possible contracts, which are *assumed* while verifying their callers.

**The Trace-Induction Tactic** It is often necessary to establish properties of recursive functions automatically at various phases in the analysis. For instance, it is necessary to establish the monotonicity of predicates like concrUntil shown in Figure 4.3 for creation-dispatch reasoning. To alleviate user specifications required for such tasks, our system provides an annotation @traceInduct that implements an induction tactic. For instance, consider the function concrUntilMono shown below (in Scala syntax) that asserts the monotonicity of concrUntil in the model program. (Such a function is auto-generated by our tool to verify the monotonicity of the concrUntil.)

```
/** A recursive function of the source program **/
def concrUntil(s: SCons, i: BigInt): Bool = {
  if(i > 0) cached(s.tail) && concrUntil(s.tail, i−1)
  else true
}

/**
 * Encoding of the monotonicity property
 * in the model program using @traceInduct tactic
 * (Auto−generated code snippet)
**/
@traceInduct
def concrUntilMono(s: SCons, i: BigInt, st1: Set, st2: Set): Bool = {
  (st1.subsetOf(st2) && concrUntil(s,i,st1)) ==> concrUntil(s,i,st2)
} holds
```

```
/**
 * Expansion of the @traceInduct tactic
 * (Auto-generated code snippet)
**/
def concrUntilMonoTactic(s: SCons, i: BigInt, st1: Set, st2: Set): Bool = {
  (st1.subsetOf(st2) && concrUntil(s,i,st1) ==> concrUntil(s,i,st2)) &&
      concrUntilMonoTactic(s.tail, i-1, st1, st2)
} holds
```

The function concrUntilMonoTactic shows the translation that happens internally to realize the trace induction tactic. Notice that a recursive call to concrUntilMonoTactic is introduced with the same parameters as the recursive call in the definition of concrUntil. This essentially encodes induction over the recursive calls of concrUntil, which would normally be employed for verifying the postconditions of concrUntil (by the function-level assume-guarantee reasoning). But here it is used to verify a property that uses the function.

# 5 Empirical Evaluation and Studies

> It doesn't matter how beautiful your theory is,
> if it doesn't agree with experiment, it's wrong.
> — Richard Feynman

The approach detailed in this dissertation is built into the open-source LEON verification and synthesis framework [Blanc et al., 2013] available at the GitHub repository https://github.com/epfl-lara/leon. The sub-system of LEON that performs verification of resource bounds is named ORB. ORB extensively relies on LEON APIs, especially for front-end tasks such as parsing Scala programs and constructing Abstract Syntax Trees, and also for communicating with SMT solvers and cross-validating the results. However, the core components of ORB that implements the phases described in this dissertation are independent of the rest of the code base of LEON. These include the phases: resource instrumentation (section 3.1), model program generation (section 4.4), assume-guarantee obligation generation (section 3.2 and 4.6), verification condition generation (section 3.3.1) and inference of holes (section 3.3), which together comprise approximately 16K lines of Scala code when computed through `wc -l` (excluding benchmarks and test suites).

ORB is also integrated with the interactive online interface of LEON, accessible at http://leondev.epfl.ch and has a usage documentation at http://leondev.epfl.ch/doc/resourcebounds.html. Some of the benchmarks discussed in this section are available in the online interface. All benchmarks discussed or listed in this section are available at https://github.com/epfl-lara/leon/tree/inferInv/testcases.

ORB has been used to verify resource usage of many Scala programs implemented using the core language features detail in Chapter 2, and has been evaluated over multiple resources defined in the previous chapters. At the time of writing this dissertation the number of distinct benchmarks verified by the tool was about 50. The benchmarks together comprised approximately 8K lines of Scala code when computed through `wc -l`. Many of these benchmarks are verified over multiple resources such as steps, alloc, stack and depth. To my knowledge, no prior formal verification system, including interactive theorem provers [Bertot and Castéran,

2004, Nipkow et al., 2002a] has demonstrated the ability to verify resource complexity of some of the benchmarks verified by ORB (e.g. lazy data structures). ORB not only establishes (precise) asymptotic complexity of these benchmarks but also infers concrete upper bounds on algorithmic resources such as steps or alloc.

The following sections summarize the results of verifying benchmarks with ORB. The results presented here summarize the evaluations over 30 important benchmarks representing three classes of programs: (a) *first-order functional programs* comprising data structures like red-black tree and binomial heap, (b) *lazy data structures* such as those described in Okasaki [1998] and the Conqueue data structure of Scala's data parallel library [Prokopec and Odersky, 2015], and (c) *memoized algorithms* such as dynamic programming algorithms. Each class of benchmarks and their results are detailed in a separate section. The results presented here are aimed at clarifying the following aspects of the system: (a) The ability to express complex programs, their resource bounds and the specifications necessary for proving them, (b) the performance of the verification algorithm, (c) the accuracy of the inferred bound compared to their values obtainable at runtime and (d) the advantages of techniques presented here over other approaches that are potentially applicable on similar problems. All evaluations presented in this chapter were performed on a machine with a 4 core, 3.60 GHz, Intel Core i7 processor with 32GB RAM, running Ubuntu operating system.

## 5.1 First-Order Functional Programs and Data Structures

In this section, I summarize the results of using ORB to verify first-order functional programs. The resource verification of these programs uses techniques described in section 3. Figure 5.1 lists the benchmarks belonging to this class of programs that were verified using ORB. The figure lists the benchmarks and the lines of code in each benchmark in column *Loc*. The column *T* shows the number of functions in each benchmark with a resource template and the column *S* the number of specification functions in each benchmark. Specification functions are not verified for resource usage. They are only proven to be terminating using the termination checker of the LEON verification system [Nicolas Voirol and Kuncak, 2017]. The figure also shows a sample template for the steps resource for one or more functions in the benchmark. The benchmarks comprise approximately 1.8K lines of Scala code, 150 functions and 82 resource templates (for each resource considered). Below I explain the benchmarks in more detail.

### 5.1.1 Benchmark Descriptions

**List and Queues** The benchmark *list* implements a set of list manipulation operations like *append, reverse, remove, find* and *distinct* which removes duplicates. The quadratic resource template shown in the Figure 5.1 is for the function *distinct*. The benchmark *cvars* compares two different strategies for concatenating lists. One strategy exhibits a cubic behavior on a sequence of concatenation operations (templates shown in Figure 5.1) and the other exhibits

| ***Benchmark*** | ***LOC*** | ***T*** | ***S*** | ***Sample resource template*** steps ≤ |
|---|---|---|---|---|
| List Operations (*list*) | 60 | 7 | 1 | $?\cdot \mathsf{size}(\mathsf{l})^2+?$ |
| List Concatenations (*cvar*) | 40 | 4 | 1 | |
| *strategy 1* | | | | $?\cdot nm^2+?\cdot nm+?\cdot n+?\cdot m+?$ |
| *strategy 2* | | | | $?\cdot nm+?\cdot n+?\cdot m+?$ |
| Doubly ended queue (*deq*) | 86 | 6 | 0 | $?\cdot \mathsf{qsize}(\mathsf{q})+?$ |
| Quciksort, Insertion sort (*sort*) | 325 | 7 | 2 | $?\cdot \mathsf{size}(\mathsf{l})^2+?$ |
| Mergesort | | | | $?\cdot \mathsf{size}(\mathsf{l})\log\mathsf{size}(\mathsf{l})+?$ |
| Binary search tree (*bst*) | 91 | 5 | 3 | |
| *addAll* | | | | $?\cdot \mathsf{lsize}(\mathsf{l})\cdot(\mathsf{height}(\mathsf{t})+\mathsf{lsize}(\mathsf{l}))+?\cdot \mathsf{lsize}(\mathsf{l})+?$ |
| *removeAll* | | | | $?\cdot \mathsf{lsize}(\mathsf{l})\cdot \mathsf{height}(\mathsf{t})+?\cdot \mathsf{lsize}(\mathsf{l})+?$ |
| Binary Trie (*trie*) | 119 | 4 | 2 | $?\cdot \mathsf{inpsize}(\mathsf{inp})+?$ |
| Redblack tree (*rbt*) | 109 | 7 | 2 | $?\cdot \lfloor\log(\mathsf{size}(\mathsf{t})+1)\rfloor+?$ |
| AVL tree (*avl*) | 190 | 4 | 5 | $?\cdot \mathsf{height}(\mathsf{t})+?$ |
| Leftist heap (*lheap*) | 197 | 9 | 10 | |
| *merge* | | | | $?\cdot \mathsf{rheight}(\mathsf{h1})+?\cdot \mathsf{rheight}(\mathsf{h2})+?$ |
| *removeMax* | | | | $?\cdot \log(\mathsf{size}(\mathsf{h})+1)+?$ |
| *sort* | | | | $?\cdot \mathsf{size}(\mathsf{l})\log(\mathsf{size}(\mathsf{l})+1)+\mathsf{size}(\mathsf{l})+?$ |
| Binomial heap (*bheap*) | 204 | 5 | 5 | |
| *merge* | | | | $?\cdot \mathsf{treenum}(\mathsf{h1})+?\cdot \mathsf{treenum}(\mathsf{h2})+?$ |
| *deleteMin* | | | | $?\cdot \mathsf{treenum}(\mathsf{h1})+?\cdot \mathsf{minchildren}(\mathsf{h2})+?$ |
| Prop. logic transforms (*prop*) | 63 | 4 | 1 | $?\cdot \mathsf{size}(\mathsf{formula})+?$ |
| Loop transforms (*loop*) | 102 | 5 | 5 | $?\cdot \mathsf{size}(\mathsf{stmts})+?$ |
| Constant Propagation (*cprop*) | 294 | 10 | 5 | |
| *computeSummaries* | | | | $?\cdot \mathsf{psize}(\mathsf{p})\cdot \mathsf{iter}+?\cdot \mathsf{iter}+?$ |
| Speed benchmarks(*speed*) | 107 | 5 | 3 | $?\cdot(k+1)\cdot(\mathsf{len}(\mathsf{sb1})+\mathsf{len}(\mathsf{sb2}))+?\cdot \mathsf{size}(\mathsf{str1})+?$ |
| Fold operations (*fold*) | 88 | 4 | 2 | |
| *listfold* | | | | $?\cdot k^2+?$ |
| *treefold* | | | | $?\cdot \mathsf{size}(\mathsf{t})+?$ |

Figure 5.1 – Benchmarks implemented as first-order functional Scala programs

a quadratic behavior. The benchmark *deq* is a doubly-ended queue with *enqueue*, *dequeue*, *pop* and *concat* operations, implemented using two lists.

**Sorting** The benchmark *sort* contains the implementations of quick sort, insertion sort, merge sort. The tool was able to establish the precise running time bound of these algorithms. However, for quick sort and merge sort the bounds relied on non-trivial axioms of multiplication, which were manually provided as proofs hints to the system. In the case of quick sort, the axioms were also verified using the system. In the case of merge sort, the axioms weren't provable within the system due to the incompleteness in the nonlinear integer reasoning of the underlying SMT solvers. However, those axioms were verified independently using another verification engine.

**Search Trees** The benchmark *bst* implements a binary search tree with operations like *insert*, *remove*, *find*, *addall* and *removeall*. The last two functions add or remove a sequence of elements from the tree. The function lsize(l) used in the templates is the size of the sequence of elements to be inserted/removed. The benchmark *rbt* is an implementation of red-black tree with *insert* and *find* operations, and *avl* is an implementation of AVL tree with *insert, delete* and *find* operations. Our system establishes a logarithmic time bound for the insert function of the red-black tree. However for AVL tree the logarithmic time bound could not be expressed as it requires reasoning about logarithms to the base of an irrational number (namely golden ratio), which is outside the scope of the system. *trie* is an implementation of a binary trie with operations: *insert* – that inserts a sequence of input bits into the tree, *find* – that look up if the tree contains a sequence of bits, *create* –that creates a new tree from an input sequence and *delete*–that deletes a sequence of input bits from the tree. The function inpSize used in the template computes the size of the input list.

**Heaps** The benchmark *lheap* is a leftist heap data-structure implementation with *merge*, *insert* and *remove max* operations. It also additionally implemented a *sort* function that sorts an unsorted list using the heap sort algorithm. The time bounds in the benchmarks are specified in terms of the function rheight that computes the height of the right most leaf of the tree. However, the insert and removemax operations were specified a logarithmic bound on the size of the heap as shown. The sort function has an $n \cdot \log n$ time bound.

The benchmark *bheap* implements a binomial heap with *merge*, *insert* and *deletemin* operations. The functions treenum and minchildren (used in templates), compute the number of trees in a binomial heap and the number of children of the tree containing the minimum element, respectively. The logarithmic time bound is not established for this benchmark as it requires reasoning about a power series. However, the established bounds are strong enough to imply the logarithmic bound.

**AST Manipulations** The benchmark *prop* implements a set of propositional logic transformations like converting a formula to negation normal form and simplifying a formula. *loop* implements simple loop transformations like converting a for-loop to a while-loop using the abstract syntax tree of a program. The benchmark *cprop* implements a bottom-up summary-based, interprocedural constant propagation. The function psize used in the resource template of the function shown in Figure 5.1 denotes the size of a program, which is the sum of the sizes of the ASTs of the bodies of all the functions. iter denotes an iteration counter that upper bounds the number of iterations required for reaching the fixpoint and depends on the height of the lattice.

**Miscellaneous** The benchmark *speed* is a functional translation of the code snippets presented in Figures 1,2 and 9 of the related work *SPEED* [Gulwani et al., 2009]. It also includes

the code snippets on which it was mentioned that the tool failed (Page 138 of [Gulwani et al., 2009]). Our tool succeeded on those code snippets when suitable resource templates were provided manually. The benchmark *fold* is a collection of fold operations over trees and lists. These were mainly included for evaluation of *depth* bounds.

### 5.1.2 Analysis Results

| | ***Sample bound inferred*** <br> *steps*$\leq$ | ***Time (s)*** <br> ***(min.time)*** | ***Avg.VC size*** | ***disj.*** | ***NL size*** |
|---|---|---|---|---|---|
| *list* | $11\mathsf{size}(\mathsf{l})^2 + 3$ | 7 (2) | 79 | 58 | 29 |
| *cvar* | $7nm^2 - 11nm + 0*n + 0*m + 3$ <br> $12nm + 11m + n + 3$ | 24 (14) | 272 | 36 | 45 |
| *deq* | $7\mathsf{qsize}(\mathsf{q}) + 25$ | 4 (0.5) | 47 | 34 | 10 |
| *sort* | $11\mathsf{size}(\mathsf{l})^2 + 3$, $99\mathsf{size}(\mathsf{l})\log\mathsf{size}(\mathsf{l}) + 7$ | 10 (3) | 70 | 66 | 26 |
| *bst* | $10(\mathsf{lsize}(\mathsf{l}) \cdot (\mathsf{height}(\mathsf{t}) + \mathsf{lsize}(\mathsf{l}))) + 2\mathsf{lsize}(\mathsf{l}) + 2$ <br> $29(\mathsf{lsize}(\mathsf{l}) \cdot \mathsf{height}(\mathsf{t})) + 10\mathsf{lsize}(\mathsf{l}) + 1$ | 30 (11) | 116 | 69 | 123 |
| *trie* | $35\mathsf{inpsize}(\mathsf{inp}) + 8$ | 5 (0.5) | 43 | 35 | 16 |
| *rbt* | $132\lfloor\log(\mathsf{size}(\mathsf{t}) + 1))\rfloor + 66$ | 33 (7) | 352 | 99 | 154 |
| *avl* | $161\mathsf{height}(\mathsf{t}) + 137$ | 85 (41) | 181 | 68 | 58 |
| *lheap* | $35\mathsf{rheight}(\mathsf{h1}) + 35\mathsf{rheight}(\mathsf{h2}) + 2$ <br> $70\log(\mathsf{size}(\mathsf{h}) + 1) + 7$ <br> $169\mathsf{size}(\mathsf{l}) \cdot \log(\mathsf{size}(\mathsf{l}) + 1) + 9$ | 46 (24) | 292 | 90 | 104 |
| *bheap* | $40\mathsf{treenum}(\mathsf{h1}) + 46\mathsf{treenum}(\mathsf{h2}) + 2$ <br> $81\mathsf{treenum}(\mathsf{h1}) + 40\mathsf{minchildren}(\mathsf{h2}) + 24$ | 23 (2) | 277 | 67 | 70 |
| *prop* | $43\mathsf{size}(\mathsf{formula}) - 17$ | 11 (0.5) | 161 | 57 | 35 |
| *loop* | $16\mathsf{size}(\mathsf{program}) - 8$ | 36 (31) | 93 | 38 | 34 |
| *cprop* | $30\mathsf{psize}(\mathsf{p}) \cdot \mathsf{iter} + 16 \cdot \mathsf{iter} + 2$ | 29 (10) | 157 | 107 | 54 |
| *speed* | $37((k+1) \cdot (\mathsf{len}(\mathsf{sb1}) + \mathsf{len}(\mathsf{sb2}))) + 15\mathsf{size}(\mathsf{str1}) + 34$ | 93 (64) | 387 | 110 | 72 |
| *fold* | $13k^2 + 3$ <br> $14\mathsf{size}(\mathsf{t}) + 3$ | 10 (1) | 40 | 66 | 27 |

Figure 5.2 – Results of running ORB on the first-order benchmarks

Figure 5.2 shows the results of running our tool on the benchmarks. The column *bound* shows the steps bound inferred by the tool for the sample template shown in Figure 5.1. This may provide some insights into the constants that were inferred. The column *time* shows the total time taken for analysing a benchmark in seconds. In parentheses I show the time the

tool spent in minimizing the bounds after finding a valid initial bound. Recall that the tool performs a binary search over the space of possible solutions as described in section 3.6 to find strongest bounds. The tool spent at most 100 seconds to infer the bounds on each benchmark. Also, in case where it took more than 50 seconds to infer a bound about 50% of the time was spent on finding the strongest bound, which implies that initial bound was found in less than a minute on all benchmarks.

The subsequent columns provide more insights into the hole inference algorithm. The column *Avg. VC size* shows the average size of the VCs generated by the benchmarks when averaged over all VC refinements. The tool performed 11 to 42 VC refinements on the benchmarks. The column *disj.* shows the total number of disjuncts falsified by solveUNSAT procedure (section 3.3.4), and the column *NL size* shows the average size of the nonlinear constraints solved in each iteration of the solveUNSAT procedure.

Our tool is able to solve all 82 templates. The results also show that our tool was able to keep the average size of the VCs and the nonlinear Farkas' constraints generated small (see section 3.3), despite having to unfold the VCs several times. This is quite important since even the state-of-the-art nonlinear constraint solvers do not scale well to large nonlinear constraints. Furthermore, as shown by the column *disj.*, the solveUNSAT algorithm only explores a fraction of the total number of disjuncts in the VC (which is $O(2^n)$, where $n$ is the VC size).

**Testing Minimality of Constants using Counterexamples.** In order to test the minimality of the constants inferred, two kinds of experiments were performed. One experiment that scaled to only simpler benchmarks is explained here. The other experiment based on runtime evaluations is detailed in the context of higher-order, memoized benchmarks discussed in the following section. In the first experiment, for each bound inferred by the tool, one coefficient of the bound was decremented while keeping the others fixed. The system was used to check if the bounds thus constructed were valid. Since these bounds do not have holes, the system, in principle, is capable of discovering a counterexample, which is an input that violates the bound, if the bounds are not valid. If a counterexample is found, it implies that there cannot exist a valid bound where all constants are smaller than the inferred bound. In other words the bound inferred is pareto optimal.

Our experiments showed that in many cases the bound inferred by the tool was pareto optimal. For instance, this was the case with benchmarks *list*, *deq* and *trie*. Notably, in the case of *trie*, the counterexample emitted by the tool had an input character sequence of size 50. However, for more complex benchmarks like red-black tree, it was not possible to discover counterexamples statically, due to the complexity of the inputs that induce worst-case behavior, and also due to the large constants in the bounds. To estimate the accuracy of the bound inferred by the tool in such cases, runtime profiling of the benchmarks were performed over many inputs that enforce the worst case behavior. These experiments are discussed in more detail in the following section in the context of more complex higher-order, memoized benchmarks.

| | ***Inferred depth bound:*** depth≤ |
|---|---|
| *msort** | $45\mathsf{size}(\mathsf{l}) + 1$ |
| *qsort* | $7\mathsf{size}(\mathsf{l})^2 + 5\mathsf{size}(\mathsf{l}) + 1$ |
| *trie* | $8\mathsf{inpsize}(\mathsf{inp}) + 1$ |
| *rbt* | $22\mathsf{blackHeight}(\mathsf{t}) + 19$ |
| *avl* | $51\mathsf{height}(\mathsf{t}) + 4$ |
| *bheap* | $7\mathsf{treenum}(\mathsf{h1}) + 7\mathsf{treenum}(\mathsf{h2}) + 2$ |
| *prop** | $5\mathsf{nestingDepth}(\mathsf{formula}) - 2$ |
| *fold** | $6k + 1$<br>$5\mathsf{height}(\mathsf{tree}) + 1$ |

Figure 5.3 – Results of inferring bounds on depths of benchmarks

| | ***Inferred alloc bound:*** alloc≤ |
|---|---|
| *list* | $\mathsf{size}(\mathsf{l})^2 + 1$ |
| *bst* | $2(\mathsf{lsize}(\mathsf{l}) \cdot \mathsf{height}(\mathsf{t})) + \mathsf{lsize}(\mathsf{l}) + 1$ |
| *trie* | $4\mathsf{inpsize}(\mathsf{inp})$ |
| *rbt* | $9\lfloor \log(\mathsf{size}(\mathsf{t}) + 1)) \rfloor + 8$ |
| *bheap* | $2\mathsf{treenum}(\mathsf{h1}) + 2\mathsf{treenum}(\mathsf{h2}) + 1$ |
| *prop* | $3\mathsf{size}(\mathsf{formula}) - 3$ |

Figure 5.4 – Results of inferring bounds on the number of heap-allocated objects

Specifically, for the red-black tree benchmark these experiments showed that the (worst-case) runtime steps count was 86% of the value statically inferred by the tool.

**Inference of Depth Bounds** The tool was used to infer bounds on the depth usage of the benchmarks (see section 3.1.1 for the definition of depth). In this case, the tool is able to solve all the templates provided in the benchmarks including the merge sort and quick sort programs. The constants in the depth bounds are much smaller for every benchmark compared to steps, even if depth is not asymptotically smaller than steps. Figure 5.3 shows the inferred depth bounds for selected benchmarks. The benchmarks where the depth bound is asymptotically smaller than the corresponding steps bound are starred. Specifically, the tool is able to establish that the depth of *mergesort* is linear in the size of its input, the depth of negation normal form transformation is proportional to the nesting depth of its input formula, and also that the depth of fold operations on trees is linear in the height of the tree.

| | ***Inferred call-stack bound:*** stack≤ |
|---|---|
| *list** | $17\mathsf{size}(\mathsf{l}) + 18$ |
| *cvar** | $18nm + n + 14 * m + 19$<br>$nm + 15m + 17n + 19$ |
| *isort** | $17\mathsf{size}(\mathsf{l}) + 16$ |
| *msort** | $28\mathsf{size}(\mathsf{l}) + 52$ |
| *qsort** | $26\mathsf{size}(\mathsf{l}) + 29$ |
| *bst** | $21\mathsf{height}(\mathsf{t}) + 39$<br>$38\mathsf{height}(\mathsf{t}) + \mathsf{lsize}(\mathsf{l}) + 56$ |
| *rbt* | $54\lfloor\log(\mathsf{size}(\mathsf{t}) + 1))\rfloor + 100$ |

Figure 5.5 – Results of inferring bounds on the call-stack usage

**Bounds on Heap-Allocated Objects** Figure 5.4 shows the results of verifying the bounds on the usage of the alloc resource on selected benchmarks. The alloc resource measures the number of heap-allocated objects. Since our benchmarks are first order, the heap-allocated objects comprises only datatype instance. As shown in the figure, the constants in this case are rather small compared to steps bounds, and in many cases one. However, the alloc bound asymptotically matches the steps bound in most cases. This is because the benchmarks are mostly functional data structures, whose operations like insert or delete would have to replicate the entire portion of the data structure that is traversed by the operation. Hence, the heap-allocation is proportional to the work done i.e, steps.

**Bounds on the Call-Stack Usage** Figure 5.5 shows the results of verifying the call-stack usage bounds on selected benchmarks. (This resource was explained in section 2.3). The call-stack usage is measured in units of 64 bit words. The instrumentation for call-stack usage also takes into account tail-call optimizations. It considers that the tail calls would be optimized away and do not contribute to the stack space usage. This optimization is performed by default by the Scala compiler.

The benchmarks that are starred in the Figure 5.5 indicate cases where the stack space usage was asymptotically smaller than steps usage. Note that this is the case especially for functions whose steps usage is quadratic or higher e.g. for *qsort*, the *addAll* function of *bst*, or the *distinct* function of *list*. The only benchmark where stack space usage is quadratic is in the benchmark *cvar*. Even here the complexity dropped from $O(nm^2)$ to $O(nm)$ in one of the resource bounds.

### 5.1.3 Comparison with CEGIS and CEGAR

Here, I summarize the results of comparing the inference algorithm used in ORB with *Counterexample Guided Inductive Synthesis*(CEGIS) [Solar-Lezama et al., 2006] which, to our knowl-

edge, is the only existing approach that could potentially be used to solve a $\exists\forall$ formula with ADTs, recursive (or uninterpreted) functions and nonlinear operations. CEGIS is an iterative algorithm that, given a formula $\phi$ with holes and other variables $\bar{x}$, makes progress by finding a solution for the holes that rules out at least one satisfying assignment for $\bar{x}$ that was feasible in the earlier iterations. This is in contrast to the solveUNSAT algorithm which makes progress by finding a solution for the holes that falsifies *disjuncts* of $\phi$ that was satisfiable in the earlier iterations. This can be seen as ruling out an infinite set of statisfying assignments to $\bar{x}$ in each iteration. Furthermore, our approach is guaranteed to terminate but CEGIS, in theory, may diverge if the possible values for $\bar{x}$ is infinite.

To compare our algorithm with CEGIS, CEGIS was implemented within our system, and evaluated on the benchmarks described in this section. The results showed that CEGIS diverges even on the simplest of the benchmarks. It follows an infinite ascending chain along which the parameter corresponding to the constant term of the template increases indefinitely. CEGIS was also evaluated by bounding the values of the parameters to be $\leq 200$. In this case, CEGIS worked on 5 small benchmarks (viz. *list*, *bst*, *deq*, *trie* and *fold*) but timed out on the rest after 30min. For the benchmarks on which it worked, it was 2.5 times to 64 times slower than ORB.

**CEGAR** Another closely related technique for inferring invariants for recursive functions using user-provided templates is *Counterexample Guided Abstraction Refinement* (CEGAR). CEGAR is also an iterative algorithm. In every iteration, CEGAR computes an abstraction of the input program, and searches for a counterexample path in the abstract program that violates the given property. If a counterexample is found, the abstraction is refined so that it (and also other related counterexample paths) are not feasible in the refined abstraction. Typically, the refinement is constructed by computing an *interpolant* that provides a succinct condition for the infeasibilty of the concrete path that corresponds to the abstract counterexample path in the original program. The concrete path represents a finite execution and typically goes through recursive calls a finite number of times (unlike a static path). Tools such as HSF [Beyene et al., 2013] can compute interpolants belonging to a given template. However, there are not many off-the-shelf tools that can perform interpolation in the presence of ADTs, recursive (or uninterpreted) functions and nonlinear operations. Nonetheless, interpolation also suffers from similar issues as CEGIS, since for any finite execution, the resources consumed by the execution is a constant. This suggest that, in theory, interpolation-based CEGAR can always come up with increasing values for the constant term of a template, which would provide a valid bound for the concrete paths explored until a particular point, but could never provide a bound that holds for all paths (much like CEGIS).

| ***Benchmark*** | ***LOC*** | ***BC*** | ***T*** | ***S*** |
|---|---|---|---|---|
| Lazy Selection Sort (*sel*) | 70 | 36kb | 4 | 1 |
| Prime Stream (*prims*) | 95 | 51kb | 7 | 2 |
| Fibonacci Stream (*fibs*) [Bird and Wadler, 1988] | 199 | 59kb | 5 | 5 |
| Hamming Stream (*hams*) [Bird and Wadler, 1988] | 223 | 78kb | 8 | 6 |
| Stream library (*slib*) [Swierstra, 2015] | 408 | 0.1mb | 22 | 5 |
| Lazy Mergesort (*msort*) [Apfelmus, 2009] | 290 | 0.1mb | 6 | 8 |
| Real time queue (*rtq*) [Okasaki, 1995, 1998] | 207 | 69kb | 5 | 6 |
| Deque (*deq*) [Okasaki, 1995, 1998] | 426 | 0.1mb | 16 | 7 |
| Lazy Numerical Rep.(*num*)[Okasaki, 1998] | 546 | 0.1mb | 6 | 25 |
| Conqueue (*conq*) [Prokopec and Odersky, 2015] | 880 | 0.2mb | 12 | 33 |

Figure 5.6 – Higher-order, lazy benchmarks comprising 4.5K lines of Scala code

| ***Benchmark*** | ***AT*** | steps ≤ ***Resource bounds*** alloc ≤ | |
|---|---|---|---|
| *sel* | 1m | $15k \cdot l.size + 8k + 13$ | $2k \cdot l.size + 2k + 2$ |
| *prims* | 1m | $16n^2 + 28$ | $6n - 11$ |
| *fibs* | 2m | $45n + 4$ | $4n$ |
| *hams* | 1m | $129n + 4$ | $16n$ |
| *slib* | 1m | $25l.size + 6$ | $3l.size$ |
| *msort* | 1m | $36k\lfloor \log l.size \rfloor + 53l.size + 22$ | $6k\lfloor \log l.size \rfloor + 6l.size + 3$ |
| *rtq* | 1m | 40 | 7 |
| *deq* | 5m | 893 | 78 |
| *num* | 1m | 106 | 15 |
| *conq* | 5m | | |
| *pushLeftAndPay* | | 124 | 23 |
| *concatNonEmpty* | | $29\lvert xs.lvl - ys.lvl \rvert + 8$ | $2\lvert xs.lvl - ys.lvl \rvert + 1$ |

Figure 5.7 – Steps and Alloc bounds inferred by ORB for higher-order, lazy benchmarks

## 5.2 Higher-Order and Lazy Data Structures

In this section, I summarize the results of using ORB to verify resource usage of higher-order programs and lazy data structures. Figure 5.6 shows selected benchmarks that were verified by our approach. Each benchmark was implemented and specified in a purely functional subset of Scala extended with the specification constructs detailed in section 4. The benchmarks were carefully chosen from some of the most challenging benchmarks proposed in the literature of lazy data-structures. For instance, the benchmark *rtq* has been mentioned as being outside the reach of prior works (section Limitations of [Danielsson, 2008]). The benchmarks *deq* [Okasaki, 1995], *conq* [Prokopec, 2014] are much more complicated than *rtq*. For each benchmark, the figure shows the total lines of Scala code and the size of the compiled JVM byte code in columns *LOC* and *BC*. The benchmarks comprise a total of 4.5K lines of Scala code and 1.2MB of bytecodes. The column *T* shows the number of functions with resource bound templates, and the column *S* the number of specification functions. The benchmarks had a total of 123 resource templates each for steps and alloc resource. The system was able to solve all

| *B* | *I* | (*dynamic/static) * 100* | | *(optimal/static) * 100* | |
|---|---|---|---|---|---|
| | | ***steps*** | ***alloc*** | ***steps*** | ***alloc*** |
| *sel* | 10k | 99 | 99 | 100 | 100 |
| *prims* | 1k | 60 | 89 | 82 | 100 |
| *fibs* | 10k | 99 | 99 | 100 | 100 |
| *hams* | 10k | 86 | 83 | 98 | 100 |
| *slib* | 10k | 65 | 75 | 85 | 88 |
| *rtq* | $2^{20}$ | 93 | 83 | 97 | 87 |
| *msort* | 10k | 90 | 91 | 96 | 97 |
| *deq* | $2^{20}$ | 48 | 48 | 59 | 62 |
| *num* | $2^{20}$ | 94 | 97 | 96 | 100 |
| *conq* | $2^{20}$ | 72 | 54 | 82 | 72 |
| *Avg.* | | 81 | 82 | 90 | 91 |

Figure 5.8 – Comparison of the resource usage bounds inferred statically against runtime resource usage

the resource templates by inferring constants that yield valid upper bounds on the specified resource.

Figure 5.7 shows the bounds inferred by ORB on these benchmarks. The column *AT* shows the time taken by our system rounded off to minutes to verify the specifications and infer the constants. As shown by the figure, all benchmarks were verified within a few minutes. (These benchmarks take longer than first-order benchmarks presented earlier because of the complexity in modeling the higher-order and lazy evaluation features accurately. ) The column *Resource bounds* shows a sample bound for steps and alloc resource. The constants in the bound were automatically inferred by the tool. As shown in the figure, many bounds use recursive functions, and almost 20 bounds have nonlinear operations. As explained in section 3.5, nonlinear operations like $\lfloor \log \rfloor$ are expressed as a recursive function that uses integer division: $\log(x) =$ **if**$(x \geq 2)\log(x/2) + 1$ **else** (base cases). Their properties like monotonicity are proven and instantiated manually. A few bounds were disjunctive (like the bound shown in Figure 1.3, and *conq*). However, the most challenging bounds to prove were the constant time bounds of scheduling-based lazy data structures viz. *rtq*, *deq*, *num*, and *conq* due to their complexity. Before I describe the results and benchmarks in detail, below I provide an overview of the experiments that were performed to gauge the accuracy of the inferred bounds.

### 5.2.1 Measuring Accuracy of the Inferred Bounds

While the resource instrumentation in the case of first-order programs is quite straightforward, this is not the case for higher-order programs with memoization. Recall that the model generation phases perform numerous transformations to express the input programs in a form that is amenable to verification. Furthermore, the inference algorithm itself could result in loss of completeness, especially on the model programs due to their complexity. A natural

question that arises in this case is whether the bounds inferred by the tool on programs that use such high-level language features have any correspondence at all to their real resource usage at runtime. In other words, will the constants provide any more information about the execution apart from serving to establish an asympototic complexity? The experiments described in this section were designed to answer these questions.

Each benchmark presented in Figure 5.6 were instrumented to track steps and alloc resources as defined by the operational semantics. The benchmarks were then executed on concrete inputs that were likely to expose their worst case behavior (but not necessarily since it is difficult to determine the worst case input for some benchmarks). The sizes of the inputs were varied in fixed intervals upto 10k for most benchmarks. However, for those benchmarks with nonlinear behavior smaller inputs that scaled within a cutoff time of 5 min were used, as tabulated in the column *I* of Figure 5.8. For scheduling based data structures (discussed shortly) the input were varied in powers of two until $2^{20}$, which results in their worst-case behavior. Using large inputs has the advantage that it places large emphasis on the precision of coefficient of the fastest growing terms.For every top-level (externally accessible) function in a benchmark, the mean ratio between the runtime resource usage and the static resource usage predicted by our tool was computed using the following formula:

$$Mean\left(\frac{\text{resource consumed by the } i^{th} \text{ input}}{\text{static estimate for } i^{th} \text{ input}} \times 100\right)$$

The column *dynamic/static * 100* of Figure 5.8 shows this metric for each benchmark when averaged over *all* top-level functions in the benchmark. This metric is a measure of the worst-case runtime resource usage as a percentage of the static estimate. As shown in the figure, when averaged across all benchmarks, the runtime resource usage is 81% of what was inferred statically for steps, and is 82% for alloc. In all cases, the inferred bounds are sound upper bounds for the runtime resource usage. I now discuss the reasons for some of the inaccuracy in the inferred bounds.

In our system, there are two factors that influence the overall accuracy of the bound: (a) the constants inferred by tool, and (b) the resource templates provided by the user. For instance, in the *prims* benchmark shown in Figure 1.3 of the introduction, the function isPrimeNum(n) has a worst-case steps count of $11i - 7$, which will be reached only if $i$ is prime. (It varies between $O(\sqrt{i})$ and $O(i)$ otherwise.) Hence, for the function primesUntil(n), which transitively invokes isPrimeNum function on all numbers until $n$, no solution for the template: $? * n^2 + ?$ can accurately match its worst-case, runtime steps count. Another example is the $O(k \cdot \lfloor \log(l.size) \rfloor)$ resource bound of *msort* benchmark. In any actual run, as $k$ increases the size of the stream that is accessed (which is initially $l$) decreases. Hence, $\lfloor \log(l.size) \rfloor$ term decreases in steps.

To provide more insights into the contribution of each of these factors to the inaccuracy, the following experiment was performed. For each function, each constant in its resource bound was reduced, keeping the other constants fixed, until the bound violated the resources usage

of at least one dynamic run. I call such a bound as a *pareto optimal* bound with respect to the dynamic runs. Note that if there are $n$ constants in the resource bound of a function, there would be $n$ pareto optimal bounds for the function. To estimate the optimality of the constants inferred by the tool, the pareto optimal bounds are used as the baseline, instead of the runtime resource usage. The mean ratio between the resource usage predicted by the pareto optimal bound and that predicted by the bound inferred by the tool was measured for each benchmark.

The column *optimal/static * 100* of Figure 5.8 shows this metric for each benchmark when averaged over all pareto optimal bounds of all top-level functions in the benchmark. A high percentage for this metric is an indication that any inaccuracy is due to imprecise templates, whereas a low percentage indicates a possible incompleteness in the resource inference algorithm, which is often due to nonlinearity or absence of sufficiently strong invariants. As shown in Figure 5.8, the constants inferred by the tool were 91% accurate for steps and 94% accurate for alloc, when compared to the pareto optimal values that fits the runtime data. Furthermore, the imprecision due to templates is a primary contributor for inaccuracy, especially in benchmarks where the accuracy is lower than 80% (such *Viterbi* and *prims*). In the sequel, I discuss the benchmarks and the results of their evaluation in more detail.

### 5.2.2 Scheduling-based Lazy Data Structures

One of the most challenging class of benchmarks considered in our evaluation are the scheduling-based lazy data structures proposed by Okasaki [1998]. The benchmarks *rtq*, *deq*, *num*, and *conq* belong to this class. These data structures use lazy evaluation to implement worst-case, constant time as well as *persistent* queues and deques using a strategy called scheduling. These are one of the most efficient persistent data structures. For instance, the *rtq* [Okasaki, 1995] benchmark takes a few nanoseconds to persistently enqueue an element into a queue of size $2^{30}$. The *conq* data structure [Prokopec and Odersky, 2015] is used to implement data-parallel operations provided by the standard Scala library. To my knowledge there exists no prior approach that proves the resource bounds of these benchmarks. Moreover, the verification of these benchmarks also led to the discovery and fixing of a missing corner case of the rotateDrop function shown in Fig 8.4 of [Okasaki, 1998], which was unraveled by the system.

Though the data structures differ significantly in their internal representation, invariants, resource usage and the operations they support, fundamentally they consists of streams called *spines* that track content, and a list of references to closures nested deep within the spines: *schedules*. The schedules help materialize the data structure lazily as they are used by a client. I now provide a brief overview of the kind of specifications that were required to verify the resource bounds of these benchmarks using the example of real-time queue.

```
1  object RealTimeQueue {
2  sealed abstract class Stream[T] {
3    def isEmpty: Boolean = this == SNil[T]()
4
5    lazy val tail: Stream[T] = {
6      require(!isEmpty)
7      this match {
8          case SCons(x, tailFun) ⇒ tailFun()
9      }
10   }
11
12   def size: BigInt = {
13     this match {
14       case SNil() ⇒ BigInt(0)
15       case c@SCons(_, _) ⇒ 1 + (c.tail*).size
16     }
17   } ensuring (_ ≥ 0)
18 }
19 private case class SCons[T](x: T, tailFun: () ⇒ Stream[T]) extends Stream[T]
20 private case class SNil[T]() extends Stream[T]
21
22 /**
23  * A property that holds for stream where all elements have been memoized.
24  */
25 def isConcrete[T](l: Stream[T]): Boolean = {
26   require(l.valid)
27   l match {
28     case c @ SCons(_, _) ⇒
29       cached(c.tail) && isConcrete(c.tail*)
30     case _ ⇒ true
31   }
32 }
33
34 /**
35  * A function that lazily performs an operation equivalent to
36  * 'f ++ reverse(r) ++ a'. Based on the implementation
37  * in Pg.88 of Functional Data Structures by Okasaki [Okasaki, 1998].
38  */
39 def rotate[T](f: Stream[T], r: List[T], a: Stream[T]): Stream[T] = {
40     require(r.size == f.size + 1 && isConcrete(f))
41     (f, r) match {
42       case (SNil(), Cons(y, _)) ⇒ SCons[T](y, lift(a))
43       case (c@SCons(x, _), Cons(y, r1)) ⇒
44         val newa = SCons[T](y, lift(a))
45         val ftail = c.tail
46         val rot = () ⇒ rotate(ftail, r1, newa)
47         SCons[T](x, rot)
48     }
49 } ensuring (res ⇒ res.size == f.size + r.size + a.size && res.!isEmpty && steps ≤ ?)
```

Figure 5.9 – Rotate function of the Real-time queue data structure

```
72  /**
73  * Returns the first element of the stream whose tail is not memoized.
74  */
75  def firstUnevaluated[T](l: Stream[T]): Stream[T] =
76    l match {
77      case c @ SCons(_, _) ⇒
78        if (cached(c.tail))
79          firstUnevaluated(c.tail*)
80        else l
81      case _ ⇒ l
82    }
83  } ensuring (res ⇒
84    //(b) no lazy closures implies stream is concrete
85    (!res.isEmpty || isConcrete(l)) &&
86    //(c) after evaluating the firstUneval closure the next can be accessed
87    (res match {
88      case c @ SCons(_, _) ⇒ firstUneval(l) == firstUneval(c.tail)
89      case _ ⇒ true
90    }))
91
92  case class Queue[T](f: Stream[T], r: List[T], s: Stream[T]) {
93    def isEmpty = f.isEmpty
94    def valid = {
95      f.valid && s.valid &&
96      // invariant: firstUneval of 'f' and 's' are the same.
97      (firstUneval(f) == firstUneval(s)) &&
98      s.size == f.size - r.size // invariant: |s| = |f| - |r|
99    }
100 }
101 /**
102  * A helper function for enqueue and dequeue methods that forces the schedule once
103  */
104 @inline
105 def createQ[T](f: Stream[T], r: List[T], s: Stream[T]) = {
106   s match {
107     case c @ SCons(_, _) ⇒ Queue(f, r, c.tail) // force
108     case SNil() ⇒
109       val rotres = rotate(f, r, SNil[T]())
110       Queue(rotres, Nil(), rotres)
111   }
112 }
```

Figure 5.10 – Definition of Okasaki's Real-time queue data structure

```
142  /**
143   * Appends an element to the end of the queue
144   */
145  def enqueue[T](x: T, q: Queue[T]): Queue[T] = {
146    require(q.valid)
147    createQ(q.f, Cons(x, q.r), q.s)
148  } ensuring { res ⇒
149    funeMonotone(q.f, q.s, inSt[T], outSt[T]) &&
150      res.valid && steps ≤ ?
151  }
152  /**
153   * Removes the element at the beginning of the queue
154   */
155  def dequeue[T](q: Queue[T]): Queue[T] = {
156    require(!q.isEmpty && q.valid)
157    q.f match {
158      case c @ SCons(x, _) ⇒
159        createQ(c.tail, q.r, q.s)
160    }
161  } ensuring { res ⇒
162    funeMonotone(q.f, q.s, inSt[T], outSt[T]) && res.valid && steps ≤ ?
163  }
164
165  // Properties of 'firstUneval'.
166  /**
167   * st1.subsetOf(st2) ⇒ fune(l, st2) = fune(fune(l, st1), st2)
168   */
169  @traceInduct
170  def funeCompose[T](l1: Stream[T], st1: Set[Fun[T]], st2: Set[Fun[T]]): Boolean = {
171    require(st1.subsetOf(st2) && l1.valid)
172    (firstUneval(l1) in st2) == (firstUneval(firstUneval(l1) in st1) in st2)
173  } holds
174
175  /**
176   * Monotonicity of 'firstUneval' with respect to the cache state.
177   */
178   def funeMonotone[T](l1: Stream[T], l2: Stream[T], st1: Set[Fun[T]], st2: Set[Fun[T]])
        = {
179     require((firstUneval(l1) in st1) == (firstUneval(l2) in st1)
180       && st1.subsetOf(st2))
181         funeCompose(l1, st1, st2) && funeCompose(l2, st1, st2) &&
182           (firstUneval(l1) in st2) == (firstUneval(l2) in st2)
183   } holds
184  }
```

Figure 5.11 – Queue operations of Okasaki's Real-time queue data structure

**Verifying Real-Time Queue Benchmark**

Figures 5.9,5.10 and 5.11 shows a complete implementation of the Okasaki's *Real-time queue* data structure [Okasaki, 1995, 1998] in our syntax. It uses the datatype Stream and the top of Figure 5.9 shows the definition of a lazy stream in our syntax. Akin to a list, the Stream datatype is defined using two constructors SCons and SNil denoting non-empty and empty streams, respectively. But, the second argument of SCons, denoting the tail of the stream, is a lazy reference. Consider the function rotate in Figure 5.9. It reverses the list r and appends it to the lazy stream f, using the stream a as a temporary storage, which is initially set to empty. Essentially, rotate(f,r,a)= f ++ reverse(r)++ a. (f and r actually represent the front and rear parts of the real-time queue data structure). However, the function performs its work lazily: every call to rotate constructs the first element of the result, and returns a stream whose tail is a suspended recursive call.

The specifications of rotate assert properties of the function that hold before and after the execution. Consider the property on the sizes of the arguments. This property is independent of the cache state i.e, it does not depend on whether the closures in the input list are forced or not. In contrast, the property isConcrete is cache state dependent: it returns true if every node of the argument stream has been forced, false otherwise. Notice that the postcondition also asserts a constant time bound for the function rotate. The requirement that isConcrete(f) holds at the beginning of the function is crucial for proving the time bound. Otherwise, forcing f at line 45 may invoke a previously suspended call to rotate, thus resulting in a cascade of forces.

As shown in Figure 5.10, the real time queue data structure has three components: a lazy stream f denoting the front of the queue, a list r denoting the rear of the queue, and a lazy stream s denoting the schedule. We define the data structure invariants using the boolean-valued function valid. Every public queue operation, namely enqueue and dequeue, require that the valid property holds for the input queue, and also ensures that the property holds for the output queue (see the definitions of the functions in Fig. 5.11).

Consider the property firstUneval(f)== firstUneval(s) that relates the schedule and the front streams that is a part of the definition of valid. The definition of firstUneval is shown in Fig. 5.11. It returns the first node in the stream that has not been forced. This property states that the unevaluated nodes of f and s are equal. In addition to this, the data structure also maintains the invariant that the size of the front is greater than the size rear, and that the size of the schedule is equal to the difference between the sizes of the front and the rear. These are succinctly captured by the second predicate of the function valid. The specification of the firstUneval function asserts a few interesting properties of the function that are needed for verification.

The data structure uses the same idea as a simple immutable queue that uses two lists, namely front and rear, that has a constant, amortized running time for *ephemeral* (i.e., non-persistent) usage. The elements are enqueued to the rear list and dequeued from the front list. Once in a while, when there are very few or no elements in the front list, the dequeue operation would

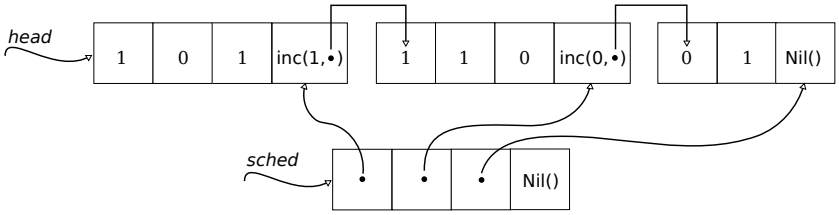



Figure 5.12 – Invariants of *conqueue* data structure [Prokopec and Odersky, 2015]

reverse the rear and append it to the front. This is captured by the rotate function of Fig. 5.11. The real time queue data structure uses a similar strategy, but it exploits lazy evaluation to perform the costly rotate operation incrementally, alongside the enqueue and dequeue operations. It thus achieves constant running time, in the worst case, for all operations even under persistent usage. For this purpose, it augments the queue with a *schedule* which is a reference to a closure that corresponds to the next step of an unfinished rotate operation. The rotate operation itself is performed lazily: every call to rotate constructs the first element of the result, and returns a stream whose tail is a suspended recursive call.

During every enqueue and dequeue operation, if the schedule is non-empty, the head of the schedule is forced (line 106 of the function createQ). This corresponds to performing one step of the rotate operation. On the other hand, if the schedule is empty, which implies that there are no pending rotate operations, a new rotate operation is initiated (lines 108 to 109). Hence, whenever a rotate operation is initiated every node of the argument f is forced. This is asserted by the isConcrete(f) predicate used in the precondition of the rotate function, which is critical for proving the O(1) time bound of rotate. Our system verifies the complete program shown in Fig. 5.11.

**Other Scheduling-based Data Structures** Similar to *rtq*, other scheduling-based data structures also consist of a *spine* that store the content, which corresponds to the front and rear streams in the case of *rtq*, and a *schedule*, which is a *list* of references to closures possibly nested deep within the spine. (In the case of *rtq* the schedule is a single reference.) The content of the spine can be other data structures. In the case of *conq*, the content is a AVL-like balanced tree called *ConcTree* [Prokopec and Odersky, 2015]. The schedules correspond to unfinished operations like enqueue initiated previously. Every operation on the data structure is performed lazily in increments that complete in a constant number of steps. Whenever a new operation is initiated, the schedules are forced so that an increment of a previous operation is performed. A complex invariant ensures that the pending operations do not cascade to result in non-constant time worst-case behavior. Figure 5.12 pictorially depicts the invariants of the *conqueue* data structure. In the figure, 1 or 0 represents a conc-tree – 1 at a position $i$ represents a conc-tree of size $2^i$ and 0 represents a conc-tree of size 1. The symbol inc denotes a suspension of a *pushLeft* function that takes two arguments: a conc-tree and (a reference to) another conqueue. If the suspension is forced it will prepend the conc-tree to the beginning of the queue and adjusts the data structure if necessary. Notice that unlike *rtq* here the schedules

are a list of references to closures.

**Results of Verifying Scheduling-based Data Structures** Figure 5.7 show the resource bounds inferred by the system for these data structures for the resources step and alloc. Notice that for the *deq* data structure the constants are as large as 893. Figure 5.8 shows the accuracy of the inferred bound. As the results in Figure 5.8 show, the inferred bounds were at least 83% accurate for *rtq* and *num* benchmarks, but have low accuracy for *deq* and *conq* benchmarks. On further analysis of *deq* we found that the bounds inferred by our system for the inner functions of *deq* were, in fact, 90% accurate in estimating the worst-case usage for the dynamic runs. But the worst-case manifested only occasionally (about once in four calls) when invoked from the top-level functions. The low accuracy seems to result from the lack of sufficient invariants for the top-level functions that prohibit the calls to inner functions from consistently exhibiting worst-case behavior. This seems to be due to a complex dependency between the functions, which was not identified due to insufficient unfoldings and SMT solver timeouts.

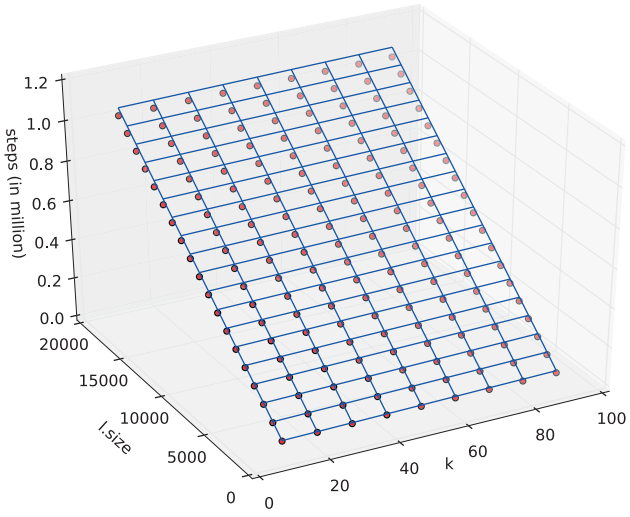
### 5.2.3 Other Lazy Benchmarks

**Cyclic Streams** The benchmarks *fibs* and *hams* implement infinite fibonacci and hamming sequences as cyclic streams using lazy *zipWith* and *merge* functions. Their implementations are based on the related work of Vasconcelos et al. [2015]. In comparison to their work in which the alloc bounds computed for *hams* were 64% accurate for inputs smaller than 10, ORB was able to infer bounds that were 83% accurate for inputs up to 10K.

**Stream Library** The benchmark *slib* is a collection of operations over streams such as *map*, *scan*, *cycle* etc. The operations were chosen from the Haskell stream library [Swierstra, 2015]. We excluded functions such as *filter* that can potentially diverge on infinite streams. The bounds presented are for a specific client of the library.

**Lazy Sorts** The benchmarks *msort* and *sel* implement lazy sorted streams that allows accessing the $k^{th}$ minimum an assorted list without performing the entire sorting. *sel* uses a selection sorting algorithm in which the minimum element is brought to the beginning of the list. When performed lazily, to the access the $k^{th}$ minimum only $k$ min operation would be performed.

The benchmark *msort* performs a bottom-up merge sort algorithm lazily [Apfelmus, 2009]. It creates a logical tree of closures of the merge function in steps linear in the size of the input list. The tree is complete and balanced. The elements of the tree can be retrieved in sorted order, and every access traverses a path from the root to the tree, and hence happens in steps logarithmic in the size of the list. Thus, the steps count for accessing the $k^{th}$ element (which is the $k^{th}$ minimum) is $O(k\lfloor \log l.size \rfloor)$. The bound inferred by the system as shown in Figure 5.7




$\text{steps} \leq 36k\lfloor \log l.size \rfloor + 53l.size + 22$

Figure 5.13 – Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for lazy merge sort

is $36k\lfloor \log l.size \rfloor + 53l.size + 22$. Figure 5.8 shows the actual runtime resource usage is at least 90% of the value estimated by the inferred bounds and is close 96% accurate when comparing against pareto optimal bounds. Figure 5.13 plots the relationship between the inferred bound and the resource usage at runtime for the resource steps. The lines in the figure show how the inferred bound changes as $l.size$ and $k$ is varied, and the dots show how the runtime resource usage changes with $l.size$ and $k$. The constants inferred are fairly accurate despite the complexity of the resource template.

## 5.3 Memoized Algortihms

Our system was used to verify the resource bounds of dynamic programming algorithms [Cormen et al., 2001, Dasgupta et al., 2008] shown in Figure 5.14 that were expressed as memoized recursive functions. The benchmarks *lcs* and *levd* implement the algorithms for finding the longest common subsequence and Levenshtein distance between two strings (represented as integer arrays), respectively. The benchmark *ks* implements the knapsack algorithm for packing a list of items, each of value $v_i$ and weight $w_i$, into a knapsack of capacity $w$ in a way that maximizes the total value of the items in the knapsack. *hs* is a memoized version of the hamming stream benchmark that computes a sorted list of numbers of the form $2^i 3^j 5^k$.

The benchmark *ws* is a weighted scheduling algorithm that optimally schedules $n$ jobs with (overlapping) start and finish times so that the total value of the scheduled jobs is maximized.

| *Benchmark* | *LOC* | *T* | *S* | *AT* | *Resource bounds* |
|---|---|---|---|---|---|
| LCS (*lcs*) | 121 | 4 | 4 | 1m | steps $\leq 30mn+30m+30n+28$<br>alloc $\leq 2mn+2m+2n+3$ |
| Levenshtein (*levd*) Distance | 110 | 4 | 4 | 1m | steps $\leq 36mn+36m+36n+34$<br>alloc $\leq 2mn+2m+2+3$ |
| Hamming (*hm*) Numbers | 105 | 3 | 3 | 3m | steps $\leq 66n+65$<br>alloc $\leq 3n+4$ |
| Weight Scheduling (*ws*) | 133 | 3 | 5 | 1m | steps $\leq 20jobi+19$<br>alloc $\leq 2jobi+3$ |
| Knapsack (*ks*) | 122 | 5 | 4 | 1m | steps $\leq 17(w \cdot i.size)+18w+17i.size+18$<br>alloc $\leq 2w+3$ |
| Packrat Parsing (*pp*) [Ford, 2002] | 249 | 7 | 5 | 1m | steps $\leq 61n+58$<br>alloc $\leq 10n+10$ |
| Viterbi (*vit*) [Viterbi, 1967] | 191 | 6 | 7 | 1m | steps $\leq 34k^2t+34k^2-6kt+14k+47t+26$<br>alloc $\leq 2kt+2k+4t+5$ |

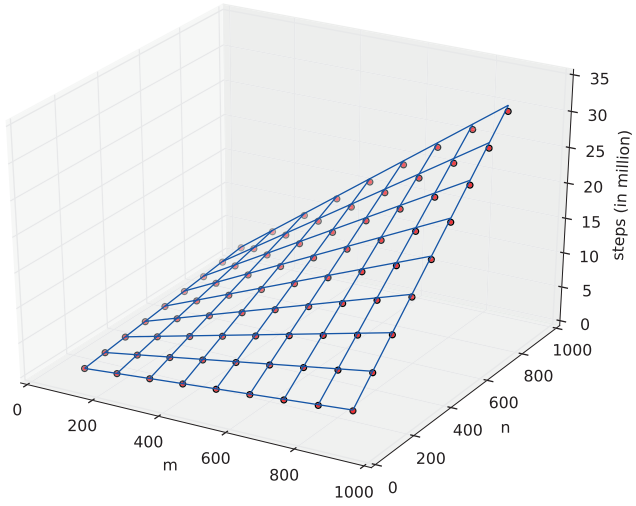
Figure 5.14 – Memoized algorithms verified by ORB

The benchmark *pp* is a memoized implementation of a packrat parser presented by Ford [2002] for the *parsing expression grammar* used in that work. *vit* is an implementation of the Viterbi algorithm for finding the most likely sequence of hidden states in the hidden Markov models. In the case of *ws*, *pp*, and *vit*, the inputs were represented as immutable arrays, which were treated as uninterpreted functions with constant resource usage for random access.

As shown in Figure 5.15, the inferred bounds for steps are on average 90% accurate for the dynamic programming algorithms except *pp* and *vit*, and is 100% accurate in the case of alloc for all benchmarks except *pp*. This is graphically illustrated for two benchmarks *ks* and *levd* in Figures 5.17 and 5.16. The figures plot the static and dynamic resource usages of the benchmarks against the sizes of the inputs for the resource steps. Both these benchmarks have

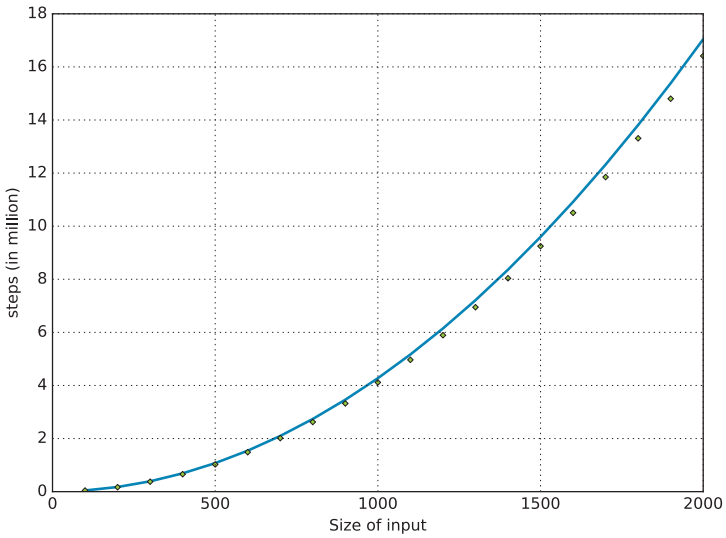
| *B* | *I* | (*dynamic/static) * 100* | | *(optimal/static) * 100* | |
|---|---|---|---|---|---|
| | | *steps* | *alloc* | *steps* | *alloc* |
| *lcs* | 1k | 88 | 100 | 95 | 100 |
| *levd* | 1k | 90 | 100 | 96 | 100 |
| *hmem* | 10k | 79 | 100 | 92 | 100 |
| *ws* | 10k | 99 | 100 | 100 | 100 |
| *ks* | 1k | 94 | 100 | 99 | 100 |
| *pp* | 10k | 77 | 70 | 88 | 84 |
| *vit* | 100 | 42 | 100 | 86 | 100 |
| *Avg.* | | 81 | 98 | 94 | 98 |

Figure 5.15 – Accuracy of bounds inferred for memozied programs




$$\text{steps} \leq 36mn + 36m + 36n + 34$$

Figure 5.16 – Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for Levenshtein distance algorithm




$$\text{steps} \leq 17(w \cdot i.size) + 18w + 17i.size + 18$$

Figure 5.17 – Comparison of the inferred bound (shown as grids) and the dynamic resource usage (shown as dots) for *ks*

a multivariate, nonlinear resource bound. In the case of *ks*, the plot shows the curve along which the values of *l.size* and *w* are identical.

In the case of benchmark *vit*, the low accuracy stems from the *cubic* template (shown in Figure 5.14), as highlighted by the results of comparison with the pareto optimal bound shown in Figure 5.8. In the case of *pp*, the evaluations were performed on random strings as it was difficult to precisely deduce the worst-case input. Nevertheless, the bounds inferred were 100% accurate for the inner functions: pAdd, pMul, and pPrim. However, the worst case behavior of the inner functions rarely manifested when called from the outer functions – only once for every 100 calls.

# 6 Related Work

## 6.1 Resource Analyses

**Push-button Static Resource Analyses** Automatic static inference of resource bounds of programs has been an actively researched area. Most approaches in this space are *push-button* tools that require little or no inputs from the user. These system use either abstract interpretation, or are based on ranking function inference or type inference. Some examples include [Albert et al., 2012, Alias et al., 2010, Avanzini et al., 2015, Brockschmidt et al., 2016, Cook et al., 2009, Danielsson, 2008, Flores-Montoya and Hähnle, 2014, Gulwani et al., 2009, Hoffmann et al., 2012, Jost et al., 2010, Le Métayer, 1988, Navas et al., 2007, Simões et al., 2012, Sinn et al., 2014, Srikanth et al., 2017, Vasconcelos et al., 2015, Zuleger et al., 2011]. Being fully automated, these approaches target simple programs and infer simple or weak bounds on resource usage, due to the absence of knowledge about any complex invariants maintained by the program.

One of the popular tools in this space is Resource-Aware ML [Hoffmann et al., 2012, Jost et al., 2010]. The system automatically infers the resource usage of ML programs using a resource-annotated type inference system, and has been demonstrated on many hand-written functional programs.

Another popular system is SPEED, which was proposed by Gulwani et al. [2009]. They present a technique for inferring symbolic bounds on loops of C programs that is based on instrumenting programs with counters, inferring linear invariants on counters and combining the linear invariants to establish a loop bound. This approach is orthogonal to ours where we attempt to find solutions to user-defined templates. In our benchmarks, we included a few code snippets on which it was mentioned that their tool did not work. Our approach was able to handle them when the templates were provided manually.

The COSTA system [Albert et al., 2012] uses abstract interpretation to construct sound abstractions of the input programs, and uses them to infer an over-approximate bound on the resource usage. It can solve recurrence equations and infer nonlinear time bounds, however,

it does not appear to support algebraic data types nor user-defined functions within resource bounds.

The work of Cook et al. [2009] is one of the few approaches that infers upper bounds on heap usage for programs that manipulate mutable dynamically-allocated data structures such as doubly-linked lists. This work is possibly closest to ours because it performs template-based analysis of imperative programs for finding heap bounds and handles program paths incrementally using the idea of path invariants from Beyer et al. [2007b]. The approach uses a separate (static) shape analysis [Bogudlov et al., 2007] to infer how the sizes of the data structures may change during a computation. Our approach is for functional programs. Our approach handles a wide range of recursive functions over ADTs and is not restricted to size. Our approach can verify resource usage of complex data-structure implementations with first-class function and memoization. Our approach supports nonlinearity and is capable of computing strongest bounds.

**Push-button Analysis for Lazy Evaluation** Danielsson [2008] present a lightweight type-based analysis for verifying time complexity of lazy functional programs and applied it to *implicit queues*. Its applicability to other benchmarks is not evaluated, and hence is unclear. As noted in the paper, the approach is limited in handling aliasing of lazy references, which is crucial for our benchmarks. Simões et al. [2012], Vasconcelos et al. [2015] present a typed-based analysis for inferring bounds on *memory allocations* of Haskell programs. They evaluated their system on cyclic hamming and fibonacci stream, which were included in our benchmarks, and discussed in section 5. In contrast to the above works, our approach is targeted at verifying user-specified bounds, and has been evaluated on more complex, real-world programs for relatively large input sizes. It also support multiple resources. To our knowledge, these are the only existing systems that directly support verification of resource bounds in the presence of lazy evaluation.

**Resource Verification via Interactive Theorem Proving** Another well-studied line of work in resource verification are semi-automatic formal frameworks that are amenable to deriving machine-checked proofs of resource bounds [Benzinger, 2004, Danner et al., 2013, Sands, 1990a,b]. These approaches are complementary to the above push-button tools in that they target very expressive bounds and complex programs that require inputs from the user. However, they are far from being fully automatic and use interactive proof assistants [Bertot and Castéran, 2004, Nipkow et al., 2002b] that involve significant manual labor. In particular, Sands [1990a,b] present one of the earliest formal frameworks for reasoning about resource usage for functional programs in the presence of higher-order features and lazy evaluation. To my knowledge, there does not exist machine-checked proofs for the resource bounds of the lazy data structures considered in our study.

**Resource Verification using Contracts** More recently, a few approaches, apart from the research presented in this dissertation, have started incorporating user specification in resource verification. Carbonneaux et al. [2014] presented a system to verify stack space bounds of C programs written for embedded systems using a quantitative Hoare logic. Previously, Alonso-Blas and Genaim [2012] present an approach where resource bounds are specified by users as templates with holes, which are inferred by the system automatically.

## 6.2 Higher-Order Program Verification

**Coinductive Datatypes** Leino and Moskal [2014] use coinduction to verify programs with possibly infinite lazy data structures. They do not consider resource properties of such programs. Blanchette et al. [2015a,b] present a formal framework for soundly mixing recursion and corecursion in the context of interactive theorem provers.

**Higher-order Contract Verification Systems** Though traditionally the notion of contracts has been restricted to first-order programs, there has been significant work in extending this notion to higher-order programs in order to make specification of properties on such programs more convenient for the users. Some of the recent works in this space include [Findler and Felleisen, 2002, Knowles and Flanagan, 2010, Kobayashi, 2009, Kobayashi et al., 2011, Nanevski et al., 2008, Nguyen and Horn, 2015, Nguyen et al., 2014, Nicolas Voirol and Kuncak, 2017, Tobin-Hochstadt and Horn, 2012, Vazou et al., 2014, Voirol et al., 2015, Vytiniotis et al., 2013, Xu, 2012, Xu et al., 2009]. These works target purely functional programs with the exception of [Nanevski et al., 2008], and typically use a dependent type system that allows types to be refined by predicates (as in Liquid Types [Vazou et al., 2014]), or by even Hoare triples [Nanevski et al., 2008]. Similar to traditional verifiers they also reduce the problem of contract checking to theorem proving by constructing VCs. However, the presence of first-class functions makes this process much trickier and more flexible, allowing for greater variation and novelty among these techniques.

Many of these systems allow users to write contracts on function-valued parameters, or refinement predicates on function types [Findler and Felleisen, 2002, Vazou et al., 2014]. To my knowledge, there does not exist any contract-based verifiers for higher-order programs that allow specifying resource properties, as in our approach. Our approach allows named functions, with contracts and resource templates, to be used inside lambdas. However, it disallows contracts on function-valued parameters and instead provides intensional-equality-based constructs to specify their properties. Though this makes the contracts very specific to the implementation, it has the advantage of reducing specification burden for closed or encapsulated programs. Supporting contracts on function-valued parameters that can refer to resource bounds would be an interesting future direction to explore.

## 6.3 Software Verification

**Template-based Invariant Inference** Beyer et al. [2007a] presents an approach for handling uninterpreted functions in templates. Our approach handles disjunctions that arise because of axiomatizing uninterpreted functions efficiently through our incremental algorithm that is driven by counterexamples and are able to scale to VCs with hundreds of uninterpreted functions. Our approach also supports algebraic data types and handles sophisticated templates that involve user-defined functions. The idea of using Farkas' lemma to solve linear templates of numerical programs goes back at least to the work of Colón et al. [2003] and has been generalized in different directions by Sankaranarayanan et al. [2004], Cousot [2005b], Gulwani et al. [2008]. Cousot [2005b] and Sankaranarayanan et al. [2004] present systematic approaches for solving nonlinear templates for numerical programs. Our approach is currently based on light-weight axiomatization of nonlinear operations which is targeted towards practical efficiency. It remains to be seen if we can integrate more complete non-linear reasoning into our approach without sacrificing scalability.

**Contract-based and Interactive Software Verification** Contract-based software verification has been a corner stone of program verification. These systems target verification of properties expressed in the form of pre-and-postconditions (i.e contracts) on functions/methods, or as assertions sprinkled within the program code. Some of the recent systems belonging to this space include DAFNY [Leino, 2010], JSTAR [Distefano and Parkinson J, 2008], GRASSHOPPER [Piskac et al., 2014], JAHOB [Zee et al., 2008], VERIFAST [Jacobs et al., 2011] and VIPER [Müller et al., 2016]. These approaches typically encode the verification problem as a logical formula, known as verification condition (VC), such that the validity of the VC entails the correctness of the contracts. The VCs are generally decided using off-the-shelf theorem provers like [Barrett et al., 2011, de Moura and Bjørner, 2008] but may also require custom decision procedures. Often, these approaches rely on users to provide sufficiently strong specifications that can be proven using a predefined set of proof strategies like mathematical induction. However, they place very few restriction on the class of properties that can be verified, which is only limited by the underlying theorem provers and the necessary manual effort.

These approaches and interactive theorem provers [Bertot and Castéran, 2004, Chlipala, 2011, Nipkow et al., 2002a] have been used to verify complex, imperative programs. Automation in our system appears above the one in interactive provers, and could be further improved using quantifier instantiation, induction, and static analysis [Beyene et al., 2013, Gurfinkel et al., 2015, Reynolds and Kuncak, 2015]. While most approaches for imperative programs target a homogeneous, mutable heap, in this work we consider an almost immutable heap except for the cache, and use a set representation to handle mutations to the cache efficiently. I believe that similar separation of heap into mutable and immutable parts can benefit other forms of restricted mutation like *write-once* fields [Arvind et al., 1989]. Using mutation directly to model caches will dramatically increase the contract overhead in our benchmarks. In fact, there are not many systems that can combine mutation with higher-order features.

**Software Model Checking** Automated approaches for software verification has been a subject of intense study. Some of the recent, popular tools include [Ball and Rajamani, 2002, Beyene et al., 2013, Calcagno et al., 2009, Cousot, 2005a, Cousot et al., 2005, Henry et al., 2012, Henzinger et al., 2002, 2004, Hoder and Bjørner, Itzhaky et al., 2014, Kobayashi et al., 2011, Nori et al., 2009]. Compared to contract-based verifiers, these approaches – also referred to as software model checkers or property checkers – are completely automated, and are often specialized for verification of specific properties such as proving absence of memory leaks, checking correctness of software API usage, or verifying absence of runtime errors. However, in principle, they could be used to verify user-provided assertions and specifications [Henzinger et al., 2002, Nori et al., 2009], and thus are closely related to contract-based software verification. However, being completely automated these approaches have only been able to handle properties belonging to very restricted domains such as linear relations between program variables. Nonetheless, decades of research in software model checking has resulted in highly effective techniques for automatically reasoning about programs, e.g. counterexample guided abstraction refinement (CEGAR) [Ball and Rajamani, 2002], abstract interpretation and widening [Cousot and Cousot, 1979, Cousot et al., 2005], interpolation [Ball and Rajamani, 2002, Henzinger et al., 2004], property-directed reachability (PDR) [Bradley, 2011, Hoder and Bjørner], constraint-solving-based invariant inference [Cousot, 2005a], bi-abduction [Calcagno et al., 2009] and also data-driven approaches [Ernst et al., 2001, Sharma et al., 2013]. While the techniques have traditionally been applicable to first-order programs, recently they have been applied to verification of higher-order programs as well [Kobayashi et al., 2011]. While some of these techniques have also been used to automate verification of resource properties, there still remains a large pool of techniques which can be used to further increase automation in resource verification and enable them scale to large applications.

# 7 Conclusion and Future Work

> You insist that there is something a machine cannot do.
> If you tell me precisely what it is a machine cannot do,
> then I can always make a machine which will do just that.
> — John von Neumann

Static analysis of resource usage behavior of programs is an important problem with numerous applications. It has hence been a subject of intense study over the past several decades. However, many existing static resource analyses such as [Albert et al., 2012, Hoffmann et al., 2012] infer best-effort upper bounds and hope that they match users' expectations. This dissertation presented a complementary approach aimed at verifying user-specified bounds on resource utilization of programs. The approach requires users to specify high-level invariants and preconditions in the form of function contracts, but automatically solves low-level constraints that are within the scope of decidable SMT theories. Users can interact with the system by providing hints and/or by fleshing out parts of the proof that are difficult to automatically infer. These poof hints are typically properties of recursive functions or nonlinear operations. Ideally, these hints are also established within the system. However, even if such hints cannot be proven within the system either because their proofs are tedious or even impossible to express within the proof system, we can still derive resource bounds that are sound modulo the soundness of the (unproved) hints.

In the world of proving correctness properties, the advantages of such contract-based verifiers over push-button techniques is broadly known and recognized. This dissertation demonstrates that such contract-based techniques will also enable verification of resource bounds of programs that are challenging for automatic as well as manual reasoning. While fully-automated techniques offer great value and are highly desirable especially with regards to ease of use, they are bound to be incomplete. No matter how sophisticated an automated analysis is there will be programs that it cannot analyze precisely. As implied by the somewhat comical quote by Von Neumann mentioned at the beginning of this section, program-specific knowledge can go a long way in pushing automated systems to programs (or tasks) that existing

systems "cannot do".

The observation that user-annotations help resource analysis is not by itself surprising. (In fact, user-annotations are likely to benefit almost every non-trivial static analysis of semantic properties.) What this dissertation addresses is the "what" and "how" aspect of incorporating user annotations in resource verification, i.e, it studies the kind of user-annotations that are required, and how they can be expressed and effectively utilized in resource verification. In my opinion, the high-level contributions of this dissertation are three fold. Firstly, this dissertation demonstrates that to express and verify resource properties we can exploit the complete verification machinery developed for establishing correctness properties e.g. function-level contracts, template-based invariant inference, algorithms for translating contract checking to logical formulas and decision produces for solving logical fragments. Secondly, it demonstrates that viewing resource bounds as invariants of a program instrumented for resources is practically viable and also beneficial. This view essentially provides a way to convert any correctness verifier into a resource verifier. Finally, it identifies effective and minimal set of primitives such as resource templates, isConcrete construct, structural equality and matching of closures, that enable expression of complex properties needed to establish resource bounds of programs written in a higher-order functional language with memoization. It also presents algorithms for verifying such specifications. In the sequel, I discuss some of the interesting enhancements that can be made to the approach presented in this dissertation with minor extensions.

**Improving Accuracy of the Cost Model** While the steps resource presented here counted every primitive operation once, it is possible to define new resources that count specific classes of the primitive operations separately. For instance, it is quite straightforward to define a resource that counts only the arithmetic/logical operations, and another resource that counts only the memory operations (such as load/store). These resources can be more effectively used to compare implementations at a fine-grained level. Similarly, it is possible to define a fine-grained alloc resource that separately counts the number of objects of specific *types* that are allocated in the heap. Furthermore, to more accurately measure heap memory usage, it possible define the cost of each allocation proportional to the size of the object that is allocated.

**Modeling Resources that can be Freed** The instrumentation presented here can be extended to support resources that could be freed or reclaimed at different points in the program, either explicitly by the user or automatically. For instance, these include resources such as the the peak memory usage with manual memory management, the number of locks held by a program, and the number of open file handlers. Such resources can be modeled if the cost of the constructs that free up such resources are assigned a value that is the negative of the amount of resources that is freed up.

However, in the presence of automatic garbage collection, estimating the peak memory usage becomes quite tricky. In such cases, one may have to conservatively approximate the behavior of the garbage collector.

**Memoization with Non-monotonic Caches** The set abstraction of a cache presented in Chapter 4 can be extended to non-monotonic caches, where certain entries can be explicitly removed, and also to bounded caches, where entries cannot be added to the cache if it is filled up to its maximum capacity. Recall that the cache is represented as a set of keys and is propagated through the expressions of the program. Constructs that explicitly remove an entry from a cache could be modeled by removing the corresponding cache key from the abstract cache state reaching the program point at which the construct appears. To model a bounded cache, one could use an additional counter that tracks the size of the cache at every program point. The cache instrumentation should be modified so that every memoized call adds an entry to the cache if and only if the size of the cache at that point is below the maximum capacity of the cache.

A more involved but interesting extension is modeling bounded caches that allow entries to be replaced using a predefined replacement policy e.g. first-come-first-serve (FCFS) policy. If we can accurately model such caches, we can also use them to model the hardware (memory) cache and compute a more precise estimate of the physical running time. A main challenge in modeling such caches is precisely tracking all the entries in the cache. Whether such detailed specifications can be expressed in a practical way and whether they can scale to complex higher-order programs such as those considered in this dissertation is an interesting future direction to explore. However, it is to be noted that there has been significant efforts ([Wilhelm et al., 2008]) in developing execution time analyses for low-level programs that take into account the effects of hardware caches.

This dissertation showed that statically verifying resource usage of complex programs is feasible provided users/developers input sufficiently detailed specifications and proof hints. However, a pragmatic and open-ended question that is not considered by this dissertation is whether *verifying abstract resource usage of software is worth the effort?* There are two main concerns that prevent this question from being answered affirmatively. The first concern is the practical value added by the abstract resource bounds that are verified, especially when we know that the highly intricate and tricky aspects of physical resource usage are not modeled by the abstract resource usage. The second concern is that the performance of a program for an average (or typical) execution scenario is of greater interest, and any deterioration in performance for an input that appears rarely can be tolerated. I conclude this dissertation by sharing my thoughts on both these aspects.

Firstly, it is a surprising fact that resource verification not only verifies performance but indirectly also helps in verifying correctness of programs. The reason for this is that most sophistication in data structures or programs is for achieving better (asymptotic) performance.

In many cases, the invariants maintained by programs for achieving the desired resource usage are quite complex. For instance, think of the color and height invariants of the red-black data structure. Their sole purpose is to make tree operations run in time logarithmic in the size of the tree. Often these invariants are complex pieces of code. A real problem with these complex specification is determining whether the specifications are correct. By verifying resource bounds we establish the correctness of these specifications. In fact, even if the specifications do not exactly match the user's expectation, it is still does not matter as long as they entail the desired resource usage. In other words, resource verification provides a way to ensure that the sophistications built in to programs for better performance are indeed correct and produce the desired effect.

Secondly, while it is true that abstract resource usage does not capture the tricky aspects of physical resource usage, in the absence of the ability to precisely estimate the former, the next best fall back is to reason about the latter. There are numerous application domains where even abstract resource usage may prove to be very useful. For instance, they can be used by compile-time or runtime optimizers to select an implementation that is likely to perform better under a specific compile-time or runtime context. They can be used to measure the changes in the memory usage of an application across different versions and hence identify memory bloats. Furthermore, they can be used to establish infeasibility of security exploits based on resource consumption of programs such as side-channel attacks. I believe that the availability of a robust system that can establish resource bounds at the level of precision described here will enable many novel applications in the time to come.

# Bibliography


E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012. doi: 10.1016/j.tcs.2011.07.009.

C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium, SAS*, pages 117–133, 2010. URL http://dl.acm.org/citation.cfm?id=1882094.1882102.

D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis Symposium, SAS*, pages 405–421, 2012. doi: 10.1007/978-3-642-33125-1_27.

H. Apfelmus. Quicksort and k-th smallest elements. 2009. URL http://apfelmus.nfshost.com/articles/quicksearch.html.

A. W. Appel. Intensional equality ;=) for continuations. *SIGPLAN Not.*, 31(2), Feb. 1996. doi: 10.1145/226060.226069.

Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, Oct. 1989. doi: 10.1145/69558.69562.

M. Avanzini, U. D. Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *International Conference on Functional Programming, ICFP*, pages 152–164, 2015. doi: 10.1145/2784731.2784753.

T. Ball and S. K. Rajamani. The s lam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.

C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification, CAV*, pages 171–177, 2011.

R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1):79 – 103, 2004. doi: http://dx.doi.org/10.1016/j.tcs.2003.10.022.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004. ISBN 978-3-642-05880-6. doi: 10.1007/978-3-662-07964-5.

T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *Computer Aided Verification, CAV*, 2013. doi: 10.1007/978-3-642-39799-8_61.

D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, 2007a. doi: 10.1007/978-3-540-69738-1_27.

D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007b. doi: 10.1145/1250734.1250769.

R. Bird and P. Wadler. *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd., 1988. ISBN 0-13-484189-1.

R. Bird, G. Jones, and O. De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, 1997.

R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system. In *Scala Workshop*, 2013.

J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *European Symposium on Programming, ESOP*, pages 359–382, 2015a. doi: 10.1007/978-3-662-46669-8_15.

J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion: a proof assistant perspective. In *International Conference on Functional Programming, ICFP*, pages 192–204, 2015b. doi: 10.1145/2784731.2784732.

G. E. Blelloch and B. M. Maggs. Parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.

I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping TVLA: making parametric shape analysis competitive. In *Computer Aided Verification, CAV*, pages 221–225, 2007. doi: 10.1007/978-3-540-73368-3_25.

A. R. Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer Berlin Heidelberg, 2011.

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, pages 13:1–13:50, 2016. doi: 10.1145/2866575.

C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGPLAN Notices*, volume 44, pages 289–300. ACM, 2009.

Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation, PLDI*, 2014. doi: 10.1145/2594291.2594301.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation, PLDI*, pages 234–245, 2011. doi: 10.1145/1993498.1993526.

M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification, CAV*, 2003. doi: aba.

B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, 2009. doi: 10.1109/FMCAD.2009.5351120.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005a. doi: 10.1007/978-3-540-30579-8_1.

P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005b. doi: 10.1007/978-3-540-30579-8_1.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979. doi: 10.1145/567752.567778.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer Berlin Heidelberg, 2005.

N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages, POPL*, pages 133–144, 2008. doi: 10.1145/1328438.1328457.

N. Danner, J. Paykin, and J. S. Royer. A static cost analysis for a higher-order language. In *Workshop on Programming languages meets program verification, PLPV*, pages 25–34, 2013. doi: 10.1145/2428116.2428123.

S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, pages 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24.

D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for java. In *Object-oriented Programming Systems Languages and Applications, OOPSLA*, pages 213–226, 2008. doi: 10.1145/1449764.1449782.

E. Dolstra. Maximal laziness: An efficient interpretation technique for purely functional {DSLs}. *Electronic Notes in Theoretical Computer Science*, 238(5):81 – 99, 2009. doi: http://dx.doi.org/10.1016/j.entcs.2009.09.042.

M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2): 99–123, 2001.

M. Fähndrich and K. R. M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, IWACO*, page 58, 2003.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming, ICFP*, pages 48–59, 2002. doi: 10.1145/581478.581484.

A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS*, pages 275–295, 2014. doi: 10.1007/978-3-319-12736-1_15.

B. Ford. Packrat parsing: Simple, powerful, lazy, linear time, functional pearl. In *International Conference on Functional Programming ICFP*, pages 36–47, 2002. doi: 10.1145/581478.581483.

J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2): 7:1–7:39, Feb. 2011. doi: 10.1145/1890028.1890030.

S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008. doi: 10.1145/1375581.1375616. URL http://doi.acm.org/10.1145/1375581.1375616.

S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages, POPL*, 2009. doi: 10.1145/1480881.1480898.

A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Compuer Aided Verification, CAV*, 2015.

J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009. ISBN 978-0-521-89957-4.

C. Hawblitzel. Automated verification of practical garbage collectors. In *Symposium on Principles of Programming Languages*, January 2009.

J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002.

T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *ACM SIGPLAN Notices*, volume 39, pages 232–244. ACM, 2004.

K. Hoder and N. Bjørner. Generalized property directed reachability. *Theory and Applications of Satisfiability Testing–SAT 2012*, page 157.

J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In *Computer Aided Verification, CAV*, pages 781–786, 2012.

J. Hoffmann, A. Das, and S.-C. Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of Principles of Programming Languages*, pages 359–373. ACM, 2017.

S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. Property-directed shape analysis. In *International Conference on Computer Aided Verification*, pages 35–51. Springer, 2014.

B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of NASA Formal Methods, NFM*, pages 41–55, 2011.

N. D. Jones and N. Bohr. Termination analysis of the untyped lamba-calculus. In *Rewriting Techniques and Applications, RTA*, pages 1–23, 2004. doi: 10.1007/978-3-540-25979-4_1. URL http://dx.doi.org/10.1007/978-3-540-25979-4_1.

S. Jost, K. Hammond, H. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages, POPL*, pages 223–236, 2010. doi: 10.1145/1706299.1706327.

D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *Foundations of Software Engineering, FSE*, pages 105–116, 2006. doi: 10.1145/1181775.1181789. URL http://doi.acm.org/10.1145/1181775.1181789.

M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0792377443.

G. Klein, P. Derrin, and K. Elphinstone. Experience report: Sel4: Formally verifying a high-performance microkernel. In *International Conference on Functional Programming, ICFP*, pages 91–96, 2009. doi: 10.1145/1596550.1596566.

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2): 6:1–6:34, Feb. 2010. doi: 10.1145/1667048.1667051.

N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Principles of Programming Languages, POPL*, pages 416–428, 2009. doi: 10.1145/1480881.1480933.

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Programming Language Design and Implementation, PLDI*, pages 222–233, 2011. doi: 10.1145/1993498.1993525.

V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3), 2006. URL http://dx.doi.org/10.1007/s10817-006-9042-1.

A. Lal and S. Qadeer. Dag inlining: A decision procedure for reachability-modulo-theories in hierarchical programs. In *Programming Language Design and Implementation*, PLDI, 2015. doi: 10.1145/2737924.2737987. URL http://doi.acm.org/10.1145/2737924.2737987.

D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10 (2):248–266, Apr. 1988. doi: 10.1145/42190.42347.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010. doi: 10.1007/978-3-642-17511-4_20.

K. R. M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *Formal Methods, FM*, pages 382–398, 2014. doi: 10.1007/978-3-319-06410-9_27.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814.

R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification, CAV*, pages 762–778, 2014. doi: 10.1007/978-3-319-08867-9_51.

R. Madhavan, S. Kulal, and V. Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of Principles of Programming Languages*, pages 330–343. ACM, 2017.

J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012. doi: 10.1145/2187671.2187672.

P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation1. *J. Funct. Program.*, 18(5-6):865–911, Sept. 2008. ISSN 0956-7968.

J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *International Conference on Logic Programming, ICLP*, pages 348–363, 2007. doi: 10.1007/978-3-540-74610-2_24.

P. C. Nguyen and D. V. Horn. Relatively complete counterexamples for higher-order programs. In *Programming Language Design and Implementation, PLDI*, pages 446–456, 2015. doi: 10.1145/2737924.2737971.

P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In *international conference on Functional programming, ICFP*, pages 139–152, 2014. doi: 10.1145/2628136.2628156.

R. M. Nicolas Voirol and V. Kuncak. Termination of open higher-order programs, EPFL-REPORT-229918. Technical report, EPFL, 2017.

T. Nipkow. Amortized complexity verified. In *Interactive Theorem Proving (ITP 2015)*, volume 9236, pages 310–324, 2015.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002a. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002b.

A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181. Springer, 2009.

C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5:583–592, 10 1995. ISSN 1469-7653. doi: 10.1017/S0956796800001489. URL http://journals.cambridge.org/article_S0956796800001489.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

D. C. Oppen. Elementary bounds for presburger arithmetic. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, 1973.

R. Piskac, T. Wies, and D. Zufferey. Grasshopper - complete heap verification with mixed specifications. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, pages 124–139, 2014. doi: 10.1007/978-3-642-54862-8_9.

A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, EPFL, 2014.

A. Prokopec and M. Odersky. Conc-trees for functional and parallel programming. In *Languages and Compilers for Parallel Computing, LCPC*, pages 254–268, 2015. doi: 10.1007/978-3-319-29778-1_16.

A. Reynolds and V. Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, pages 80–98, 2015. doi: 10.1007/978-3-662-46081-8_5.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. doi: 10.1023/A:1010027404223.

A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.

D. Sands. Complexity analysis for a lazy higher-order language. In *European Symposium on Programming, ESOP*, pages 361–376, 1990a. doi: 10.1007/3-540-52592-0_74.

D. Sands. *Calculi for Time Anlaysis of Functional Programs*. PhD thesis, Imperial College, University of London, 1990b. URL http://www.cse.chalmers.se/~dave/papers/PhDthesis.ps.

S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, 2004. URL http://doi.acm.org/10.1145/964001.964028.

D. Sereni. *Termination analysis of higher-order functional programs*. PhD thesis, University of Oxford, UK, 2006. URL http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.437001.

R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*, pages 574–592. Springer, 2013.

H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *International Conference on Functional Programming, ICFP*, pages 165–176, 2012. doi: 10.1145/2364527.2364575.

M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Computer Aided Verification CAV*, pages 745–761, 2014. doi: 10.1007/978-3-319-08867-9_50.

A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006. doi: 10.1145/1168857.1168907.

A. Srikanth, B. Sahin, and W. R. Harris. Complexity verification using guided theorem enumeration. In *Principles of Programming Languages*, pages 639–652. ACM, 2017.

P. Suter. *Programming with Specifications*. PhD thesis, EPFL, Switzerland, 2012.

P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010. doi: 10.1145/1706299.1706325. URL http://doi.acm.org/10.1145/1706299.1706325.

P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Symposium on Static Analysis SAS*, 2011. doi: 10.1007/978-3-642-23702-7_23.

W. Swierstra. Stream: A library for manipulating infinite lists. https://hackage.haskell.org/package/Stream-0.4.7.2/docs/Data-Stream.html. 2015.

S. Tobin-Hochstadt and D. V. Horn. Higher-order symbolic execution via contracts. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 537–554, 2012. doi: 10.1145/2384616.2384655.

G. S. Tseitin. On the complexity of derivation in propositional calculus. *Zapiski Nauchnykh Seminarov LOMI*, 8:234–259, 1968.

P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *European Symposium on Programming, ESOP*, 2015. doi: 10.1007/978-3-662-46669-8_32.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *International Conference on Functional Programming, ICFP*, pages 269–282, 2014. doi: 10.1145/2628136.2628161.

A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967. ISSN 0018-9448. doi: 10.1109/TIT.1967.1054010.

N. Voirol, E. Kneuss, and V. Kuncak. Counter-example complete verification for higher-order functions. In *Symposium on Scala*, pages 18–29, 2015. doi: 10.1145/2774975.2774978.

D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Principles of Programming Languages, POPL*, pages 431–442, 2013. doi: 10.1145/2429069.2429121.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389.

D. N. Xu. Hybrid contract checking via symbolic simplification. In *Workshop on Partial Evaluation and Program Manipulation, PEPM*, pages 107–116, 2012. doi: 10.1145/2103746.2103767.

D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for haskell. In *Principles of Programming Languages, POPL*, pages 41–52, 2009. doi: 10.1145/1480881.1480889.

K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *Programming Language Design and Implementation, PLDI*, 2008. doi: 10.1145/1375581.1375624.

T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for recursive data structures with integer constraints. In *International Joint Conference on Automated Reasoning*, pages 152–167. Springer, 2004.

F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis Symposium, SAS*, pages 280–297, 2011. doi: 10.1007/978-3-642-23702-7_22.

# Ravichandhran Kandhadai Madhavan

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
BC 355, Station 14
CH-1015 Lausanne, Switzerland
Phone: +41 21 69 31243
Email: ravi.kandhadai@epfl.ch
Website: `http://lara.epfl.ch/~kandhada`
Google Scholar: `http://bit.ly/2lF1W3p`
ACM author profile: `http://bit.ly/2lDulEQ`

**Research Interests**

Programming Languages, Software Analysis, Verification, Formal Methods, Computer Science Education

**Education**

**EPFL (École Polytechnique Fédérale de Lausanne)** Sep 2012 – Aug 2017 (expected)
Ph.D. candidate in Computer Science
*Advisor:* Prof. Viktor Kuncak
*PhD Dissertation:* Resource bounds verification for functional programs

**IISc (Indian Institute of Science), Bangalore** Aug 2008 – Jul 2010
Master of Engineering in Computer Science
Cumulative GPA 7/8. Awarded distinction
*Advisor:* Prof. Raghavan Komondoor
*Masters Thesis:* Demand-driven null dereference verification for Java

**College of Engineering, Guindy** Aug 2004 – Jul 2008
Bachelors of Engineering in Computer Science
Cumulative GPA 9.1/10. Awarded distinction and rank certificate

**Employment**

**Research Assistant, Microsoft Research India** Jul 2010 – Aug 2012
Rigorous Software Engineering group
*Collaborators:* G. Ramalingam, K. Vaswani
Developed a scalable, modular heap analysis technique

**Research Intern, Microsoft Research India** Summer 2013

**Teaching Experience**

**Teaching Assistant, EPFL, Switzerland** 2013 – 2016
Participated in preparing and grading assignments and exams, conducted exercise sessions, and occasionally gave lectures for the following courses:

- Basic Computer Programming I — Spring 2013
- Advanced Compiler Construction — Spring 2014
- Head TA for Computer Language Programming & Compilers — Fall 2014, 2015, 2016
  *Developed and used an online tutoring system for context-free grammars*
- Parallel and Reactive Programming in Scala — Spring 2015
- Head TA for Parallel and Concurrent Programming in Scala — Spring 2016
  *Developed and used a unit test engine for evaluating student assignments on concurrent programming*

**Teaching Assistant, IISc, Bangalore**
- Program Analysis and Verification — Spring 2010

**Reviewing Activities**

- Served in the External Review Committee (ERC) of PLDI 2017 conference (`http://bit.ly/2lkxzMK`)
- Served as additional reviewer for POPL 2014 (`http://bit.ly/2lahlF3`), SAS 2014 (`http://bit.ly/2mhyLVs`) and IJCAR 2014 (`http://bit.ly/2lPElx2`) conferences

**Fellowships & Awards**

- Awarded ACM SIGPLAN PAC grant for attending SPLASH 2015
- EPFL PhD fellowship (2012-2013)
- Ministry of Human Resource Development India scholarship for master studies (2008-2010)
- All India rank 27 (among ∼17K students) in CS Graduate Aptitude Test in Engineering (2008)
- Awarded university rank 6 in B.E. computer science (2008)
- Ranked 180th (among ∼150K students) in the state engineering entrance examinations TNPCEE (2004)

**Conference Presentations & Invited Talks**

- "Contract-based resource verification for higher-order functions with memoization", POPL 2017, Paris
- "Testing student assignments on concurrent programming", SCALA 2016, Amsterdam
- "Automating grammar comparison", OOPSLA, SPLASH 2015, Pittsburgh
- "Symbolic resource bounds inference", CAV 2014, Vienna
- "Modular heap analysis for higher-order programs", SAS 2012, Deauville
- "Purity analysis: abstract interpretation formulation", SAS 2011, Venice
- Presented invited talks at MSR India, IIT Madras, IISc Bangalore, MPI SWS

**Conference Publications**

(Every publication listed below is a full research paper and is *not* a short/tool paper.)

1. **Contract-based resource verification for higher-order functions with memoization.**
   Ravichandhran Madhavan, Sumith Kulal, Viktor Kuncak.
   *Principles of Programming Languages (POPL)*, 2017.

2. **A scala library for testing student assignments on concurrent programming.**
   Mikaël Mayer, Ravichandhran Madhavan.
   *Scala Symposium (SCALA)*, 2016.

3. **Automating grammar comparison.**
   Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak.
   In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

4. **Symbolic resource bound inference for functional programs.**
   Ravichandhran Madhavan, and Viktor Kuncak.
   In *Computer Aided Verification (CAV)*, 2014.

5. **Modular heap analysis for higher-order programs.**
   Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani.
   In *Symposium on Static Analysis (SAS)*, 2012.

6. **Null dereference verification via over-approximated weakest precondition analysis**.
   Ravichandhran Madhavan, and Raghavan Komondoor.
   In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

7. **Purity analysis: an abstract interpretation formulation.**
   Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani.
   In *Symposium on Static Analysis (SAS)*, 2011.

**Journal Publications**

8. **A framework for efficient modular heap analysis.**
   Ravichandhran Madhavan, G. Ramalingam and Kapil Vaswani.
   In *Foundations and Trends In Programming Languages (FnTPL), Volume 1, Issue 4*, 2015.