

Deductive Synthesis and Repair

THÈSE N° 6878 (2016)

PRÉSENTÉE LE 26 FÉVRIER 2016

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

Etienne KNEUSS

acceptée sur proposition du jury:

Prof. E. Bugnion, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. R. Alur, rapporteur
Prof. M. Vechev, rapporteur
Prof. C. Koch, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Acknowledgements

First and foremost I would like to thank my advisor, Viktor Kuncak, for his constant support and guidance throughout my studies. I am thankful that he trusted me with my odd research projects. It has been a real pleasure working with him.

I would like to extend my gratitude to the members of my review committee: Rajeev Alur, Martin Vechev, Chirstoph Koch, and Martin Odersky. I thank them for their constructive remarks and insightful comments during my defense.

Without my co-authors, parts of this dissertation would not exist: I thank Philippe Suter, Ivan Kuraj, Manos Koukoutos, and Régis Blanc. Specifically, I thank Philippe for managing to motivate me to enter the wonderful world of program synthesis, and Manos for his help and insights regarding the work on repair.

My life at the lab has always been enjoyable, mostly thanks to the past and present members of the LAMP and LARA labs, including Adriaan, Alex, Ali, Andrej, Andrew, Eva, Filip, Guilano, Heather, Hossein, Hubert, Ingo, Ivan, Jad, Lukas, Manohar, Manos, Miguel, Mikaël, Nada, Philippe, Régis, Sandro, Sébastien, Vojin and Vlad.

I thank Danielle Chamberlain and Yvette Gallay for their administrative support and making sure we all meet outside of EPFL, at least once a year. I also thank Holly for spending time to spot my numerous English mistakes. I thank Fabien Salvi for making sure our infrastructure meets our eccentric requirements.

I also thank my family for keeping the amount of computer-related support requests to a strict minimum.

Finally, of course I thank Aline.

Lausanne, December 2015

E. K.

Preface

A new generation of software development tools is emerging these years. These tools can symbolically explore infinitely many possible executions of programs while making use of any user-supplied contracts, much like compilers make use of types. They can not only find errors, but also correct them, ensuring that the program meets its specification, or even synthesize functions from specifications alone. This dissertation presents a preview of such tools and the new techniques needed to build them.

The central stage of the thesis occupy synthesis techniques. The remarkable idea of deductive synthesis re-appears here in the most useful form it has ever taken. Several game changing technologies complement deductive synthesis compared to its original deployment decades ago: satisfiability modulo theory solvers enable much more automated verification and unprecedented ability to find counterexamples. Using counter-examples to provide feedback into synthesis in itself is another crucial idea that emerged from model checking as well as from synthesis over bounded domains.

Even after taking best available techniques into account, synthesis over unbounded domains remains an extremely challenging problem. This thesis aims to synthesize pure recursive functions over unbounded domains, with no a priori restriction on the form of specifications or programs to synthesize. It introduces a synthesis framework that proved general enough to accommodate different synthesis steps yet specific enough to accommodate expressing algorithmic insights and serving as the starting point for an implementation. It introduces a new approach to counter-example guided synthesis, which makes use of SMT solvers to navigate an exponential space of programs while permitting the use of program execution to quickly rule out invalid candidate programs.

Synthesis can be leveraged in fully automated mode, where it can generate from specifications functions such as sorting or simple data structure operations. Its use with manual synthesis rules can be made arbitrarily expressive, though it is not necessary because an editor and verification engine remain at the disposal of the programmer.

The thesis presents program repair as a practical deployment of synthesis technology. The best way to explain where an error was to repair a program into the form where it has no error. Repair can be naturally invoked whenever verification identifies a problem, and can leverage the existing program to reduce the search space for a correct solution. Indeed, the repaired programs are several times larger than programs that can be synthesized from scratch, while the set of repairs is not nearly as bounded as in heuristic repair techniques.

The time at which these tools arise makes their deployment as a web service an obvious choice.

Chapter 0. Preface

Instead of increasingly bulky desktop development environments, web services work with zero installation, support collaboration, and can leverage scalable remote computational resources to perform their tasks. The tool shown in this thesis sets a new standard in terms of both ease of use and the breadth of functionality of tools for developing correct software.

Lausanne, 14 February 2016

Viktor Kuncak

Abstract

In this thesis, we explore techniques for the development of recursive functional programs over unbounded domains that are proved correct according to their high-level specifications. We present algorithms for automatically synthesizing executable code, starting from the specification alone. We implement these algorithms in the Leon system. We augment relational specifications with a concise notation for symbolic tests, which are helpful to characterize fragments of the functions' behavior. We build on our synthesis procedure to automatically repair invalid functions by generating alternative implementations. Our approach therefore formulates program repair in the framework of deductive synthesis and uses the existing program structure as a hint to guide synthesis. We rely on user-specified tests as well as automatically generated ones to localize the fault. This localization enables our procedure to repair functions that would otherwise be out of reach of our synthesizer, and ensures that most of the original behavior is preserved.

We also investigate multiple ways of enabling Leon programs to interact with external, untrusted code. For that purpose, we introduce a precise inter-procedural effect analysis for arbitrary Scala programs with mutable state, dynamic object allocation, and dynamic dispatch. We analyzed the Scala standard library containing 58000 methods and classified them into several categories according to their effects. Our analysis proves that over one half of all methods are pure, identifies a number of conditionally pure methods, and computes summary graphs and regular expressions describing the side effects of non-pure methods.

We implement the synthesis and repair algorithms within the Leon system and deploy them as part of a novel interactive development environment available as a web interface. Our implementation is able to synthesize, within seconds, a number of useful recursive functions that manipulate unbounded numbers and data structures. Our repair procedure automatically locates various kinds of errors in recursive functions and fixes them by synthesizing alternative implementations.

Key words: declarative programming, program synthesis, synthesis procedures, program repair, static analysis, memory effects

Résumé

Dans cette thèse, nous explorons des techniques pour le développement de programmes fonctionnels récur­sifs qui sont corrects par rapport à une spécification de haut niveau. Nous présentons des algorithmes qui synthétisent automatiquement des programmes exécutables en partant uniquement de leurs spécifications. Nous implémentons ces algorithmes dans le système Leon. Nous accompagnons ces spécifications relationnelles d’une notation concise pour les tests symboliques qui sont utiles afin de caractériser certains aspects du comportement des fonctions. Nous étendons ensuite cette procédure de synthèse afin de permettre la réparation automatique d’une fonction invalide en générant une implémentation alternative qui satisfait sa spécification. Notre approche formule ainsi la réparation de programmes dans le cadre de la synthèse déductive et se base sur le programme existant pour guider cette synthèse. Nous utilisons des tests générés automatiquement ainsi que ceux fournis par les utilisateurs pour localiser la source de l’erreur. Cette localisation nous permet de réparer des fonctions qui seraient sinon hors de portée de nos algorithmes de synthèse et nous garantit également de conserver une grande partie du comportement du programme d’origine.

Nous étudions aussi diverses manières de permettre à Leon d’interagir avec des programmes externes pour lesquels nous n’avons aucune garantie. Pour cette raison nous introduisons une analyse inter-procédurale d’effets pour des programmes Scala arbitraires, et en particulier pour les programmes avec de l’état mutable, de l’allocation dynamique d’objets et des appels dynamiques à des méthodes. Nous avons analysé la librairie standard Scala qui contient 58’000 méthodes et les avons classifiées en fonction de leurs effets. Notre analyse montre que plus de la moitié de ces méthodes sont pures, qu’une partie sont conditionnellement pures. Elle rapporte les effets des méthodes impures sous la forme d’expressions régulières.

Nous implémentons les algorithmes de synthèse et de réparation dans le système Leon et les déployons dans un nouvel environnement de développement disponible sous la forme d’une interface Web. Notre implémentation est capable de synthétiser en quelques secondes un nombre de fonctions récur­sives manipulant des structures de données. Notre procédure de réparation localise automatiquement diverses erreurs dans des fonctions récur­sives et les corrige en synthétisant une implémentation alternative.

Mots-clefs : programmation déclarative, synthèse de programme, procédure de synthèse, réparation de programme, analyse statique, effets de mémoire

Contents

Acknowledgements	i
Preface	iii
Abstract (English/Français)	v
List of figures	xv
Introduction	1
1 The Leon System	5
1.1 Verification in Leon	5
1.2 Web Interface	6
1.2.1 Live Code Updates	7
1.2.2 Verification	9
1.2.3 Execution	9
1.2.4 Synthesis	10
1.2.5 Repair	15
1.3 Symbolic Input/Output Tests	15
2 Deductive Synthesis	17
2.1 Examples	18
2.1.1 Tree Manipulations	18
2.1.2 List Manipulation	19
2.2 Formalism	26
2.3 Deductive Framework	28
2.3.1 Exploring the Space of Applications	29
2.3.2 Synthesis Rules	31
2.3.3 Cost Models	37
2.4 Symbolic Term Exploration	38
2.4.1 Grammars for Programs	38
2.4.2 From Grammars to Programs	43
2.4.3 Search Algorithm	46
2.4.4 Execution of Partial Programs	49
2.5 Ground-Term Generator	50

Contents

2.5.1	Generate and Test	50
2.5.2	Symbolic Input-Output Examples	51
2.5.3	Model-Based Enumeration	51
2.6	Evaluation	52
2.7	Related Work	54
2.8	Conclusions	58
3	Deductive Repair	59
3.1	Example	60
3.2	Deductive Guided Repair	62
3.2.1	Fault Localization	63
3.2.2	Guided Decompositions	65
3.2.3	Synthesis within Repair	65
3.3	Counterexample-Guided Similar-Term Exploration	65
3.4	Generating and Using Tests for Repair	66
3.4.1	Extraction and Generation of Tests	66
3.4.2	Classifying and Minimizing Traces	67
3.5	Evaluation	68
3.6	Further Related Work	70
3.7	Conclusions	71
4	Effect Analysis for Programs with Callbacks	73
4.1	Overview of Challenges and Solutions	75
4.2	Effect Analysis for Mutable Shared Structures	77
4.2.1	Intermediate Language Used for the Analysis	78
4.2.2	Effects as Graph Transformers	78
4.2.3	Composing Effects	80
4.2.4	Local Strong Updates	81
4.2.5	Application to the Analysis of Real-World Scala Code	82
4.3	Compositional Analysis of Higher-Order Code	83
4.3.1	Control-Flow Graph Summarization	84
4.3.2	Partial Unfolding	84
4.3.3	Combining Unfolding and Summarization	86
4.3.4	Controlled Delaying	87
4.3.5	Handling Recursion	87
4.3.6	Instantiation for Effect Graphs	88
4.3.7	Context Sensitivity	88
4.4	Producing Readable Effect Summaries	89
4.5	Evaluation on Scala Library	91
4.5.1	Overall Results	91
4.5.2	Comparative Analysis	93
4.5.3	Selected Examples	93
4.6	Related Work	95

4.7 Conclusion	97
Conclusions	99
A Synthesis Benchmarks and Solutions	101
A.1 Compiler	101
A.1.1 Header	101
A.1.2 Rewrite Implies	103
A.1.3 Rewrite Minus	104
A.2 List	104
A.2.1 Header	104
A.2.2 Insert	105
A.2.3 Delete	105
A.2.4 Merge	106
A.2.5 Diff	106
A.2.6 split	106
A.3 Sorted List	107
A.3.1 Header	107
A.3.2 Insert	108
A.3.3 Insert Always	109
A.3.4 Delete	109
A.3.5 Merge	110
A.3.6 Diff	110
A.3.7 Insertion Sort	111
A.4 Strictly Sorted Lists	111
A.4.1 Header	111
A.4.2 Insert	112
A.4.3 Delete	113
A.4.4 Merge	113
A.5 Unary Numerals	114
A.5.1 Header	114
A.5.2 Add	114
A.5.3 Distinct	115
A.5.4 Mult	115
A.6 Batched Queue	116
A.6.1 Header	116
A.6.2 Dequeue	117
A.7 Address Book	118
A.7.1 Header	118
A.7.2 Merge	119

B	Repair Benchmarks and Solutions	121
B.1	Compiler	121
B.1.1	Complete File	121
B.1.2	Desugar 1	126
B.1.3	Desugar 2	126
B.1.4	Desugar 3	127
B.1.5	Desugar 4	127
B.1.6	Desugar 5	128
B.1.7	Simplify	128
B.2	Heap	129
B.2.1	Complete File	129
B.2.2	Merge 1	131
B.2.3	Merge 2	132
B.2.4	Merge 3	132
B.2.5	Merge 4	133
B.2.6	Merge 5	133
B.2.7	Merge 6	133
B.2.8	Insert	134
B.2.9	Make Node	134
B.3	List	135
B.3.1	Complete File	135
B.3.2	Pad	142
B.3.3	++	142
B.3.4	:+	142
B.3.5	Replace	142
B.3.6	Count	143
B.3.7	Find 1	143
B.3.8	Find 2	144
B.3.9	Find 3	144
B.3.10	Size	145
B.3.11	Sum	145
B.3.12	-	145
B.3.13	Drop 1	146
B.3.14	Drop 2	146
B.4	Numerical	147
B.4.1	Complete File	147
B.4.2	power	147
B.4.3	ModDiv	148
B.5	Merge Sort	148
B.5.1	Complete File	148
B.5.2	Split	149
B.5.3	Merge 1	150

B.5.4 Merge 2	150
B.5.5 Merge 3	151
B.5.6 Merge 4	151
Bibliography	161
Curriculum Vitae	163

List of Figures

1.1	Example of a typical PureScala program that combines algebraic data-types and recursive abstraction functions. We notice a post-condition on the size function, specifying that its result is non-negative.	6
1.2	Overview of the Leon web interface. We can see the main editor pane where developers can write programs. The pane on the right displays live analysis results and updates automatically as the program changes.	7
1.3	Architecture of the web interface divided into components. Each component of the back end is an independent worker actor that runs asynchronously. Components receive code updates extracted and preprocessed into Leon trees, as well as commands sent from the interface. It provides feedback to the interface that updates its views.	8
1.4	When clicking on the invalid status in the right pane, we display verification details explaining why the function is invalid. We display the counter-example Leon discovered, as well as the resulting value when executing the function with this counter-example.	8
1.5	Exploring the execution of an invalid function with a discovered counter-example ($in1 = \text{Cons}(0, \text{Nil})$ and $v = -1$). This enables the developer to figure out which part of the post-condition is violated and why.	9
1.6	A synthesis pane is displayed whenever the input program contains a function that can be synthesized. The interface then offers manual and automated ways of synthesizing the function.	11
1.7	Searching will display a progress-bar showing the current status of the exploration within the search graph. It displays the number of nodes covered, as well as the total size of the graph. When a solution is found, we enable the developer to either import the resulting expression, or explore it to refine it.	12
1.8	Exploration displays the search tree as well as how the resulting expression is composed. For each node of the tree, the developer can choose an alternative rule to apply, consequently changing how the solution will look.	13
1.9	We display the progress for the repair attempt in a dedicated modal. When successful, we present the alternative implementation that satisfies the specification and we enable the developer to import it in place of the old code.	14

List of Figures

1.10	Partial specifications using the passes construct, enabling the matching of more than one input and providing the expected output as an expression. In the case of reverse, a single symbolic test is sufficient to fully specify the behavior of the function on all lists of size three.	15
2.1	Expression trees with an evaluation function that returns the value corresponding to a given (ground) expressions. Note that our evaluation function <i>eval</i> is total: it returns <i>Error</i> when it gets stuck. We then define two functions that we want to implement. They both rewrite an expression tree into an different shape, that should be equivalent with respect to our evaluation function.	20
2.2	User-defined list structure with the usual size and content abstraction functions. Here and throughout the chapter, the content abstraction computes a <i>set</i> of elements, but it can easily be extended to handle multisets (bags) by using the same techniques [dMB09] if stronger contracts are desired	21
2.3	Specification of sorting suitable for insertion sort	23
2.4	Synthesized insertion sort for Figure 2.3	23
2.5	Specification of removal from a sorted list.	24
2.6	Implementation synthesized for Figure 2.5	24
2.7	Normalizing rules for particular problems and specifications.	33
2.8	Normalizing rules for algebraic data-types.	34
2.9	Detupling for input and output variables.	34
2.10	Rules introducing conditional expressions.	36
2.11	Solver-based heuristics	37
2.12	Partial search-graph that leads to a deep invocation of STE. This graph corresponds to a top-level invocation of ADT-Split on the <i>in</i> variable. Then, we invoke Equality-Split between <i>v</i> and <i>h</i> on the sub-problem corresponding to the <i>Cons</i> case. Finally, we invoke STE in the sub-problem for when <i>h</i> is equal to <i>v</i>	43
2.13	Partial solution around an invocation to STE, corresponding to the search-graph displayed in Figure 2.12 (with P_{Nil} solved with $\langle T \mid Nil \rangle$ and P_{\neq} unsolved).	44
2.14	Functions generated to represent the grammar $G_{[3]}$. The non-terminal symbol $BigInt_{[1]}$ is represented by two independent functions, because it can be used twice at the same time in $BigInt_{[3]}$	45
2.15	Search algorithm.	47
2.16	Automatically synthesized functions using our system. We consider a problem as synthesized if the solution generated is correct after manual inspection. For each generated function, the table lists the size of its syntax tree and the number of function calls it contains. \checkmark indicates that the system also found a proof that the generated program matches the specification: in many cases proof and synthesis are done simultaneously, but in rare cases merely a large number of automatically generated inputs passed the specification. The final column shows the total time used for both synthesis and verification.	53

3.1	The syntax tree translation in function desugar has a strong ensuring clause, requiring the semantic equivalence of the transformed and the original tree, as defined by several recursive evaluation functions. desugar contains an error. Our system finds it, repairs the function, and proves the resulting program correct.	61
3.2	Code and invocation graph for desugar. Solid borderlines stand for passing tests, dashed ones for failing ones. Type constructors for literals have been omitted.	67
3.3	Automatically repaired functions using our system. We provide for each operation a small description of the kind of error introduced, the overall program size, the size of the invalid function, the size of the erroneous expression we locate, and the size of the repaired version. We then provide the times our tool took to gather and classify tests, and to repair the erroneous expression. Finally, we mention if the resulting expression can be verified. The source of all benchmarks can be found in the appendix.	69
4.1	Program statements \mathcal{P} considered in the target language.	78
4.2	Example of a graph representing the effects of prepend. Read edges lead to load nodes that represent unknown values, and solid edges represent field assignments.	79
4.3	Merging a graph with load nodes and strong updates in a context that does not permit a strong update. Inside nodes are imported after refining their label.	81
4.4	Example of strong update being delayed as much as possible	81
4.5	Example of unfolding with $T_{call} = \{C_1.m, C_2.m\}$.	85
4.6	Example of a chain of method calls.	86
4.7	Transformation steps from an effect-graph to a minimized DFA. The graph on the left is the <i>definite</i> effect of an impure list traversal. The center graph is the corresponding NFA whose accepting language represents paths to modified fields. The last graph is the minimized DFA to be translated to a regular expression.	90
4.8	Decomposition of resulting summaries per package.	92
4.9	Comparing strategies. Numbers below boxes are percentages. Running times are 166, 123 and 57 minutes respectively.	93
4.10	The particular four operations applied on the TreeSet collection	94
4.11	Readable effect descriptions obtained from graph summaries from four operations performed on five kinds of collections.	95

Introduction

Ensuring the trustworthiness of a software application requires two key insights: knowing what it should do, and understanding what it currently does. Any mismatch between the two represents flaws that can have dramatic repercussions. In this work, we explore multiple approaches to bridge the gap between our expectations of an application and their realizations.

Today, writing software remains a largely manual effort, despite significant progress in software development environments and tools. By exposing high-level constructs, modern programming languages relieve developers from writing boiler-plate code and other error-prone details. This trend can also be seen in the renewed appeal for domain-specific languages: experts design and implement high-level building blocks that are then composed by users. Combining high-level constructs and rich type systems, as it is the case in Scala, already prevents many programming errors. But most of the intended behavior of software applications is left implicit: we expect, for instance, a list-sorting function to return a list that is sorted, but this specification is often not stated explicitly. In fact, most popular programming languages do not offer a way to specify the behavior of the program, as they would not be able to exploit them anyway. These high-level description are often relegated to comments.

Software engineers have therefore developed many ad-hoc techniques to increase the reliability of their codebase. Common approaches include coding guidelines, human code-reviews and mechanical testing. The extent to which these techniques are applied generally depends on the cost of a potential failure. For safety-critical applications such as for flight equipment, constraining rules are well established and enforced by regulations. These typically include a correctness certification: a strong argument or proof that the application behave as expected under all circumstances.

For general-purpose software, the correctness of their behavior is mostly asserted by runtime tests. These tests cover the aspects of the application that are deemed relevant by the testers and developers. Although testing usually provides a good return on investment in terms of quality, it does not give strong guarantees. It is difficult to establish a good metric for software reliability, but the high number of bugs and security issues that are continuously discovered indicates that commonly-used software still contains many preventable issues, although mostly in edge-cases. This observation is also true for critical security software that has been manually studied and thoroughly tested. We thus believe that the situation can be greatly

improved by enabling the mainstream use of verification. One of the strengths of software verification is that, by considering the entire set of possible inputs, it does not discriminate against edge-cases. Whereas the construction of arbitrarily complex verified software is possible in principle, verifying programs after they have been developed is extremely time-consuming [LS09, KAE⁺09] and it is difficult to argue that it is cost-effective.

At the root of our research stands the Leon verifier for purely-functional Scala programs [Sut12b]. Leon quickly detects errors in functional programs by reporting concrete counterexamples and can also prove the correctness of programs [BKKS13, Sut12a, SDK10, SKK11]. An important aspect of the tool is its modularity: The verification of an individual function against its specification can begin before the entire software system is completed and the resulting verification tasks are partitioned into small pieces. As a result, it can provide rapid feedback that enables specifications and implementations to be developed simultaneously.

In this thesis, we explore multiple approaches that aid the development of verified software. Although the verifier itself is a key component, we believe the developers can also benefit from techniques for *synthesis* and *repair* from specifications. Specifications in terms of relational properties generalize existing declarative programming language paradigms by enabling the statement of *constraints* between inputs and outputs [JL87, KKS12] as opposed to always specifying outputs as functions from input to outputs. Unlike deterministic implementations, constraints can be composed using conjunctions, which enables the problem to be described as combination of orthogonal requirements.

The purely-functional subset of Scala offers interesting properties that Leon is able to exploit. Any non-trivial applications are however unlikely to rely solely on purely functional features of Scala. Whereas important parts of the application logic may be expressed in functional style, dependencies to user-interactions or global state will be implemented by non-functional code. We believe Leon can also be useful for these hybrid applications: it can modularly analyze the functional components, assuming certain properties of the non-functional fragment. For this purpose, we enabled Leon to support the definition of arbitrary Scala functions accompanied by a specification. We also developed an effect system for Scala that characterizes the behavior of arbitrary Scala code. The resulting effects provide an overview of the behavior of selected functions, which aids program understanding.

Contributions and Outline

In this thesis, we explore synthesis and repair techniques for the integrated development of verified programs.

- We describe a deductive synthesis framework that takes a given set of synthesis rules and applies them according to a cost function. The framework accepts a synthesis problem as a relational specification between input and output variables. It returns the function

from inputs to outputs as a solution. Within the above framework we implement rules for synthesis of algebraic data-type equations and disequations [Sut12a], as well as a number of general rules for decomposing specifications based on their logical structure or case splits on commonly useful conditions. We evaluate the current reach of our synthesizer in fully automated mode by synthesizing recursive functions on nested algebraic data-types, including those that perform operations on lists defined by using a general mechanism for algebraic data-types, as well as on other custom data-types (e.g. trees) defined by the user.

- One of the main strengths in our framework is a new form of counterexample-guided synthesis. To generate bodies of functions, we have symbolic term generators that systematically generate well-typed programs built from a selected set of operators (such as algebraic data-type constructors and selectors). To test candidate terms against specifications, we use the Leon verifier. To accelerate this search, the rule accumulates previously found counter-examples. Moreover, to quickly bootstrap the set of examples it uses systematic generators that can enumerate, in a fair way, any finite prefix of a countable set of structured values. The falsification of generated bodies is done by a direct execution of code. For this purpose, we developed a lightweight bytecode compiler for our functional implementation language (a subset of Scala), allowing us to use code execution as a component of counterexample search in the constraint solver.
- We present an automatic algorithm that *repairs* invalid functions by returning a valid alternative implementation. We implement this algorithm inside the synthesis framework with rules tailored for repair. The algorithm combines techniques for fault localization as well as the exploration of *similar programs*.
- We deploy the synthesis framework with repair in a web-browser-based environment with continuous compilation, highlighting the modular aspect of our techniques. The synthesis framework can be interrupted at any point to yield a partial solution with a possibly simpler synthesis problem. The repair algorithm can be invoked to automatically fix invalid functions. The search space can be interactively explored.
- We describe several ways of integrating Leon functions and data-structures within Scala applications, and discuss multiple ways of enabling interactions between them. We discuss ways of accounting for the potentially non-deterministic behavior of arbitrary Scala code.
- In order to detect and account for unpredictable behaviors in arbitrary Scala code, we present the design, implementation, and evaluation of a new static analysis for method side-effects: this analysis is precise and scalable even in the presence of callbacks, including higher-order functions. The key design aspects of our analysis include a relational analysis domain that computes summaries of code blocks and methods by flow-sensitively tracking side-effects and performing strong updates and an automated effect classification and presentation of effect abstractions in terms of regular expressions, thus facilitating their understanding by developers.

List of Figures

The following chapters are organized as follows:

Chapter 1 introduces the Leon verifier and its target language: PureScala. We provide an overview of the Leon system as implemented for its web-interface.

Chapter 2 presents the deductive synthesis framework. We describe the problem of synthesizing programs from specification and provide the complete set of deductive rules implemented in our system.

Chapter 3 describes the repair algorithm. We implement the search for the alternative implementation as an extension of our synthesis framework.

Chapter 4 describes the design and implementation of an analysis that characterizes the memory effects of arbitrary Scala functions. The effect system is tailored to handle higher-order constructs that are ubiquitous in Scala programs.

1 The Leon System

In this thesis, we explore several techniques to help developers build verified Scala software. We build on the Leon verification system, that provides tools for reasoning about programs written in purely-functional Scala (PureScala). Leon supports most functional features of the Scala language, which notably include algebraic data-types, recursive functions and higher-order constructs. In addition to this purely functional fragment, Leon also supports some imperative features such as mutable local variables, while loops or some form of mutable arrays. Leon handles these constructs by first translating them to their functional counterparts.

1.1 Verification in Leon

In its library, Scala defines a notation [Ode10] for specifying runtime checks in the form of assertions, pre-conditions, and post-conditions. Leon reuses the same notation and extracts the specifications for our functions. We then verify that these specifications are satisfied in all scenarios. We show in Figure 1.1 a typical example of a PureScala program with a specification on size.

At the core of the Leon verification procedure is an algorithm for reasoning about formulas that refer to user-defined recursive functions. Our goal is to check the satisfiability of such formulas and possibly generate models. This enables us to generate verification conditions that indicate the presence of an error. In the case of size, the verification condition looks as follows:

$$\exists l. \text{size}(l) < 0$$

On one hand, being able to satisfy this formula and finding a model for l indicates that the specification of size is violated for this valuation of the input variable. Satisfying the verification condition is thus equivalent to finding counter-examples. On the other hand, we can prove the correctness of the function (with respect to its post-condition) by determining the unsatisfiability of the verification condition.

```
abstract class List[T]
case class Cons[T](h: T, t: List[T]) extends List[T]
case class Nil[T]() extends List[T]

def size[T](lst: List[T]): BigInt = {
  lst match {
    case Cons(h, t) => size(t) + BigInt(1)
    case Nil()      => BigInt(0)
  }
} ensuring { res =>
  res ≥ 0
}
```

Figure 1.1 – Example of a typical PureScala program that combines algebraic data-types and recursive abstraction functions. We notice a post-condition on the size function, specifying that its result is non-negative.

The algorithm proceeds by iteratively examining longer and longer execution traces through the recursive functions. It alternates between an over-approximation of the executions, where only unsatisfiability results can be trusted, and an under-approximation, where only satisfiability results can be concluded. The status of each approximation is checked using state-of-the-art SMT solvers such as Z3 from Microsoft Research [dMB08] and CVC4 [BCD⁺11]. Leon communicates with these solvers by using the SMT-LIB interface. The algorithm is a *semi-decision procedure*, meaning that it is theoretically complete for counter-examples: if a formula is satisfiable, Leon will eventually produce a model [SKK11]. Additionally, the algorithm works as a decision procedure for a certain class of formulas [SDK10]. However, the algorithm might never terminate for some formulas. We therefore equip it with a timeout, guaranteeing that it will eventually return with either SAT, UNSAT, or UNKNOWN.

In the past, we have used this core algorithm in the context of verification [SKK11], but also as part of an experiment in providing run-time support for declarative programming that uses constructs similar to **choose** [KKS12]. In both cases, we find the performance in finding models to be suitable for the task at hand. Throughout this thesis, we will use this algorithm extensively within synthesis and repair for deciding a variety of automatically generated formulas.

1.2 Web Interface

In this work, we explore ways of improving the process of developing verified software. It is thus only natural to lift the original command-line-based incarnation of the Leon system to an integrated development environment that permits more advanced interactions with the developers. In order to facilitate its access, we deploy this environment as a public web interface. We show an overview of the interface in Figure 1.2. We describe here some of the main components of our web interface and how features, such as synthesis and repair, are deployed.

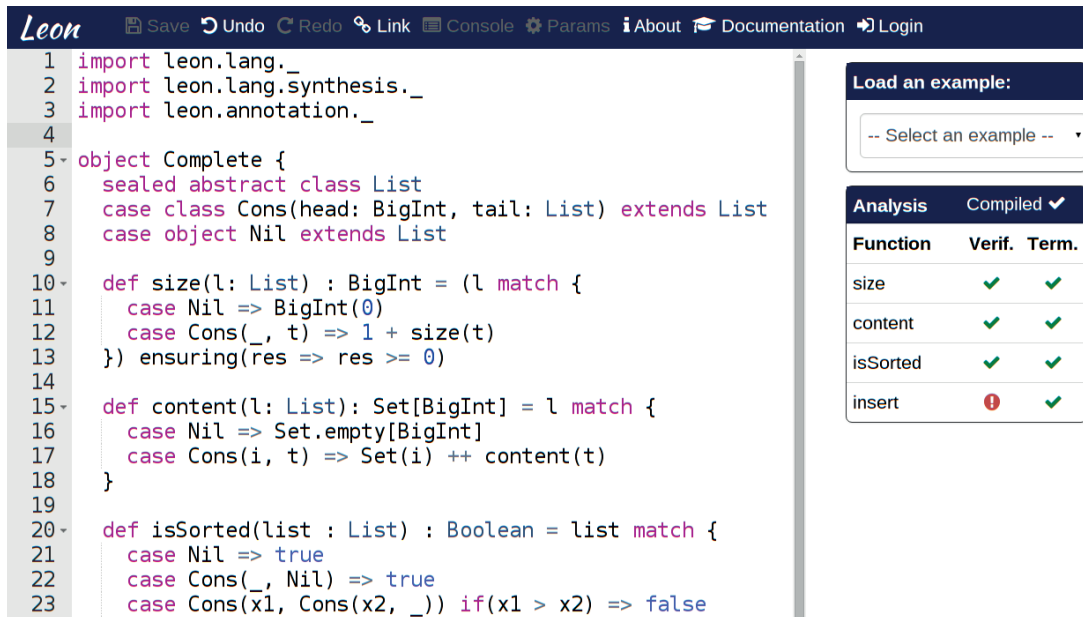


Figure 1.2 – Overview of the Leon web interface. We can see the main editor pane where developers can write programs. The pane on the right displays live analysis results and updates automatically as the program changes.

The interface is organized into independent components which enables multiple aspects of Leon to co-exist in the same interface. We show in Figure 1.3 a brief overview of the architecture of the interface. The Javascript front end communicates with the back end using a web-socket, which enables bidirectional transfer of events and messages. Each component runs in an individual worker actor that is spawned for each client. The extraction component receives code updates from the front end, and dispatches the corresponding Leon trees to all workers. These workers also receive commands directly from the front end, and reply with feedback that translates into modification of the views of the interface. Workers are able to communicate between them as well. For instance, the execution workers gets new tasks to evaluate as soon as the verification worker discovers new counter-examples.

1.2.1 Live Code Updates

As it is typically the case with development environments, we automatically recompile the program as it updates. New versions of the program are sent to the server that recompiles it. Like standard development environments, compilation warnings and errors are immediately displayed in their context, at the position at which they occur. We rely on a dependency graph coupled with a hash-based way of identifying functions across compilations to only invalidate meta-data associated with functions that have been modified either directly or through one of its dependency. This enables past results to be kept in the cache if the modifications done by the developer do not affect it.

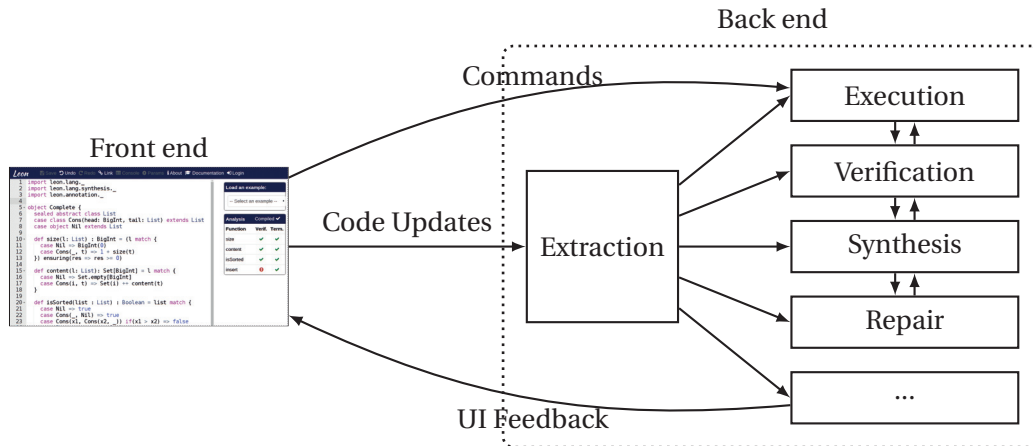


Figure 1.3 – Architecture of the web interface divided into components. Each component of the back end is an independent worker actor that runs asynchronously. Components receive code updates extracted and preprocessed into Leon trees, as well as commands sent from the interface. It provides feedback to the interface that updates its views.

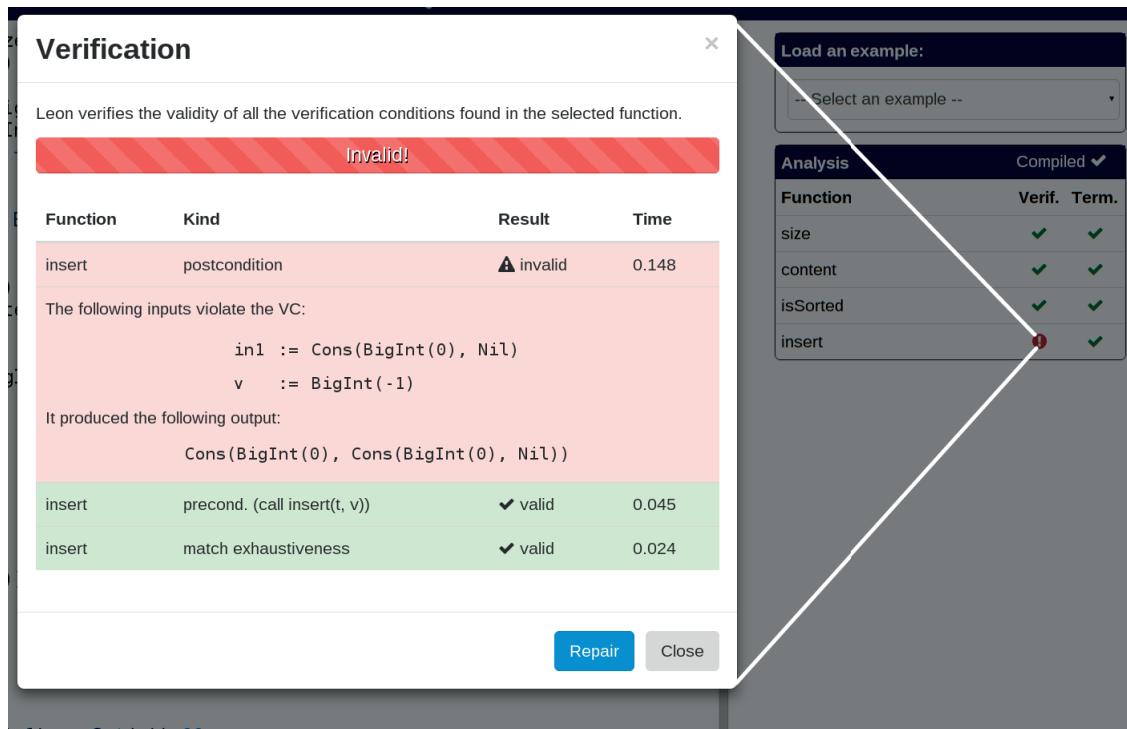


Figure 1.4 – When clicking on the invalid status in the right pane, we display verification details explaining why the function is invalid. We display the counter-example Leon discovered, as well as the resulting value when executing the function with this counter-example.

1.2.2 Verification

Leon automatically verifies functions with specifications and displays the results as soon as they are computed in a pane on the right. By clicking on the status, the developer obtains additional details about the verification results, as pictured in Figure 1.4. In the case of an invalid function, we display the counter-example found by Leon, as well as the resulting value that the function generated for these inputs.

```

27- def insert(in: List, v: BigInt): List = {
28-   require(isSorted(in))
29-   in match {
30-     case Cons(h, t) =>
31-       if (v < h) {
32-         Cons(h, in)
33-       } else if (v == h) {
34-         in
35-       } else {
36-         Cons(h, insert(t, v))
37-       }
38-     case Nil =>
39-       Cons(v, Nil)
40-   }
41- }
42- } ensuring { res => false
43-   (content(res) == content(in) ++ Set(v)) &&
44-   isSorted(res)
45- }
```

Figure 1.5 – Exploring the execution of an invalid function with a discovered counter-example ($in1 = \text{Cons}(0, \text{Nil})$ and $v = -1$). This enables the developer to figure out which part of the post-condition is violated and why.

1.2.3 Execution

The interface comes with a tracing evaluator that records intermediate evaluation results. This enables the evaluation of functions to be displayed partially, at the command of the developer. We identify two main types of use of evaluation: parameterless functions, that are naturally executable, and invalid functions, evaluated using one of their discovered counter-example. This exploration of the evaluation results enables the developer to collect information about the produced result and about why it violates the specification.

Concretely, the tracing evaluator returns the list of values for all intermediate expressions, abstracted by their range positions. Note that this tracing is done only for the top-level execution frame; we therefore do not record intermediate results within recursive calls or invocations of other functions, because these results would be hard to interpret when displayed within the interface.

We take the simple expression " $a + b * c$ " as example. Given $a = 1$, $b = 2$, and $c = 3$, our

Chapter 1. The Leon System

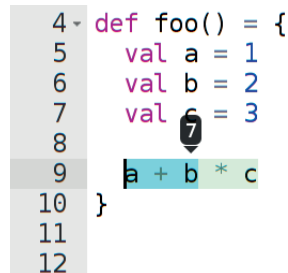
evaluator generates the following information (in the interest of clarity, we reduce the positions to their column only; our technique naturally extends to lines as well):

a	:	(0, 1)	→	1
b	:	(4, 5)	→	2
c	:	(8, 9)	→	3
b * c	:	(4, 9)	→	6
a + b * c	:	(0, 9)	→	7

We then extract the range corresponding to the selection done by the developer in the editor. When no explicit selection is made, we take the current cursor position as an empty range. Next, we match this range to the best candidate within the evaluator results. For this we define a difference function for ranges that computes how well a given range matches the selection.

This enables us to find the range that has a minimal difference with the range selected by the user. For instance, given the selection `[a + b] * c`, which corresponds to (0, 6), we select the result of the full expression. We favor ranges that occur within bigger expressions over ranges that are beside it: The selection `a + b[] * c` therefore matches the expression `b * c` and not `b`.

Finally, we display a tooltip with the result and highlight the expression for which the result corresponds to



```
4 def foo() = {  
5   val a = 1  
6   val b = 2  
7   val c = 3  
8  
9   a + b * c  
10 }  
11  
12
```

Note that we apply a cut-off so that we do not display the closest range if its too far away.

We believe this exploration is especially useful when investigating invalid functions, as it enables the developer to not only see the inputs and the generated outputs, but also which part of the specification was violated. We illustrate this with a function that inserts into a sorted list in Figure 1.5. Because the function is invalid, we automatically execute it with the discovered counter-example. By selecting part of the specification, we are able to see why the result is considered invalid.

1.2.4 Synthesis

The synthesis framework described in Chapter 2 can be invoked whenever the program contains either synthesis holes (??? within a specified function), or the **choose** construct.

```
def content(l: List): Set[BigInt] = l match {
  case Nil => Set.empty[BigInt]
  case Cons(i, t) => Set(i) ++ content(t)
}

def delete(inl: List, v: BigInt) = {
  choose { (out : List) =>
    content(out) == content(inl) -- Set(v)
  }
}
```

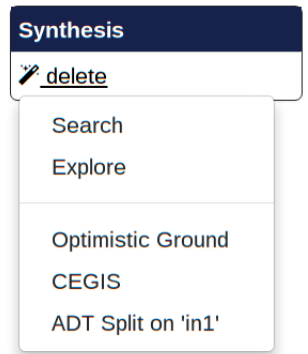


Figure 1.6 – A synthesis pane is displayed whenever the input program contains a function that can be synthesized. The interface then offers manual and automated ways of synthesizing the function.

In Figure 1.6, we find a **choose** construct for the problem of deleting from a list. This produces an extra synthesis pane on the right where we can invoke the procedure. By clicking on the synthesis problem, we are presented with several options: We can either let the procedure search for a solution, as described in Section 2.3.1, or manually explore the invocations of the deductive rules available.

Automated Synthesis

Leon can search for solutions by exploring the search graph automatically. Upon finding a solution, it displays the synthesized expression and enables the developer to either import the solution code to replace the **choose**, or to explore the solution and possibly refine it. We show a successful synthesis modal for the problem of deleting from a list, in Figure 1.7.

Refactoring Steps and Exploration

The graph representing the search for solutions can be explored interactively in two ways: The developers can apply individual rules and obtain the resulting code, where sub-problems have become more specific **choose** constructs. This corresponds to applying development steps that help the developer write repetitive code such as pattern-matching expressions or other conditionals. Another way is to navigate through the search tree by choosing, at each point which rule instantiation to apply. This is illustrated in Figure 1.8. The developer chooses what rule to apply for each synthesis problem. When selecting a particular rule, the resulting partial program is displayed, and sub-problems are again represented by choices.

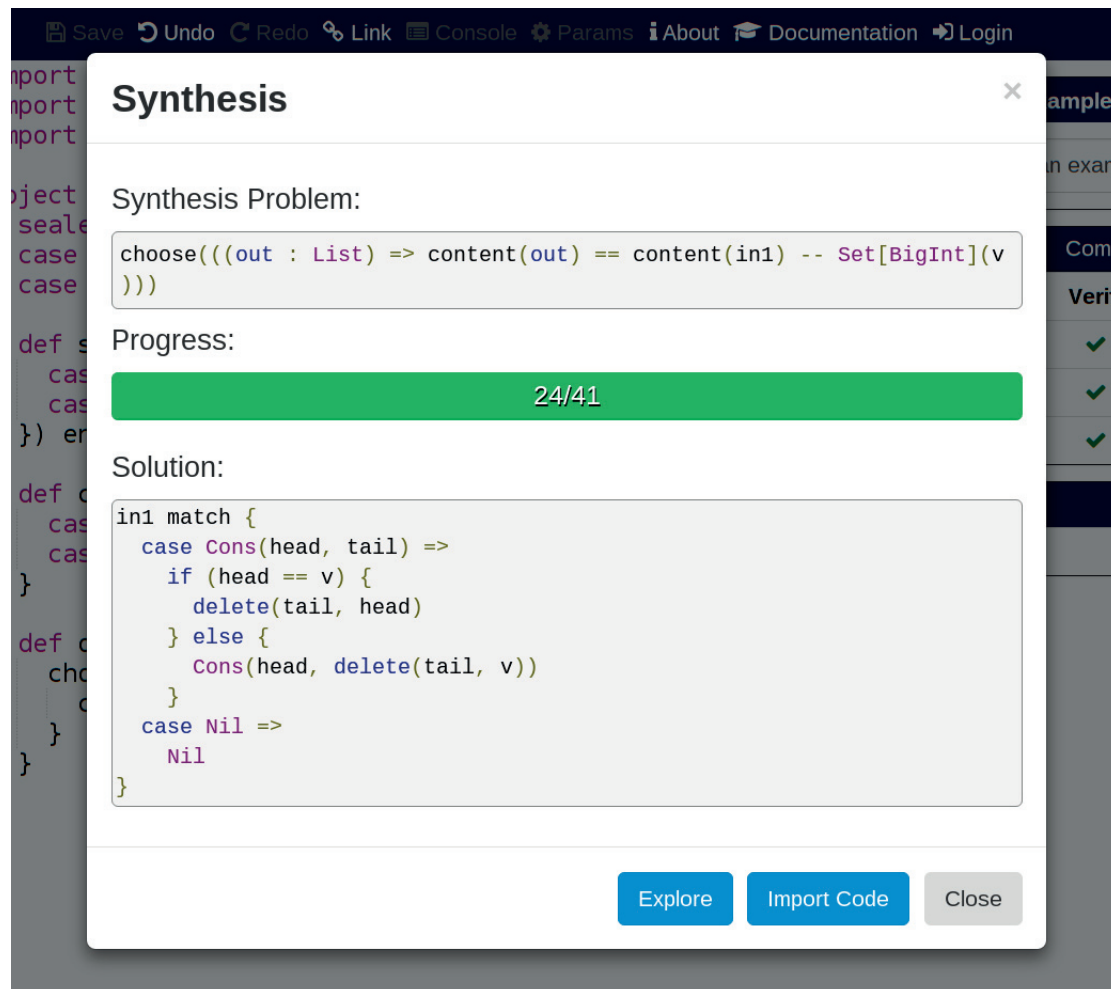


Figure 1.7 – Searching will display a progress-bar showing the current status of the exploration within the search graph. It displays the number of nodes covered, as well as the total size of the graph. When a solution is found, we enable the developer to either import the resulting expression, or explore it to refine it.

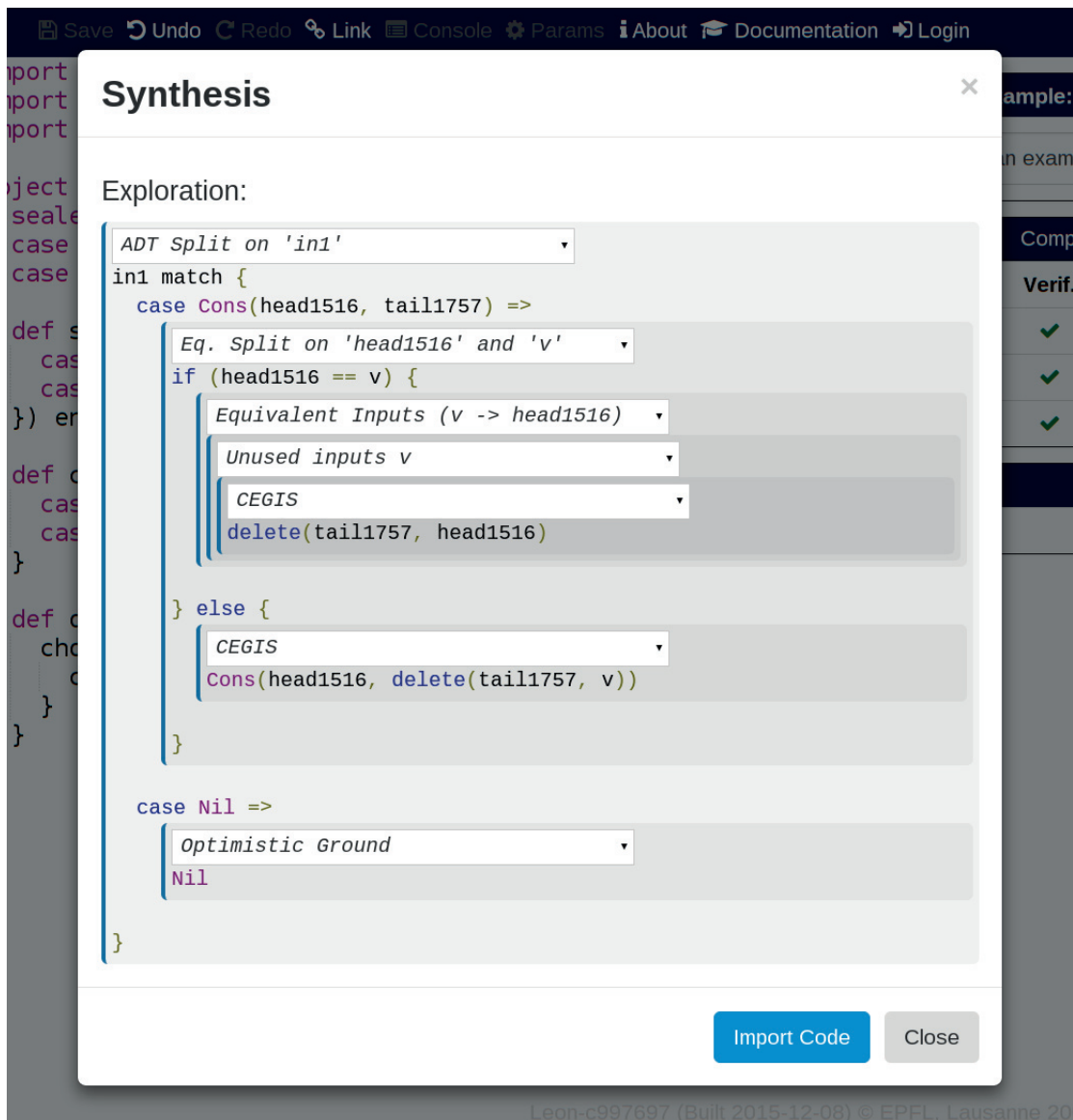


Figure 1.8 – Exploration displays the search tree as well as how the resulting expression is composed. For each node of the tree, the developer can choose an alternative rule to apply, consequently changing how the solution will look.

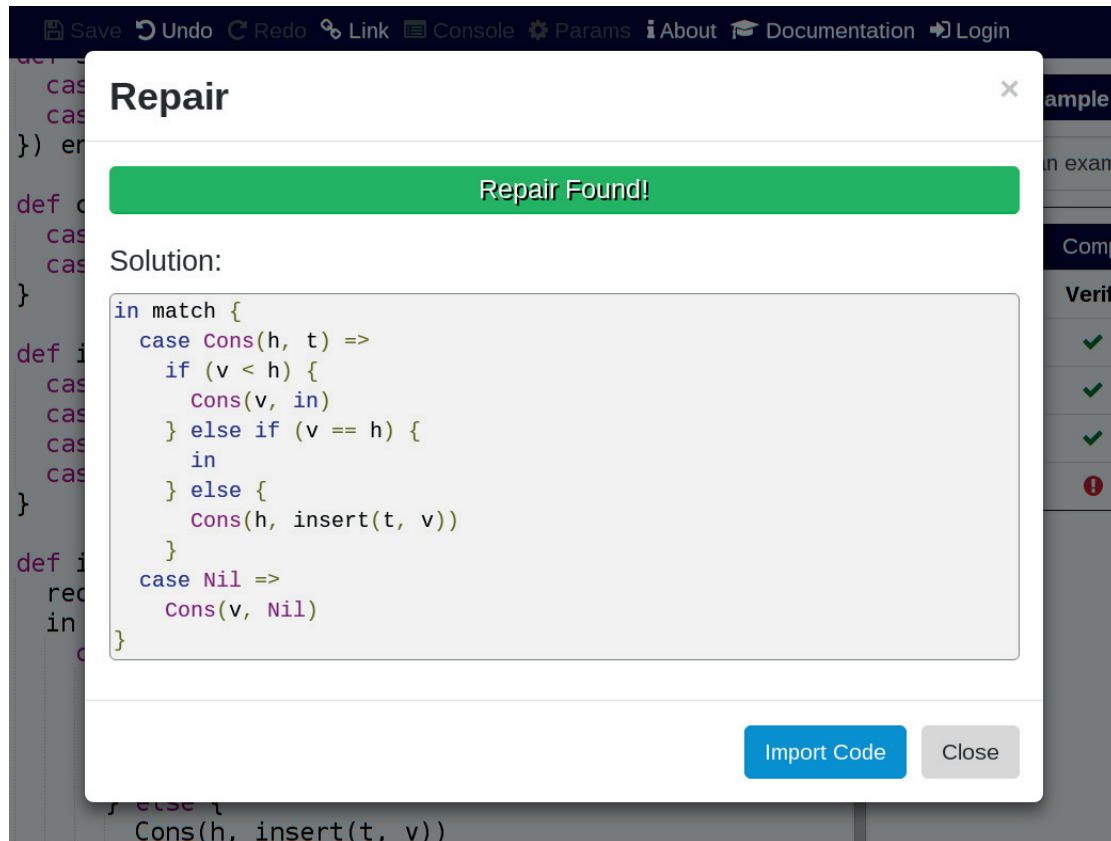


Figure 1.9 – We display the progress for the repair attempt in a dedicated modal. When successful, we present the alternative implementation that satisfies the specification and we enable the developer to import it in place of the old code.

1.2.5 Repair

To enable the developers to automatically repair invalid functions, we implement within our web interface the repair techniques described in Chapter 3. As you can see in Figure 1.4, we expose a *repair* button whenever a function is detected as invalid. Clicking the button will open a subsequent pane in which it describes the progress of the repair attempt. When successful, we show the alternative implementation that satisfy the specification, as displayed in Figure 1.9. We enable the developer to immediately import the new version in place of the invalid code, similarly to synthesis.

1.3 Symbolic Input/Output Tests

It is often interesting for the user to specify how a function behaves by listing a few examples of inputs and corresponding outputs. However, having to provide full inputs and outputs can be tedious and impractical. To make specifying families of tests convenient, we define a *passes* construct to express input-output examples. We rely on pattern matching in our language to symbolically describe sets of inputs and their corresponding outputs. This gives us an expressive way of specifying classes of input-output examples. Not only the pattern can match more than one input, but the corresponding outputs are given by an expression that depends on the pattern's variables. This puts our symbolic input/output at the frontier between traditional input/output tests and full-blown relational specifications. Wildcard patterns are particularly useful when the function does not depend on all aspects of its inputs. For instance, a function that reverses a generic list does not inspect the values of individual list elements. Similarly, the function computing the power of two integers could be partially specified by providing a few cases. Both examples are illustrated in Figure 1.10.

```
def reverse[T](l: List[T]): List[T] = {
  // ...
} ensuring { res =>
  (l, res) passes {
    case h1 :: h2 :: h3 :: Nil() =>
      h3 :: h2 :: h1 :: Nil()
  }
}

def power(a: BigInt, b: BigInt) = {
  // ...
} ensuring { res =>
  ((a, b), res) passes {
    case (a, 0) if a != 0 => 1
    case (1, b) => 1
    case (a, 1) => a
    case (2, 10) => 1024
  }
}
```

Figure 1.10 – Partial specifications using the *passes* construct, enabling the matching of more than one input and providing the expected output as an expression. In the case of *reverse*, a single symbolic test is sufficient to fully specify the behavior of the function on all lists of size three.

Although our *passes* construct does not add any expressive power to our relational specifications, it offers a concise and predictable notation from which we can extract and generate

concrete input/output tests, as we will see in Section 2.5.2 and Section 3.4.1.

Our passes constructs are in a way similar to external testing functions in the framework of parameterized unit tests [TS05] (PUT). Our approach allows the test to accompany the function definition, and the structure of passes enables us to efficiently extract pairs of input-output tests, because inputs-conditions are clearly separated from outputs-computations. In the PUT framework, they rely on symbolic execution to extract inputs-conditions, necessary to then generate valid input-output pairs.

2 Deductive Synthesis

In this chapter, we build on the Leon verifier described in Chapter 1 to introduce one of the principal contributions of this thesis: synthesis algorithms, techniques and tools that integrate synthesis into the development process for functional programs. We present a synthesizer that, starting solely from their contracts, can construct the bodies of functions.

The programs that our synthesizer produces typically manipulate unbounded data types, such as algebraic data types and unbounded integers. Due to the use of deductive synthesis and the availability of a verifier, when the synthesizer succeeds with a verified output, the generated code is correct for all possible input values.

Our synthesizer uses specifications as the description of the synthesis problems. Even though it can additionally accept input/output examples to illustrate the desired functionality, we view such illustrations as a special form of input/output relation: as input/output examples correspond to tests and provide a description of a finite portion of the desired functionality, we primarily focus on symbolic descriptions, which ensures the desired behavior over an arbitrarily large or even infinite domain. From such descriptions, our synthesizer can automatically generate input/output examples when needed, but can also directly transform specifications into executable code.

We integrate our synthesizer into Leon, where it tightly cooperates with the underlying verifier, enabling it to achieve performance orders of magnitude better than when using simpler generate-and-test approaches. The techniques we use include symbolic transformation based on synthesis procedures, as well as synthesis of recursive functions by using counterexample-guided strategies. We evaluate a number of system architectures and trade-offs between symbolic and concrete reasoning in our implementation and arrive at an implementation that appears successful, despite the large space of possible programs. We therefore believe we have achieved a new level of automation for a broad domain of recursive functional programs. We consider that a particular strength of our system is that it can synthesize code that satisfies a given relational specification for all values of inputs, and not only given input/output pairs.

Our techniques aim at a high automation level, but we are aware that any general-purpose automated synthesis procedure will ultimately face limitations: the developers may want to synthesize code larger than the scalability of automated synthesis permits, or they may want to control not only the observational behavior but also the structure of the code to be constructed. We therefore deploy the synthesis algorithm as an interactive assistance that enables the developer to interleave manual and automated development steps. In our system, the developer can decompose a function and leave the subcomponents to the synthesizer; or, conversely, the synthesizer can decompose the problem, solve some of the subproblems, and leave the remaining open cases for the developer. To facilitate such synergy, we deploy an anytime synthesis procedure, that maintains a ranked list of current problem decompositions. The developers can interrupt the synthesizer at any time to display the current solution and continue manual development.

The approach to synthesis we follow in this work is for deriving programs by a succession of independently validated steps. We exploit a previous (purely theoretical) version of the framework in [JKS13]; the new framework supports the notion of path condition and is the first time we report on the practical realization of this framework.

2.1 Examples

We begin by illustrating through a series of examples how developers use our system to write programs that are correct by construction. We first illustrate our system and the nature of the interactions with it, through operations on expression trees and (sorted) user-defined lists, thus reflecting along the way on the usefulness of programming with specifications.

2.1.1 Tree Manipulations

We begin our overview with an example manipulating abstract-syntax trees, similarly to what you would find in a toy compiler. The trees represent operations on booleans as well as mathematical integers.

To provide meaning to the trees, the developer defines in Figure 2.1 the untyped expression trees, as well as an evaluation function that provides implicit meaning to trees. In case of type mismatch, the evaluation function `eval` returns an error value. This enables the developer to write semantics-preserving transformation functions.

In Figure 2.1, we also define two rewrite-functions that we want to implement; They take a certain tree as input and are expected to return a different tree that should nonetheless be equivalent to their input, with respect to `eval`.

In about three seconds, Leon synthesizes the following two functions:

```
def rewriteMinus(in: Minus): Expr = Plus(in.lhs, UMinus(in.rhs))
```

```
def rewritemplies(in: Minus): Expr = Or(Not(in.lhs), in.rhs)
```

What is interesting here is that even though these transformations are somewhat predictable and appear trivial to most developers, they fully depend on the provided evaluation function. Given less natural evaluation semantics, the required transformation is likely to be less predictable as well. However, our synthesizer would be equally good at finding it.

2.1.2 List Manipulation

We show how our system behaves when we synthesize operations on lists. The developer partially specifies lists by using their effect on the set of elements. As shown in Figure 2.2, we start from a standard recursive definition of lists, along with recursive functions that compute their size as a non-negative integer and their content as a set of integers.

Splitting a list

We first consider the task of synthesizing the split function as used in, e.g., merge sort. As a first attempt to synthesize split, the developer can try the following specification:

```
def split(lst: List): (List, List) = {
  choose { (r: (List, List)) =>
    content(lst) == content(r._1) ++ content(r._2)
  }
}
```

Because it tends to generate simpler solutions before more complex ones, Leon here instantly generates the following function:

```
def split(lst: List): (List, List) = (lst, Nil)
```

Although it satisfies the contract, it is not particularly useful. This shows the difficulty in using specifications, but the advantage of a synthesizer such as ours is that it enables the developer to quickly refine the specification and obtain a more desirable solution. To avoid obtaining a single list, together with an empty one, the developer refines the specification by enforcing that the sizes of the resulting lists should not differ by more than one:

```
def split(lst: List): (List, List) = {
  choose { (r: (List, List)) =>
    content(lst) == content(r._1) ++ content(r._2)
    && abs(size(r._1) - size(r._2)) ≤ 1
  }
}
```

Again, Leon instantly generates a correct, useless, program:

```
def split(lst: List): (List, List) = (lst, lst)
```

```
abstract class Expr
case class Plus(lhs: Expr, rhs: Expr) extends Expr
case class Minus(lhs: Expr, rhs: Expr) extends Expr
case class UMinus(e: Expr) extends Expr
case class LessThan(lhs: Expr, rhs: Expr) extends Expr
case class And(lhs: Expr, rhs: Expr) extends Expr
case class Implies(lhs: Expr, rhs: Expr) extends Expr
case class Or(lhs: Expr, rhs: Expr) extends Expr
case class Not(e: Expr) extends Expr
case class Eq(lhs: Expr, rhs: Expr) extends Expr
case class Ite(cond: Expr, thn: Expr, els: Expr) extends Expr
case class BoolLiteral(b: Boolean) extends Expr
case class IntLiteral(i: BigInt) extends Expr
```

```
abstract class Value
case class BoolValue(b: Boolean) extends Value
case class IntValue(i: BigInt) extends Value
case object Error extends Value
```

```
def eval(e: Expr): Value = e match {
  case Plus(l, r) =>
    (eval(l), eval(r)) match {
      case (IntValue(il), IntValue(ir)) => IntValue(il+ir)
      case _ => Error
    }
}
```

```
// ~100 lines...
```

```
}
```

```
def rewriteMinus(in: Minus): Expr = {
  choose{ (out: Expr) =>
    eval(in) == eval(out) && !(out.isInstanceOf[Minus])
  }
}
```

```
def rewriteImplies(in: Implies): Expr = {
  choose{ (out: Expr) =>
    eval(in) == eval(out) && !(out.isInstanceOf[Implies])
  }
}
```

Figure 2.1 – Expression trees with an evaluation function that returns the value corresponding to a given (ground) expressions. Note that our evaluation function `eval` is total: it returns `Error` when it gets stuck. We then define two functions that we want to implement. They both rewrite an expression tree into an different shape, that should be equivalent with respect to our evaluation function.


```

sealed abstract class List
case class Cons(head: BigInt, tail: List) extends List
case object Nil extends List

def size(lst: List): BigInt = {
  lst match {
    case Nil => BigInt(0)
    case Cons(_, rest) => BigInt(1) + size(rest)
  }
} ensuring(_ ≥ 0)

def content(lst : List) : Set[BIGInt] = {
  lst match {
    case Nil => Set()
    case Cons(e, rest) => Set(e) ++ content(rest)
  }
}

```

Figure 2.2 – User-defined list structure with the usual size and content abstraction functions. Here and throughout the chapter, the content abstraction computes a *set* of elements, but it can easily be extended to handle multisets (bags) by using the same techniques [dMB09] if stronger contracts are desired

We can refine the specification by stating that the *sum* of the sizes of the two lists should match the size of the input one:

```

def split(lst: List): (List, List) = {
  choose { (r: (List, List)) =>
    content(lst) == content(r._1) ++ content(r._2)
    && abs(size(r._1) - size(r._2)) ≤ 1
    && (size(r._1) + size(r._2)) == size(lst)
  }
}

```

We then finally obtain a useful split function:

```

def split(lst: List): (List, List) = lst match {
  case Nil => (Nil, Nil)
  case Cons(h, Nil) => (Nil, Cons(h, Nil))
  case Cons(h1, Cons(h2, t2)) =>
    val r = split(t2)
    (Cons(h1, r._1), Cons(h2, r._2))
}

```

We observe that in this programming style, users can write (or generate) code by conjoining orthogonal requirements, such as constraints on the sizes and contents, which are only indirectly related. The rapid feedback makes it possible to go through multiple candidates rapidly, strengthening the specification as required.

We believe this rapid feedback is mandatory when developing from specifications. One reason is that, as contracts are typically partial, results obtained from under-specifications can be remote from the desired output. Thus, a desirable strategy is to rapidly iterate and refine specifications until the output matches the expectations.

Insertion sort

Sorting is an example often used for illustrating declarative descriptions of problems. We therefore continue this overview of Leon's synthesis capabilities by showing how it synthesizes an implementation of several sorting algorithms, starting from insertion sort. Figure 2.3 shows the specification of the problem. From this, Leon generates the solution in Figure 2.4 within seconds and without further hints.

Advantages of specifications. Comparing Figure 2.3 and Figure 2.4, which are of similar size, we might wonder what we have gained by using specifications instead of implementations. Whereas only a widespread use of synthesis systems will give the true answer, we anticipate at least three reasons (with 3. partly following from 2.):

1. **Flexibility:** By supporting synthesis from specifications, we do not eliminate the ability to directly write implementations when this is more desirable, rather we add the freedom and the expressive power to describe problems in additional ways that could be appropriate; the new mechanism does not harm performance or readability when not used;
2. **Narrower gap between requirements and software:** Natural language and mathematical descriptions of structures often have the form of *conjunctions* that more directly map to **choose** constructs than to recursive functions that compute the precise objects. We view the sorting process as one of the many possible ways of obtaining a collection that has (1) the same elements and (2) is sorted, as opposed to thinking of a particular sorting algorithm.
3. **Reusability when introducing new operations:** Once we specify key invariants and abstraction functions, we can reuse them to define new versions of these operations; a related concept is the ability to express orthogonal requirements independently [Jac95].

We next illustrate the last point using examples of reusability as we add new operations: synthesizing removal from a sorted list given the specification for insertion, and synthesizing merge sort given a specification for sort.

Removal and merge for sorted lists. Suppose that, after synthesizing insertion into a sorted list, the developer now wishes to specify the removal and merge of two sorted lists. Figure 2.5

```

def isSorted(lst: List): Boolean = lst match {
  case Nil  $\Rightarrow$  true
  case Cons(_, Nil)  $\Rightarrow$  true
  case Cons(x1, xs @ Cons(x2, _))  $\Rightarrow$   $x1 \leq x2$  && isSorted(xs)
}

def insert(lst: List, v: Int): List = {
  require(isSorted(lst))
  choose { (r: List)  $\Rightarrow$ 
    isSorted(r) && content(r) == content(lst) ++ Set(v)
  }
}

def sort(lst : List): List = {
  choose { (r: List)  $\Rightarrow$ 
    isSorted(r) && content(r) == content(lst)
  }
}

```

Figure 2.3 – Specification of sorting suitable for insertion sort

```

def insert(lst: List, v: Int): List = {
  require(isSorted(lst))
  lst match {
    case Cons(h, t)  $\Rightarrow$ 
      if (v > h) {
        Cons(h, insert(t, v))
      } else if (h == v) {
        insert(t, v)
      } else {
        Cons(v, insert(t, h))
      }
    case Nil  $\Rightarrow$  Cons(v, Nil)
  }
}

def sort(lst: List): List = {
  lst match {
    case Cons(h, t)  $\Rightarrow$  insert(sort(t), h)
    case Nil  $\Rightarrow$  Nil
  }
}

```

Figure 2.4 – Synthesized insertion sort for Figure 2.3

```
def delete(in1: List, v: Int) = {  
  require(isSorted(in1))  
  choose { (out: List) =>  
    isSorted(out) && (content(out) == content(in1) -- Set(v))  
  }  
}  
  
def merge(in1: List, in2: List) = {  
  require(isSorted(in1) && isSorted(in2))  
  choose { (out: List) =>  
    isSorted(out) && (content(out) == content(in1) ++ content(in2))  
  }  
}
```

Figure 2.5 – Specification of removal from a sorted list.

```
def delete(in1: List, v: Int): List = {  
  require(isSorted(in1))  
  in1 match {  
    case Cons(h, t) =>  
      if (v == h) {  
        delete(t, v)  
      } else {  
        Cons(h, delete(t, v))  
      }  
    case Nil => Nil  
  }  
} ensuring {(out : List) =>  
  isSorted(out) && (content(out) == content(in1) -- Set(v))  
}  
  
def merge(in1: List, in2: List): List = {  
  require(isSorted(in1) && isSorted(in2))  
  in1 match {  
    case Cons(h, t) => merge(t, insert(in2, h))  
    case Nil => Nil  
  }  
} ensuring {(out : List) =>  
  isSorted(out) && (content(out) == content(in1) ++ Set(v))  
}
```

Figure 2.6 – Implementation synthesized for Figure 2.5

shows the specification of these operations. Note that, once we have gone through the process of defining the invariant for what a sorted list means using function `isSorted` in Figure 2.3, to specify these two new operations we only need to write the concise specification in Figure 2.5. The system then automatically synthesizes the full implementations in Figure 2.6. We expect that the pay-off from such re-use grows as the complexity of structures increases.

Merge sort

Now suppose that the developer wants to ensure that the system, given a sorting specification, synthesizes merge sort instead of insertion sort. To do this, the developer can introduce the function `merge` into the scope instead of `insert`. In our current version of the system, Leon then synthesizes the following code:

```
def sort(lst : List): List = lst match {
  case Cons(h, tail) => merge(sort(tail), Cons(h, Nil))
  case Nil => Nil
}
```

Although the result is a valid synthesis output according to the given contract, it remains an implementation of the insertion sort algorithm, because `merge` is called on a list that is split in a systematically unbalanced way. Even if the split function we synthesized or implemented before is in the scope, the system may decide not to use it in the generated code.

Interactive synthesis and verified refactoring. In a situation such as above, where more control is needed, we enable the developer to refine the code, either with manual edits or by applying synthesis rules in the form of *verified refactoring* steps (such as those around which entire systems were built [BGL⁺97]).

Because synthesis can be invoked only for a fraction of a function, it enables two new usage scenarios: (1) the expert developer knows how to write the inductive part of the function, but uses synthesis as a fast auto-completion tool for simple base-cases, (2) the developer knows what the overall structure of the program will be, but uses synthesis to discover the rest of the implementation.

We illustrate the first scenario with the following partial implementation of merge-sort:

```
def sort(lst: List): List = {
  lst match {
    case Cons(_, Cons(_, _)) =>
      val (s1, s2) = split(list)
      merge(sort(s1), sort(s2))

    case _ => ???
  }
} ensuring { (out: List) =>
```

```
    isSorted(out) && content(out) == content(lst)
}
```

Leon completes this implementation in 0.6 seconds, by filling the synthesis hole with `lst`.

2.2 Formalism

In this section, we introduce the necessary formalism in order to precisely define the problem of synthesizing relational specifications into executable implementations. We describe the notation we use for concisely describing synthesis problems and their associated solutions. This enables us to describe deductive steps in a compact way.

The input to our synthesizer, referred to as a *synthesis problem*, is given by a predicate that describes a desired relation between a set of input and a set of output variables, as well as the context (program point) at which the synthesis problem appears. We represent such a problem as a quadruple

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket$$

where:

- \bar{a} denotes the set of *input variables*,
- Π is the *path condition* of the synthesis problem,
- ϕ is the *synthesis predicate*, and
- \bar{x} denotes the set of *output variables*,

The free variables of ϕ must be a subset of $\bar{a} \cup \bar{x}$. The path condition refers to a formula that holds for inputs variables at the point of the synthesis hole, and the free variables of Π are therefore a subset of \bar{a} .

To illustrate this notation, we consider the following partial function:

```
def f(a: BigInt): BigInt = {
  if(a ≥ 0) {
    ???
  } else {
    0
  }
} ensuring {
  res ⇒ res ≥ 0 && a + res ≤ 5
}
```

The representation of the corresponding synthesis problem is

$$\llbracket a \langle a \geq 0 \triangleright res \geq 0 \wedge a + res \leq 5 \rangle x \rrbracket \quad (2.1)$$

When successful, our procedure will produce a *synthesis solution* represented by the pair

$$\langle P \mid \bar{T} \rangle$$

where P is the *precondition*, and \bar{T} is the *program term*. The free variables of both P and \bar{T} must range over \bar{a} . The intuition is that, whenever the path condition and the precondition are satisfied, evaluating $\phi[\bar{x} \mapsto \bar{T}]$ should evaluate to true, i.e. \bar{T} are realizers for a solution to \bar{x} in ϕ given the inputs \bar{a} . Furthermore, for a solution to be as general as possible, the precondition must be as weak as possible. We denote that a solution pair *solves* a given synthesis problem with the following notation:

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle$$

For our solution pair to be valid, the following two properties must hold:

– *Relation refinement:*

$$\Pi \wedge P \models \phi[\bar{x} \mapsto \bar{T}]$$

This property states that whenever the path- and precondition hold, the program \bar{T} can be used to generate values for the output variables \bar{x} such that the predicate ϕ is satisfied.

– *Domain preservation:*

$$\Pi \wedge (\exists \bar{x} : \phi) \models P$$

This property states that the precondition P cannot exclude inputs for which an output would exist such that ϕ is satisfied.

A valid solution to the synthesis problem (2.1) is given by $\langle a \leq 5 \mid 0 \rangle$. We thus have that

$$\llbracket a \langle a \geq 0 \triangleright x \geq 0 \wedge a + x \leq 5 \rangle x \rrbracket \vdash \langle a \leq 5 \mid 0 \rangle$$

The precondition $a \leq 5$ characterizes exactly the input values for which a solution exists, and for all such values, the constant 0 is a valid solution term for x . Note that the solution is in general not unique; alternative solutions for this particular problem also include, for example, $\langle a \leq 5 \mid 5 - a \rangle$.

For certain classes of formulas, a given set of rules can ensure completeness of synthesis. This is for instance the case for integer linear-arithmetic relations [KMPS10] or algebraic data-types [JKS13] without recursive functions. We exploit these past results and implement the procedures as deductive rules, thus making Leon complete for such relations. Our approach however targets synthesis problems that go beyond these decidable fragments (e.g, by allowing arbitrary recursive functions).

In the general case, producing partial solutions, as well as their corresponding weakest pre-conditions is a difficult task, as both have to be discovered simultaneously. Several of the implemented rules will assume that the solution is complete (under a given path-condition) and only attempt to find solutions where the precondition is **true**.

2.3 Deductive Framework

Building on our correctness criteria for synthesis solutions, we now describe *inference rules* for synthesis. Such rules describe relations between synthesis problems, thus capturing how some problems can be solved by reduction to others.

We distinguish three important operations in our deductive framework, and illustrate them with an example of a simple If-Zero rule that we informally describe. The rule decomposes a problem into two subproblems, one where one integer input variable is known to be zero and one where it is explicitly non-zero.

Rule Instantiation Applying rules to a given synthesis problem yields a list of instantiations. Each instantiation corresponds to a distinct way of applying the rule to the problem. The If-Zero rule thus gets instantiated as many times as there are integer input variables to our problem. For instance, given a synthesis problem with three integer inputs:

$$\llbracket a_0, a_1, a_2 \langle \Pi \triangleright \phi \rangle x \rrbracket$$

instantiating If-Zero on this problem yields three independent rule instantiations: "If-Zero on a_0 ", "If-Zero on a_1 ", and "If-Zero on a_2 ".

Application of a Rule Instantiation A rule instantiation serves as intermediate step between a problem and a decomposition and/or solution. Applying a rule instantiation effectively decomposes the problem and generates the corresponding sub-problems. Applying the instantiation "If-Zero on a_1 " yields the following sub-problems:

$$\llbracket a_0, a_2 \langle \Pi[a_1 \mapsto 0] \triangleright \phi[a_1 \mapsto 0] \rangle x \rrbracket$$

$$\llbracket a_0, a_1, a_2 \langle a_1 \neq 0 \wedge \Pi \triangleright \phi \rangle x \rrbracket$$

A rule instantiation can return an empty list of sub-problems. We call such rules *closing rules*. Instead of decomposing problems, they attempt to find solutions immediately. Our framework defines several of such rules, that are necessary for reaching any solutions.

We expect most of the computational work to take place in the application function, because some rules depend heavily on SMT solvers to carry out additional checks.

Solutions Recomposition A rule instantiation indicates how tentative solutions to the sub-problems can be recomposed to create a solution for the overall problem. For our "If-Zero on a_1 " instantiation, the recomposition function is defined as:

$$(\langle P_0 \mid \bar{T}_0 \rangle, \langle P_{\neq 0} \mid \bar{T}_{\neq 0} \rangle) \mapsto \langle (a_1 = 0 \wedge P_0) \vee (a_1 \neq 0 \wedge P_1) \mid \text{if}(a_1 = 0) \{ \bar{T}_0 \} \text{ else } \{ \bar{T}_{\neq 0} \} \rangle$$

By using the recomposition function independently from the decomposition itself, the framework knows what to expect from solutions generated by this instantiation before fully applying them. This enables the framework to prioritize certain rule instantiations. For instance, our "If-Zero on X" instantiations might be discouraged if our cost model assigns a high cost to *if-then-else* expressions. This prioritization is done before applying the decomposition function.

The validity of each rule can be established independently from its instantiations, or from the contexts in which it is used. This in turn guarantees that the programs obtained by successive applications of validated rules are correct by construction.

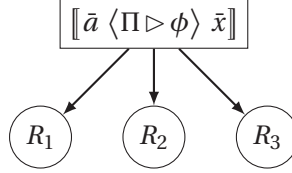
2.3.1 Exploring the Space of Applications

Our synthesis framework currently defines more than 20 generic rules. These rules can apply in multiple ways on a given problem and, as we have seen previously, a given deductive rule can generate many sub-problems (its premises or dependencies). We can thus see the space of rule applications as an *And-Or* graph. The graph is extended to store state variables for each node, distinguishing three important states for each node:

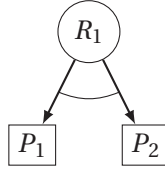
- *Open*: it is not yet known if the node can be solved.
- *Solved*: the node has at least one solution, it is possible to obtain a non-empty stream of solutions for this node.
- *Impossible*: the node definitely has no solution.

We restrict our graph in the sense that a given node does not mix *and*-descendants with *or*-descendants. We thus lift the and/or distinction to the level of nodes.

Or Node. An *or*-node stores synthesis problems. Its descendants are the multiple rule instantiations that apply to this problem. To solve the synthesis problem, it is sufficient that one of the rule-instantiations leads to a solution. The root of our graph is an *or*-node: the initial synthesis problem. We display *or*-nodes as rectangular nodes in our graphs.



And Node. We represent rule applications that decompose a problem into a list of sub-problems as *and*-nodes. The and-descendants represent the all the necessary premises expressible as sub-problems for a given rule to apply. We display *and*-nodes as round nodes in our graphs, the edges leading to their descendants are also connected to indicate that they are and-edges.



The state of each node can be mostly determined recursively by the states of its descendants, according to the following intuitive principles:

1. An *or*-node is *impossible* if all of its descendants are impossible (this also includes the case where no descendants exist).
2. An *and*-node is *impossible* if one of its descendants is *impossible*.
3. An *and*-node without descendants might either have found solutions for the problem, or failed to apply. It is thus either *solved* or *impossible* (see for instance the rule Ground-1 in 2.7).
4. An *or*-node is *solved* if at least one of its descendants is solved. The stream of solutions of the *or*-node is a combination of the streams of solutions of the *solved* descendants.
5. If all descendants of an *and*-node are solved, we take the cross-product of its sub-solutions, and build corresponding solutions for the overall problem by following the rule's description. If this stream of solution is non-empty, the *and*-node is *solved*, otherwise it is *impossible*.

Note that an *impossible* node refers to the ability of our tool to find any solution. It does not indicate that the problem is knowingly infeasible. On the contrary, a problem that is detected to be infeasible will typically be "solved" with an empty solution (a solution with false as precondition), as illustrated in the Optimistic-Ground rule in Figure 2.11. This distinction is crucial for correctly understanding how rules such as Case-Split will behave when the specification renders one case infeasible.

The search graph can be expanded on demand. The descendants of an *or*-node are obtained by instantiating all defined rules on the corresponding problem. As for *and*-nodes, the application function can be invoked to possibly generate sub-problems or immediate solutions.

2.3.2 Synthesis Rules

We describe our deductive rules in the form of inference rules. We provide one example of this rule notation, with added emphasis:

$$\frac{\boxed{\llbracket \bar{a}_1 \langle \Pi_1 \triangleright \phi_1 \rangle \bar{x}_1 \rrbracket}_{(b)} \vdash \boxed{\langle P_1 \mid \bar{T}_1 \rangle}_{(c)} \quad \boxed{\llbracket \bar{a}_2 \langle \Pi_2 \triangleright \phi_2 \rangle \bar{x}_2 \rrbracket}_{(b)} \vdash \boxed{\langle P_2 \mid \bar{T}_2 \rangle}_{(c)}}{\boxed{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket}_{(a)} \vdash \boxed{\langle P_r \mid \bar{T}_r \rangle}_{(d)}}$$

This rule is to be understood as follows: From an initial problem matching (a), we can decompose it in one or more sub-problems (b). Assuming we find solutions for all sub-problems (c), we construct a solution (d) for the overall problem. The sub-problems (b) are typically variations of (a), and the overall solution (d) is a function of the sub-solutions (c).

To illustrate the various rules, we assume the presence of a List data-type in the program, defined as follows:

```
abstract class List
case class Cons(h: Int, t: List) extends List
case object Nil extends List
```

Normalizing Rules

The set of normalizing rules can be seen as simplifiers. These rules normalize the specification syntactically and are expected to gradually turn a synthesis problem into a normal form.

In Figure 2.7, we describe several normalizing rules that exploit particular kinds of problems or specifications. As a first example, consider the rule One-point that reads as follows: "If the predicate of a synthesis problem contains a top-level atom of the form $x_0 = t$, where x_0 is an output variable not appearing in the term t , then we can solve a simpler problem where x_0 is substituted by t , obtain a solution $\langle P \mid \bar{T} \rangle$ and reconstruct a solution for the original one by

first computing the value for t and then assigning as the result for x_0 ". Note that the way we compose the solution enables t to refer to variables in \bar{x} .

The Assert rule transfers a specification fragment that only refers to input variables to the path condition. The two Ground rules are applicable when no input variables are available. In this scenario, the problem becomes purely existential. We can thus ask an SMT solver for a solution. If the solver returns a model, we evaluate it for each output variable to yield a solution. If the solver returns UNSAT (guaranteeing the absence of valid output values), we know the problem is impossible to solve. The two rules described here represent two outcomes of the same call to the SMT solver. If the solver times out, the rule does not apply. Unconstrained-Output detects problems where one output value is left unconstrained; we can give it any value. We therefore instantiate the corresponding type to obtain a solution for this output variable. Similarly, the rule Unused-Input removes input variables that are unconstrained. Strictly speaking, unconstrained input variables can be used as value for unconstrained output variables, but we exclude this scenario. Note that unlike the output variables, input variables could also be constrained by the path-condition. The Independent-Split rule exploits specifications that are conjunctions of independent formulas. This typically occurs during the synthesis of a tuple of unrelated values. We implement a family of rules for detecting equivalences between inputs. The rule Equivalent-Inputs-1 detects obvious equalities between two inputs and ensures that only one variable is used by the specification. Equivalent-Inputs-2 identifies inputs that are fully characterized by other inputs. In our case, the rule detects the implicit equivalence $l = \text{Cons}(h, t)$ implied by the constraints on the type of l , as well as constraints on the fields of the ADT. As we can see, these rules do not directly eliminate the inputs that are replaced with their equivalence from the input set, we leave this to the dedicated rule Unused-Input.

In Figure 2.8, we define several unification-based rules that either detect impossible problems, or rewrite them to make their specifications more explicit. Although the rules are defined in order to apply to arbitrary ADTs, we describe their instantiation for the List data-type type defined earlier. The first unification rule detects unsatisfiable predicate $lst = \text{Cons}(\dots, lst)$ (which can only describe infinite structures). Similarly, the rule Unification-2 enforces the fact that two distinct ADTs cannot be equal. Unification-3 explicitly exposes equalities between arguments. As described here, these rules match formulas at the top-level of the specification but could also apply to inner formulas. The ADT-Unwrap rule converts occurrences of ADT literals to equivalent constraints over selectors. The equality constraints exposed can then be exploited by further rules.

The two rules described in Figure 2.9 transforms specifications to equivalent formulas, but with a shape that enables other rules. These rules apply when a variable is known to be one specific case of an algebraic data-type. They expose the fields of the ADT as separate variables, which enables rules such as ADT-Unwrap to apply.

$$\begin{array}{c}
 \text{ONE-POINT} \\
 \frac{\llbracket \bar{a} \langle \Pi \triangleright \phi[x_0 \mapsto t] \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle \quad x_0 \notin \text{vars}(t)}{\llbracket \bar{a} \langle \Pi \triangleright x_0 = t \wedge \phi \rangle x_0, \bar{x} \rrbracket \vdash \langle P \mid (t[\bar{x} \mapsto \bar{T}], \bar{T}) \rangle} \\
 \\
 \text{ASSERT} \quad \frac{\text{vars}(\phi_1) \cap \bar{x} = \emptyset \quad \llbracket \bar{a} \langle \Pi \wedge \phi_1 \triangleright \phi_2 \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi_1 \wedge \phi_2 \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle} \quad \text{GROUND-1} \quad \frac{M \models \phi}{\llbracket \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \top \mid M(\bar{x}) \rangle} \\
 \\
 \text{GROUND-2} \quad \frac{\neg \exists M. M \models \phi}{\llbracket \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \perp \mid \text{error} \rangle} \quad \text{UNCONSTRAINED-OUTPUT} \quad \frac{x_0 \notin \text{vars}(\phi) \quad v_0 \in \text{typeOf}(x_0) \quad \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x_0, \bar{x} \rrbracket \vdash \langle P \mid (v_0, \bar{T}) \rangle} \\
 \\
 \text{UNUSED-INPUT} \quad \frac{a_0 \notin \text{vars}(\Pi) \cup \text{vars}(\phi) \quad \llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket a_0, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle} \\
 \\
 \text{INDEPENDENT-SPLIT} \quad \frac{\text{vars}(\phi_1) \cap \bar{x}_2 = \emptyset \quad \text{vars}(\phi_2) \cap \bar{x}_1 = \emptyset \quad \llbracket \bar{a} \langle \Pi \triangleright \phi_1 \rangle \bar{x}_1 \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket \bar{a} \langle \Pi \triangleright \phi_2 \rangle \bar{x}_2 \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi_1 \wedge \phi_2 \rangle \bar{x}_1, \bar{x}_2 \rrbracket \vdash \langle P_1 \wedge P_2 \mid (\bar{T}_1, \bar{T}_2) \rangle} \\
 \\
 \text{EQUIVALENT-INPUTS-1} \quad \frac{\llbracket a_1, a_2, \bar{a} \langle \Pi[a_2 \mapsto a_1] \triangleright \phi[a_2 \mapsto a_1] \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket a_1, a_2, \bar{a} \langle \Pi \triangleright a_1 = a_2 \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle} \\
 \\
 \text{EQUIVALENT-INPUTS-2} \quad \frac{\llbracket h, t, l, \bar{a} \langle \Pi[l \mapsto \text{Cons}(h, t)] \triangleright \phi[l \mapsto \text{Cons}(h, t)] \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket h, t, l, \bar{a} \langle \Pi \triangleright l.\text{isInstanceOf}[\text{Cons}] \wedge l.\text{head} = h \wedge l.\text{tail} = t \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}
 \end{array}$$

Figure 2.7 – Normalizing rules for particular problems and specifications.

$$\begin{array}{c}
 \text{UNIFICATION-1} \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \text{lst} = \text{Cons}(\dots, \text{lst}) \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle \perp \mid \text{error} \rangle \\
 \\
 \text{UNIFICATION-2} \\
 \hline
 \llbracket \bar{a} \langle \Pi \triangleright \text{Cons}(h, t) = \text{Nil}() \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle \perp \mid \text{error} \rangle \\
 \\
 \text{UNIFICATION-3} \\
 \frac{\llbracket \bar{a} \langle \Pi \triangleright h_1 = h_2 \wedge t_1 = t_2 \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \text{Cons}(h_1, t_1) = \text{Cons}(h_2, t_2) \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle} \\
 \\
 \text{DISUNIFICATION} \\
 \frac{\llbracket \bar{a} \langle \Pi \triangleright (h_1 \neq h_2 \vee t_1 \neq t_2) \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \text{Cons}(h_1, t_1) \neq \text{Cons}(h_2, t_2) \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle} \\
 \\
 \text{ADT-UNWRAP} \\
 \frac{\llbracket \bar{a} \langle \Pi \triangleright ah = e.h \wedge at = e.t \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \text{Cons}(ah, at) = e \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}
 \end{array}$$

Figure 2.8 – Normalizing rules for algebraic data-types.

$$\begin{array}{c}
 \text{DETUPLE-INPUT} \\
 \frac{\text{typeOf}(a_1) = \text{Cons} \quad \llbracket h, t, \bar{a} \langle \Pi[a_1 \mapsto \text{Cons}(h, t)] \triangleright \phi[a_1 \mapsto \text{Cons}(h, t)] \rangle \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket a_1, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P[h \mapsto a_1.h, t \mapsto a_1.t] \mid \bar{T}[h \mapsto a_1.h, t \mapsto a_1.t] \rangle} \\
 \\
 \text{DETUPLE-OUTPUT} \\
 \frac{\text{typeOf}(x_1) = \text{Cons} \quad \llbracket \bar{a} \langle \Pi \triangleright \phi[x_1 \mapsto \text{Cons}(h, t)] \rangle h, t, \bar{x} \rrbracket \vdash \langle P \mid \bar{T} \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle x_1, \bar{x} \rrbracket \vdash \langle P \mid (\text{Cons}(h, t), \bar{T}(\bar{x})) \rangle}
 \end{array}$$

Figure 2.9 – Detupling for input and output variables.

Conditionals

In order to synthesize programs that include conditional expressions, we need rules such as Case-split in Figure 2.10. The intuition behind Case-split is that a disjunction in the synthesis predicate can be handled by an *if-then-else* expression in the synthesized code, and each subproblem (corresponding to predicates ϕ_1 and ϕ_2 in the rule) can be treated separately. As we would expect, the precondition for the final program is obtained by taking the disjunction of the preconditions for the subproblems. This corresponds to the intuition that the disjunctive predicate is realizable if and only if one of its disjuncts is. Note as well that even though the disjunction is symmetrical, in the final program we necessarily privilege one branch over the other one. An extreme case is when the first precondition is true and the “else” branch becomes unreachable.

The rules Equality-Split and Inequality-Split will introduce conditionals by splitting the input space. Equality-Split will divide the search space by considering the cases where two inputs of the same type are equal or not. Note that for the case where the two variables are equal, we immediately substitute the variables. The rule Inequality-Split does the same for comparable inputs; it only includes machine integers (bit-vectors) and mathematical integers. For any pair of integer inputs, we consider the three possible cases: $a_1 > a_2$, $a_1 < a_2$ and $a_1 = a_2$. This rule is, for instance, useful to synthesize a function that computes the maximum of two integers.

The Input-Split rule relies on a Boolean input variable to split the search space. In practice, Boolean input variables are likely to be flags or options that dictate how the function should behave. The rule hence specializes the problem for each input value.

The rule If-Split exploits a specification that is divided in multiple cases, where the condition of the division depends only on input-variables. This case-analysis is imitated by the rule to produce a solution that does a similar case-analysis. If-Split can also be seen as normalizing the if-then-else expression “if(c) { t } else { e }” into $(c \wedge t) \vee (\neg c \wedge e)$. This then enables us to instead apply Case-Split followed by Assert on its sub-problems; thus yielding equivalent sub-problems and solutions.

The ADT-Split rule introduces a pattern-matching expression that decomposes a given ADT argument into all its defined cases. The input variable is decomposed into the fields of the given ADT case. We provide in 2.10 an example of ADT-Split for the List ADT.

Solver-Based Heuristics

In Figure 2.11, we continue our overview of rules with two rules in that rely on an SMT solver to find potential immediate solutions. Unlike for Ground rules, the problems here does contain input variables and thus remains of $\exists\forall$ nature. However, a literal solution (a solution term that does not reference input variables) might exist.

The rules Optimistic-Ground-1 and Optimistic-Ground-2 implement this approach by opti-

$$\begin{array}{c}
 \text{CASE-SPLIT} \\
 \frac{\llbracket \bar{a} \langle \Pi \triangleright \phi_1 \wedge \phi_3 \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket \bar{a} \langle \Pi \triangleright \phi_2 \wedge \phi_3 \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket \bar{a} \langle \Pi \triangleright (\phi_1 \vee \phi_2) \wedge \phi_3 \rangle \bar{x} \rrbracket \vdash \langle P_1 \vee P_2 \mid \text{if}(P_1) \{ \bar{T}_1 \} \text{ else } \{ \bar{T}_2 \} \rangle} \\
 \\
 \text{EQUALITY-SPLIT} \\
 \frac{\text{typeOf}(a_1) = \text{typeOf}(a_2) \quad \llbracket a_1, \bar{a} \langle \Pi[a_2 \mapsto a_1] \triangleright \phi[a_2 \mapsto a_1] \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket a_1, a_2, \bar{a} \langle \Pi \wedge a_1 \neq a_2 \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket a_1, a_2, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle (a_1 = a_2 \wedge P_1) \vee (a_1 \neq a_2 \wedge P_2) \mid \text{if}(a_1 = a_2) \{ \bar{T}_1 \} \text{ else } \{ \bar{T}_2 \} \rangle} \\
 \\
 \text{INEQUALITY-SPLIT} \\
 \frac{\text{typeOf}(a_1) = \text{typeOf}(a_2) \quad \text{typeOf}(a_1) \in \{Int, BigInt\} \quad \llbracket a_1, a_2, \bar{a} \langle \Pi \wedge a_1 = a_2 \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket a_1, a_2, \bar{a} \langle \Pi \wedge a_1 < a_2 \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle \quad \llbracket a_1, a_2, \bar{a} \langle \Pi \wedge a_1 > a_2 \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_3 \mid \bar{T}_3 \rangle}{\llbracket a_1, a_2, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \dots \mid \text{if}(a_1 = a_2) \{ \bar{T}_1 \} \text{ else } \{ \text{if}(a_1 < a_2) \{ \bar{T}_2 \} \text{ else } \{ \bar{T}_3 \} \} \rangle} \\
 \\
 \text{INPUT-SPLIT} \\
 \frac{\text{typeOf}(a_1) = Boolean \quad \llbracket \bar{a} \langle \Pi[a_1 \mapsto \top] \triangleright \phi[a_1 \mapsto \top] \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket \bar{a} \langle \Pi[a_1 \mapsto \perp] \triangleright \phi[a_1 \mapsto \perp] \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket a_1, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle (a_1 \wedge P_1) \vee (\neg a_1 \wedge P_2) \mid \text{if}(a_1) \{ \bar{T}_1 \} \text{ else } \{ \bar{T}_2 \} \rangle} \\
 \\
 \text{IF-SPLIT} \\
 \frac{\text{vars}(c) \cap \bar{x} = \emptyset \quad \llbracket \bar{a} \langle \Pi \wedge c \triangleright e_1 \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket \bar{a} \langle \Pi \wedge \neg c \triangleright e_2 \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket \bar{a} \langle \Pi \triangleright \text{if}(c) e_1 \text{ else } e_2 \wedge \phi \rangle \bar{x} \rrbracket \vdash \langle (c \wedge P_1) \vee (\neg c \wedge P_2) \mid \text{if}(c) \{ \bar{T}_1 \} \text{ else } \{ \bar{T}_2 \} \rangle} \\
 \\
 \text{ADT-SPLIT (LIST)} \\
 \frac{\text{typeOf}(a_1) = List \quad \llbracket \bar{a} \langle \Pi[a_1 \mapsto Nil] \triangleright \phi[a_1 \mapsto Nil] \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid \bar{T}_1 \rangle \quad \llbracket h, t, \bar{a} \langle \Pi[a_1 \mapsto Cons(h, t)] \triangleright \phi[a_1 \mapsto Cons(h, t)] \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid \bar{T}_2 \rangle}{\llbracket a_1, \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid a_1 \text{ match } \{ \text{case } Nil \Rightarrow \bar{T}_1 \text{ case } Cons(h, t) \Rightarrow \bar{T}_2 \} \rangle} \\
 \text{with } P = a_1 \text{ match } \{ \text{case } Nil \Rightarrow P_1 \text{ case } Cons(h, t) \Rightarrow P_2 \}
 \end{array}$$

Figure 2.10 – Rules introducing conditional expressions.

mistically searching for literal solutions. One important difference with Ground is that ϕ can reference input variables. The solver initially searches for a model for $\exists \bar{a} \bar{x}. \Pi \wedge \phi$. If no such model exists, the problem is known to be impossible. If a model M is found, we still have to validate it by ensuring the literal solution is valid for arbitrary inputs. We thus perform a secondary verification query to the SMT solver looking for models of $\exists \bar{a}. \Pi \wedge \neg \phi[\bar{x} \mapsto M(\bar{x})]$ (i.e, counter-examples). If no counter-examples exist, the solution found in the first step is returned as a solution. Otherwise, we can refine the initial formula and try again. This procedure is known as a counter-example guided inductive synthesis (CEGIS) loop. Because there exists an infinite number of literals, this search is not guaranteed to terminate. In practice, we found that this rule is unlikely to find solutions for the kind of problems we target, we thus abort after three incorrect attempts. It is however efficient at detecting impossible problems: If the specification ϕ is unsatisfiable due to contradictions in the formula, Optimistic-Ground will close this branch of the search tree with a solution that has \perp as a pre-condition.

$$\begin{array}{c}
 \text{OPTIMISTIC-GROUND-1} \\
 \frac{M \models \phi \quad \forall \bar{a}. \phi[\bar{x} \mapsto M(\bar{x})]}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \top \mid M(\bar{x}) \rangle}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{OPTIMISTIC-GROUND-2} \\
 \frac{\neg \exists M. M \models \phi}{\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \perp \mid \text{error} \rangle}
 \end{array}$$

Figure 2.11 – Solver-based heuristics

2.3.3 Cost Models

Our framework defines several rules that can apply in different ways to the same problem. This results in a search graph with a high branching-factor. We also note that as the rules do not generally guarantee progress, our search graph is typically of infinite depth. The shape of our search graph hence prevents both depth- and breadth-first search strategies: A depth-first search could get trapped in an infinite branch without making any progress and breadth-first search is overly inefficient.

We observe that for the benchmarks we consider, a deduction tree of depth 4 or 5 is generally sufficient. However the smallest necessary sub-tree for deriving a solution makes only a small fraction of the full search graph of depth 5. This calls for a more advanced search strategy.

Our search algorithm uses a variant of best-first search strategy similar to A* ; it underestimates the cost of the complete solution according to cost-models. The cost function needs to underestimate the cost of a potential final solution. The framework takes a cost function as parameter

$$\mathcal{C} : \text{Solution} \rightarrow \mathbb{N}$$

The search is also implicitly bounded by a maximal cost \mathcal{C}_{max} , used to assign costs to impossible nodes. We determine the cost of any given node as follows: the cost of a closed node is the

cost of its solution and the cost of an impossible node is \mathcal{C}_{max} . For open *or*-nodes, the cost is determined by the minimal cost of its descendants. Finally, the cost of open *and*-nodes uses the recomposition function to obtain a partial solution, on which we apply our cost function.

Our cost-models supports prioritizing certain shapes of solutions. For instance, we could assign a high-cost to nested *if-then-else* expressions. This would force the framework to first try alternative derivations before exploring nested applications to rules that introduce *id-then-else*. The cost-model we use in practice makes use of this prioritization to discourage the introduction of branching constructs deep in the solution expression.

2.4 Symbolic Term Exploration

In this section, we describe a closing rule that is responsible for closing many of the branches in derivation trees. We call it symbolic term exploration (STE).

The core idea behind STE is to symbolically represent many possible terms (programs) and to iteratively prune them out by using counter-examples and test case generation until either (1) a valid term is proved to solve the synthesis problem or (2) all programs in the search space have been shown to be inadequate. Because we have rules that take care of introducing branching constructs or recursive functions, we focus STE on the search for terms consisting only of constructors and calls to existing functions.

2.4.1 Grammars for Programs

The space of programs covered by our symbolic exploration is determined by a context-free grammar of well-typed expressions E , described by the triple

$$G = (N, P, S)$$

where

- N is a finite set (the typed non-terminals),
- P is a finite subset of $N \times (\bar{N}, \bar{E} \mapsto E)$ (the productions), and
- $S \in N$ (the start symbol).

We diverge from the standard description of grammars and ignore the set of terminals. Instead, we view the resulting expressions directly as trees, and we provide productions as a relation between non-terminals and *node constructors*. These node constructors provide a list of non-terminal dependencies as well as a function for building the resulting expression from sub-expressions.

The non-terminals are typed, and the constructor functions should produce expressions of the corresponding type. This grammar is implicitly restricted to produce only program terms that typecheck. The arity of a production is defined by the arity of the constructor function.

The purpose of our procedure is to find out, given a synthesis problem $\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket$ and a grammar G with a starting symbol such as $\text{typeOf}(S) = \text{typeOf}(\bar{x})$, whether

$$\exists e \in \mathcal{L}(G). \forall \bar{a}. \Pi \Rightarrow \phi[\bar{x} \mapsto e]$$

When we discover such an expression e , our procedure solves the synthesis problem with the following solution:

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle \top \mid e \rangle$$

It is not required that a grammar produces *every* value for a given type; we could hypothesize, for instance, that our synthesis solutions will only need some very specific constants, such as 0, 1 or -1 . But it is more likely that our synthesis solutions requires the use of input variables and existing functions. As illustration, we consider the following synthesis problem:

$$\llbracket i, j, l \langle \Pi \triangleright \text{sum}(x) = \text{sum}(l) + i + j \rangle x \rrbracket$$

The variables x and l are of type `IntList` and both i and j are integers. The function `sum` is a user-defined function that computes the sum of the elements of a list. Here, we describe an example of a grammar of `IntList` expressions $G = (N, P, \text{IntList})$ with the following productions:

$$\begin{aligned} \text{IntList} &::= \text{Nil} \mid l \\ &::= \text{Cons}(\text{BigInt}, \text{IntList}) \\ \text{BigInt} &::= 0 \mid 1 \mid -1 \mid 42 \mid i \mid j \\ &::= \text{BigInt} + \text{BigInt} \mid \text{BigInt} - \text{BigInt} \mid \text{BigInt} * \text{BigInt} \end{aligned}$$

We observe that this grammar contains the three input variables i , j , and l . Of course, it cannot refer to x . This grammar produces an infinite number of terms, among which we find several that are solutions of our synthesis problem:

- $\text{Cons}(i, \text{Cons}(j, l))$,
- $\text{Cons}(i+j, l)$,
- $\text{Cons}(i+1, j-1, l)$,
- $\text{Cons}(i+j, \text{Cons}(0, \text{Cons}(0, l)))$,
- and so on.

Here, we describe how we explore this grammar in order to effectively find a solution amongst the space of represented programs.

Bounding the Space of Programs

As we pointed out in Section 2.4.1, it is not necessary for our grammar to be complete. In fact, it is also desirable to make it finite: this guarantees that our search procedure terminates. Because our deductive framework sequentially orchestrates multiple rules, we would have STE fail quickly rather than prevent other rules from ever applying.

A natural way of limiting the space of programs represented by a given grammar is to bound the unfolding depth. This consequently bounds the depth of the resulting expression trees. Instead of modifying how we compute the set of programs for a given grammar, we instead bound the depth by applying a transformation on the grammar itself. We use $G_{\downarrow d}$ to denote the grammar G at depth at most d . This implies that $\mathcal{L}(G_{\downarrow i}) \subseteq \mathcal{L}(G_{\downarrow j})$ for all $i \leq j$.

Given the grammar G presented in Section 2.4.1, the depth-based grammar $G_{\downarrow 3}$ has starting symbol $IntList_{\downarrow 3}$ and the following productions:

$$\begin{aligned}
 IntList_{\downarrow 3} &::= Cons(BigInt_{\downarrow 2}, IntList_{\downarrow 2}) \mid IntList_{\downarrow 2} \\
 IntList_{\downarrow 2} &::= Cons(BigInt_{\downarrow 1}, IntList_{\downarrow 1}) \mid IntList_{\downarrow 1} \\
 IntList_{\downarrow 1} &::= Nil \mid I \\
 BigInt_{\downarrow 2} &::= BigInt_{\downarrow 1} + BigInt_{\downarrow 1} \mid BigInt_{\downarrow 1} - BigInt_{\downarrow 1} \mid BigInt_{\downarrow 1} * BigInt_{\downarrow 1} \mid BigInt_{\downarrow 1} \\
 BigInt_{\downarrow 1} &::= 0 \mid 1 \mid -1 \mid 42 \mid i \mid j
 \end{aligned}$$

This operation ensures that $\mathcal{L}(G_{\downarrow d})$ is finite for a any given d , thus enabling us to exhaustively explore the corresponding space of programs. But bounding the depth of expressions provides control that is too coarse-grained: the number of expressions represented by a grammar explodes as depth increases. We can see that with the synthesis problem seen before: Its smallest solution, $Cons(i+j, I)$, is contained in $\mathcal{L}(G_{\downarrow 3})$ (but not in $\mathcal{L}(G_{\downarrow 2})$). But $\mathcal{L}(G_{\downarrow 3})$ also contains expressions such as $Cons(i+j, Cons(0, I))$, which are also solutions, but strictly bigger and thus harder to reason about. As a result of bounding the depth, we consider not only more expressions than necessary but many of these expressions are overly complex.

A more granular way of bounding the search space is to bound the size instead of the depth of the resulting expressions. Again, we present this bounding operation as a transformation of grammars. This time, we look for expressions of size exactly s , denoted by $G_{|s|}$. We illustrate this transformation by applying it to our example grammar. $G_{|5|}$ has starting symbol $IntList_{|5|}$

and contains the following productions:

$$\begin{aligned}
IntList_{|5|} &::= Cons(BigInt_{|3|}, IntList_{|1|}) \mid Cons(BigInt_{|1|}, IntList_{|3|}) \\
IntList_{|5|} &::= Cons(BigInt_{|2|}, IntList_{|2|}) \\
IntList_{|3|} &::= Cons(BigInt_{|1|}, IntList_{|1|}) \\
IntList_{|1|} &::= Nil \mid I \\
BigInt_{|3|} &::= BigInt_{|1|} + BigInt_{|1|} \mid BigInt_{|1|} - BigInt_{|1|} \mid BigInt_{|1|} * BigInt_{|1|} \\
BigInt_{|1|} &::= 0 \mid 1 \mid -1 \mid 42 \mid i \mid j
\end{aligned}$$

This time, $\mathcal{L}(G_{|5|})$ contains our solution but does not contain $Cons(i+j, Cons(0, I))$, as this would correspond to an expression of size 7. This does not prevent multiple solutions from existing in the same expression size: here both $Cons(i+j, I)$ and $Cons(i, Cons(j, I))$ are valid solutions of the same size.

Handling Generic Functions and Data-Structures

Supporting generic language features brings interesting challenges, mainly because this makes the set of types infinite. To illustrate, we consider a generic list data-type $List[T]$ and its size function. Providing a complete grammar for generating integer expressions becomes impossible, as the sets of non-terminals and productions are infinite:

$$\begin{aligned}
BigInt &::= size(List[Int]) \\
BigInt &::= size(List[List[Int]]) \\
BigInt &::= size(List[List[List[Int]]]) \\
BigInt &::= \dots
\end{aligned}$$

Note that this problem also occurs without any user-defined generic operations. For instance, checking equality between two values ($a == b$) is a language construct that is implicitly generic. There is thus immediately an unbounded number of ways to generate Boolean values.

In practice, we work around these issues by limiting the type instantiations for generic operations, but we do so in a way that retains the most interesting operations: We consider all types directly present as types of input variables, and allow for them to be wrapped *once* in any generic types. Given a single input variable of type $List[T]$, we will include productions for T and $List[List[T]]$ but not $List[List[List[T]]]$. This guarantees a finite set of non-terminals and productions – a prerequisite for our expression grammars.

Generating Recursive Calls

When considering candidate expressions, it is often useful to include calls to user-defined functions. However, the synthesized expression has to terminate. We conservatively ensure this by excluding calls to functions that refer back to the function currently being synthesized.

Chapter 2. Deductive Synthesis

Because we assume that other functions terminate for all inputs prior to synthesis, we need to prevent introducing calls that cause the synthesized function to loop. However, our purely functional language often requires us to synthesize recursive implementations. Consequently, the synthesizer must be able to generate calls to the function currently getting synthesized.

We must therefore take special care to avoid introducing calls resulting in a non-terminating implementation. (Such an erroneous implementation would be conceived as valid if it trivially satisfies the specification due to inductive hypothesis over a non-well-founded relation.)

Our technique consists of recording the function arguments a at the entry point of the function, f , and keeping track of these arguments through the decompositions. We represent this information with a syntactic predicate $\Downarrow[f(a)]$. For example, we consider the synthesis of a function deleting elements from a list of integers.

```
def delete(in: IntList, v: BigInt): IntList = {
  ???
} ensuring {
  out  $\Rightarrow$  content(out) == content(in) -- Set(v)
}
```

To solve this problem, we need to traverse the list and remove elements whenever they match v . This traversal is done by recursively calling `delete`. We inject the entry call information within the path-condition of the root synthesis problem:

$$\llbracket in, v \langle \Downarrow[delete(in, v)] \triangleright content(out) = content(in) \setminus \{v\} \rangle out \rrbracket$$

This predicate tells our system that it is currently synthesizing the implementation of a call to `delete(in, v)` and plays no logical role in the path condition itself. This synthesis problem can then be decomposed as usual, by the various deduction rules available in the framework. An interesting case to consider is the problems stemming from the decomposition done by the ADT-Split rule on in ; it specializes the problem to known cases of `IntList`. Notably, the sub-problem corresponding to the `Cons` variant looks as follows:

$$\llbracket h, t, v \langle \Downarrow[delete(Cons(h, t), v)] \triangleright content(out) = content(Cons(h, t)) \setminus \{v\} \rangle out \rrbracket$$

We can see that in has been substituted by `Cons(h, t)`, which tells us that we are currently synthesizing the result of a call to `delete(Cons(h, t), v)`. We also have that by the theory of algebraic data-types, t is strictly smaller than `Cons(h, t)`. As a result, we assume that calls to `delete` with t as first argument are likely to terminate. Therefore, they are inserted within the grammar as ways to generate `IntList` values:

$$IntList ::= delete(t, BigInt)$$

This relatively simple technique enables the introduction of recursive calls that are not trivially non-terminating. In the case where it still introduces infinite recursion, the filtering of candidate expressions by using concrete execution will discard them at a later stage, though we found that this seldom occurs in practice.

2.4.2 From Grammars to Programs

Our STE procedure works with a program representing the current state of synthesis, as well as all the possible expressions represented by the grammar. We consider our synthesis of delete, after applying ADT-Split and then Equality-Split. This corresponds to the search graph displayed in Figure 2.12.

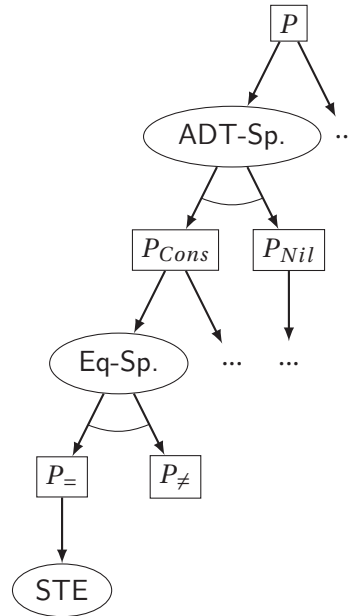


Figure 2.12 – Partial search-graph that leads to a deep invocation of STE. This graph corresponds to a top-level invocation of ADT-Split on the in variable. Then, we invoke Equality-Split between v and h on the sub-problem corresponding to the Cons case. Finally, we invoke STE in the sub-problem for when h is equal to v .

This search graph can be used to build a partial implementation of delete. This enables us to compute a good over-approximation of the solution, if a given rule was to be used in the overall solution. For the purpose of illustration, we assume that we already solved P_{Nil} with $\langle \top \mid \text{Nil} \rangle$ and that P_{\neq} remains unsolved. We display the partial solution *around* STE in Figure 2.13.

The unsolved parts are represented by synthesis holes, for which only the specification and referential transparency can be assumed. This partial implementation will be especially useful when the candidate expressions contain recursive calls: this refined version of delete is thus more precise than simply assuming its postcondition.

```

def delete(in: IntList, v: BigInt): IntList = {
  in match {
    case Cons(h, t) =>
      if (h == v) {
        STE
      } else {
        ???
      }
    case Nil => Nil
  }
} ensuring {
  res => content(res) == content(in) -- Set(v)
}

```

Figure 2.13 – Partial solution around an invocation to STE, corresponding to the search-graph displayed in Figure 2.12 (with P_{Nil} solved with $\langle T \mid Nil \rangle$ and P_{\neq} unsolved).

This, for instance, enables individual-solution candidates to be plugged into the partial implementation in place of STE. We can therefore trivially validate each candidate individually by invoking verification on the resulting program. This would, however, be rather inefficient in the presence of large grammars.

Instead, we symbolically explore the grammar by using an SMT-solver. Hence, we represent the grammar provided as parameter to STE as a series of functions. In our example, we will consider a size-bound grammar with the following productions:

$$\begin{aligned}
 IntList_{|3|} &::= Cons(BigInt_{|1|}, IntList_{|1|}) \\
 &::= delete(t, BigInt_{|1|}) \\
 IntList_{|1|} &::= Nil \mid t \\
 BigInt_{|3|} &::= BigInt_{|1|} + BigInt_{|1|} \mid BigInt_{|1|} - BigInt_{|1|} \mid BigInt_{|1|} * BigInt_{|1|} \\
 BigInt_{|1|} &::= 0 \mid 1 \mid 2 \mid v \mid h
 \end{aligned}$$

We convert each non-terminal into one or more independent functions that return candidate expressions. For instance, with the grammar described above, we could represent the non-terminals with the functions listed in Figure 2.14.

These functions represent the productions for a given non-terminal, and the choices of which production to pick is provided by global boolean variables (B-variables). We can see that by varying the B-values, we obtain results that correspond to the evaluation of all expressions permitted by the input grammar. This program thus represents the input grammar. We also need extra constraints to prevent multiple B-valuations from representing the same programs. As is, both $B1 \wedge B2 \wedge B5$ and $B1 \wedge B2 \wedge B5 \wedge B6$ represent $Cons(0, Nil)$. Hence, we enforce that


```

def genIntList3(h: BigInt,
               t: IntList,
               v: BigInt): IntList = {
  if (B1) {
    Cons(genBigInt1a(h, t, v), genIntList1(h, t, v))
  } else {
    delete(t, genBigInt1a(h, t, v))
  }
}

def genIntList1(h: BigInt,
               t: IntList,
               v: BigInt): IntList = {
  if (B2) {
    Nil
  } else {
    t
  }
}

def genBigInt3(h: BigInt,
               t: IntList,
               v: BigInt): IntList = {
  if (B3) {
    genBigInt1a(h, t, v) + genBigInt1b(h, t, v)
  } else if (B4) {
    genBigInt1a(h, t, v) - genBigInt1b(h, t, v)
  } else {
    genBigInt1a(h, t, v) * genBigInt1b(h, t, v)
  }
}

def genBigInt1a(h: BigInt,
                t: IntList,
                v: BigInt): BigInt = {
  if (B5) {
    0
  } else if (B6) {
    1
  } else if (B7) {
    2
  } else if (B8) {
    h
  } else {
    v
  }
}

def genBigInt1b(h: BigInt,
                t: IntList,
                v: BigInt): BigInt = {
  if (B9) {
    0
  } else if (B10) {
    1
  } else if (B11) {
    2
  } else if (B12) {
    h
  } else {
    v
  }
}

```

Figure 2.14 – Functions generated to represent the grammar $G_{|3|}$. The non-terminal symbol $BigInt_{|1|}$ is represented by two independent functions, because it can be used twice at the same time in $BigInt_{|3|}$.

only one production must be selected, which translates for `genBigInt1a` to the constraint

$$(\neg B5 \vee \neg B6) \wedge (\neg B6 \vee \neg B7) \wedge (\neg B7 \vee \neg B8) \wedge (\neg B5 \vee \neg B7) \wedge (\neg B5 \vee \neg B8) \wedge (\neg B6 \vee \neg B8)$$

We note that non-terminals used more than once in the same production need to be duplicated to ensure a complete coverage of the grammar. For instance, we duplicate the function `genBigInt1` because we rely on two independent values of `genBigInt1` at the same time in productions of `genBigInt3`. Without this, no valuation of B-variables would represent $h + v$.

We plug a call to the starting symbol's function in place of the `STE` invocation, in our program representing the current search tree. This gives us a program that represents candidate solutions. It remains to search for a valid B-valuation.

2.4.3 Search Algorithm

The search algorithm is composed mainly of a counter-example guided inductive synthesis (CEGIS) loop in order to symbolically explore program candidates from the input grammar. We provide a high-level algorithm in Figure 2.15. The algorithm gradually increases the target expression size, consequently increasing the coverage. To ensure termination, we bound the expression size. For a given target size s , we start by computing the set of candidate programs in $\mathcal{L}(G)_{|s|}$. This set of candidates is then filtered (`Filter`) by using concrete execution. Whenever the number of discarded candidates is deemed substantial, we skip the symbolic discovery and individually validate candidates. Otherwise, we rely on a modified CEGIS loop: we first look for one candidate programs that works on at least one input (`FindCandidate`). Given such a candidate, we validate it for all inputs. The validation procedure (`ValidateCandidate`) either confirms that the candidate is a solution, or returns a counter-example (an input for which the candidate fails to produce a valid result). When discovering a counter-example, we use it to filter the set of remaining candidates. In contrast, if no candidate can be found, we continue increasing the target expression size. We now describe in further detail the three important components of our algorithm.

Discovering Programs

Given the set of functions and constraints representing a grammar of a given size, as explained in Section 2.4.2, finding a valid program is reduced to finding a valuation for the B-variables such that the following formula is true:

$$\exists \bar{B} \forall \bar{a}. C(\bar{B}) \wedge (\Pi \Rightarrow \phi[\bar{x} \mapsto S_{|s|}(\bar{a})]) \quad (2.2)$$

where

- $C(\bar{B})$ are the extra-constraints enforcing a bijective mapping from valid B-valuations

Input: *tests*, the initial set of input tests
Input: *G*, a grammar for the search space

```

1  for s ← 1 to MaxSize do
2    Pall ←  $\mathcal{L}(G)_{|s|}$ 
3    P ← Filter(Pall, tests)
4    if  $\frac{|P|}{|P_{all}|} \leq \text{validateUpTo}$  then
5      for p ∈ P do
6        Valid | cex ← ValidateCandidate(p)
7        if Valid then
8          return Success( $\langle \top \mid p \rangle$ )
9        else
10         tests ← tests ∪ {cex}
11         P ← Filter(P, {cex})
12       end
13     end
14   else
15     continue ← true
16     while |P| > 0 ∧ continue do
17       if p ← FindCandidate(P) then
18         Valid | cex ← ValidateCandidate(p)
19         if Valid then
20           return Success( $\langle \top \mid p \rangle$ )
21         else
22           tests ← tests ∪ {cex}
23           P ← Filter(P, {cex})
24         end
25       else
26         continue ← false
27       end
28     end
29   end
30   return Failed
31 end

```

Figure 2.15 – Search algorithm.

and candidate programs.

- $S_{|s|}(\bar{a})$ is a call to the function that produces candidate values for \bar{x} .

The quantifier alternation makes this constraint difficult to handle by traditional SMT solvers. Instead, we search for a candidate program that works on *at least one input*. This translates to the following alternative formula:

$$\exists \bar{B} \bar{a}. C(\bar{B}) \wedge \Pi \wedge \phi[\bar{x} \mapsto S_{|s|}(\bar{a})] \quad (2.3)$$

If there does not exist such a B-valuation, then our grammar at the given size s is not expressive enough to encode a solution to the problem. Otherwise, we extract a B-valuation \bar{B}_0 from the model. We now proceed by validating this candidate program.

Falsifying Programs

Given a candidate program expressed as a B-valuation (\bar{B}_0), we check whether the candidate program is valid for all inputs:

$$\forall \bar{a}. \bar{B} = \bar{B}_0 \wedge (\Pi \Rightarrow \phi[\bar{x} \mapsto S_{|s|}(\bar{a})]) \quad (2.4)$$

Note that \bar{B}_0 are constants that encode a single, deterministic, program and that it is known to satisfy $C(\bar{B})$. With this in mind, it becomes clear that we are truly solving for \bar{a} . We translate it to the following formula that encodes the presence of a counter-example:

$$\exists \bar{a}. \bar{B} = \bar{B}_0 \wedge \Pi \wedge \neg \phi[\bar{x} \mapsto S_{|s|}(\bar{a})] \quad (2.5)$$

If no such \bar{a} exist, then we have found a program that realizes ϕ for all inputs and we are done. Whereas if we can find \bar{a}_0 , then this constitutes an input that confirms that our program does not meet the specification.

Eventually, because the set of possible assignments to \bar{b} is finite (for a given instantiation depth) this refinement loop terminates. If we have not found a program, we can increase the target expression size and try again. When the maximal size is reached, we abandon.

Filtering with Concrete Execution

Although the termination of our search procedure is in principle guaranteed by the finite amount of programs paired with their successive eliminations in the refinement loop, the set of programs of a given size typically grows rapidly as the target size increases. As the number

of candidates augments, the difficulty for the solver to satisfy (2.2) or (2.5) also increases. As a complement to symbolic elimination, we use concrete execution on a set of input tests to rule out many programs. For each new size, we filter the corresponding search space with all the tests available so far. In practice, this typically reduces the amount of programs to be symbolically considered by multiple orders of magnitude. A filter is also used to further reduce the search space when we discover new counter-examples, while validating candidate programs.

We prime the set of tests with inputs that satisfy the path condition. These either come directly from the synthesis framework or are generated within STE by Leon’s ground-term generators described in Section 2.5. Our input generators are lazy: if a single test excludes all programs, no other input tests are generated.

To make testing efficient in our implementation, we compile on the fly the expression $\phi[\bar{x} \mapsto S_{|s|}(\bar{a})]$ to JVM bytecode. The expression uses both the inputs \bar{a} and an assignment to \bar{B} to compute whether the program represented by the B-values succeeds in producing a valid output for \bar{a} .

This encoding of all candidate programs into executable functions enables us to rapidly test and potentially discard hundreds or even thousands of candidates within a fraction of a second. The filtering procedure periodically re-orders the tests available so that those that are most efficient at discarding candidates are tried first. The acceleration achieved through filtering with concrete executions is particularly important when STE is applied to a problem it cannot solve. In such cases, filtering often rules out all candidate programs and symbolic reasoning is never applied.

2.4.4 Execution of Partial Programs

As we have seen in Figure 2.13, the solution program that represents the current search tree is sometimes partial, because unsolved branches are “implemented” using synthesis holes. In order to test candidate solutions that are recursive, we thus need to be able to properly execute synthesis holes.

We therefore extend our evaluators with support for runtime constraint solving. Given a synthesis problem and a valuation for input values, the evaluator solves the constraint that is now purely existential to discover valid output values. The evaluators invoke the SMT solvers at runtime, once the input values are known. We cache the models of the solver invocations for two main reasons: Firstly, the solver can be slow if the necessary models is big. Caching thus improves the running time of subsequent calls with equivalent input contexts. Secondly, it is important for consistency reasons that identical invocations produce the same result, because referential transparency is generally assumed in functional programs. This however cannot be assumed of multiple identical calls to a SMT solvers, as solvers involve some pseudo non-determinism that is hard to control. Caching thus ensures that results are consistent with

a fully-implemented functional program.

This approach of solving synthesis holes at runtime has been explored in previous work [KKS12], and enables the use of constraints as advanced control structures. In their approach, the user was able to iterate over multiple solutions using the monadic “for” construct. In our setting, we are only interested in finding one solution for each pair of synthesis hole and context.

2.5 Ground-Term Generator

Given that our programs and specifications are both executable, we rely on runtime execution to filter candidate programs efficiently. This filtering executes each candidate program on a set of input examples and checks that their output satisfies the postcondition.

We therefore need to gather interesting input examples so that filtering removes most invalid candidates. The first challenge comes from the fact that the inputs to our synthesis problems are constrained by the path-condition. For instance, the problem of inserting into a sorted list would be given as follows:

$$\llbracket l, v \mid \langle isSorted(l) \triangleright isSorted(res) \wedge content(res) = content(l) \cup \{v\} \rangle res \rrbracket$$

To evaluate candidate solutions, we thus need to obtain several values for the pair (l, v) where the value for l is sorted.

2.5.1 Generate and Test

The first approach we use is to naively enumerate several values of the necessary types. We rely on a value grammar that describe a set of literal terms. The use of grammars enables us to reuse the machinery developed in Section 2.4.1. We explore a given grammar using memoization-based enumerators. This proves to be very efficient in practice: we typically generate millions of values within seconds. However, the generated tests need to satisfy the path-condition. We therefore use an evaluator to filter the candidate inputs for which the path-condition returns “true”. To perform this efficiently, we implemented an evaluator that first compiles the path-condition to JVM bytecode.

Although this is interesting when the path-condition is weak, it yields disappointing results when the path-condition is non-trivial. In our example, we can see that naively enumerating lists and selecting only those that are sorted becomes inefficient as the list size increases: All lists of size one are sorted, but only a small fraction of the lists of size 10 are sorted. This fraction in fact approaches zero as the list size increases.

2.5.2 Symbolic Input-Output Examples

As we have seen in Section 1.3, we introduce a notation for expressing symbolic tests, where inputs can have symbolic holes, and outputs are defined as expressions of these holes. The developers can thus use this notation to specify fragments of their functions. We exploit symbolic tests found in the user-provided specification by generating several concrete inputs matching the pattern provided by the user by *filling* symbolic holes with generated values.

We note that like regular pattern matches, symbolic holes can be guarded using an **if** construct in the pattern, and they are also implicitly constrained by the function's precondition. Generating values for symbolic holes is in practice as difficult as generating standard input examples. However, we believe that symbolic tests provided by the user hint at interesting features of the function. By generating tests that match the provided patterns, we are likely to test the relevant aspects of the function's intended behavior.

2.5.3 Model-Based Enumeration

We have seen that enumeration-based approaches are limited when the generated values are constrained by non-trivial formulas. Satisfying formulas by providing models for free variables is something that SMT solvers are designed to perform efficiently. It is thus natural to turn to these solvers to generate interesting examples with non-trivial constraints. However, typical SMT solvers will only produce one model per satisfiable formulas. Instead, we can look for different examples by refining the constraint, so that it excludes previously found models, and ask the solver again. We describe this generator of inputs in the form of a simple coroutine:

```

1  $C \leftarrow \Pi$ 
2 while  $\exists M. M \models C$  do
3   | yield  $M(\bar{a})$ 
4   |  $C \leftarrow C \wedge \bar{a} \neq M(\bar{a})$ 
5 end
```

However, we have seen before that there exists an infinite number of sorted lists of size one, and because finding a model of a list of size one is easier than finding bigger models, this naive refinement loop seen above does not enumerate models in a fair way. As a result, it will never reach inputs of a size bigger than one, again limiting our examples to trivial ones. It will be very efficient at finding examples which require "big" integer literals, though.

We would however like to use the solver even for our cases with sorted lists. Our enumeration procedure synthesizes a set of size functions for every types involved. These size functions enable us to also inject formulas in C , which constraints the size of the model. As a result, we

can ask the solver to find models of gradually increasing size:

```

1  $n \leftarrow 0$ 
2  $C \leftarrow \Pi$ 
3 while  $\exists M. M \models C$  do
4   yield  $M(\bar{a})$ 
5    $n \leftarrow n + 1$ 
6    $C \leftarrow C \wedge \bar{a} \neq M(\bar{a})$ 
7   if  $5 \mid n$  then
8      $C \leftarrow C \wedge (size(\bar{a}) > size(M(\bar{a})))$ 
9   end
10 end

```

Here, we force a variation of model size every five models.

2.6 Evaluation

To evaluate our system, we developed benchmarks with reusable abstraction functions. The various examples used as illustrations already point to some of the results we obtain. Here, we summarize further results and discuss some of the remaining benchmarks. The synthesis problem descriptions are available in the appendix, along with their solution as computed by Leon.

Our set of benchmarks, displayed in Figure 2.16, covers the synthesis of various operations over custom data structures with invariants, specified through the lens of abstraction functions. These benchmarks use specifications that are both easy to understand and shorter than resulting programs (except in trivial cases). Most importantly, the specification functions are easily reused across synthesis problems. We believe these are key factors in the evaluation of any synthesis procedure.

Figure 2.16 shows the list of functions we successfully synthesized. In addition to the address book and sorted list examples shown in Section 2.1, our benchmarks include operations on unary numerals, defined as is standard as “zero or successor”, and on an amortized queue implemented with two lists from a standard book on functional data-structure implementation [Oka99]. Each synthesized program has been manually validated to be a solution that a programmer might expect. The synthesis can be performed in order, meaning that an operation will be able to reuse all previously synthesized ones, thus mimicking the usual development process. For instance, multiplication on unary numerals is synthesized as repeated invocations of additions.

Typically, our system also proves automatically that the resulting program matches the specification for all inputs. In some cases, the lack of inductive invariants prevents a fully-automated proof of the synthesized code (we stop verification after a timeout of 5 seconds). In most cases

Operation	Prog. Size	Sol. Size	Calls	Proof	sec.
Compiler.rewriteMinus	218	6	0	✓	1.6
Compiler.rewriteImplies	218	6	0	✓	1.5
List.insert	59	3	0	✓	0.8
List.delete	61	18	2	✓	3.8
List.merge	75	11	1	✓	6.1
List.diff	106	11	2	✓	5.5
List.split	96	24	2	✓	3.4
SortedList.insert	91	29	2		14.9
SortedList.insertAlways	105	31	2	✓	20.3
SortedList.delete	91	18	2		7.2
SortedList.merge	138	11	2	✓	6.6
SortedList.diff	136	11	2	✓	5.5
SortedList.insertionSort	125	10	2	✓	1.9
StrictSortedList.insert	91	29	2	✓	12.3
StrictSortedList.delete	91	16	1		9.4
StrictSortedList.merge	138	11	2	✓	7.0
UnaryNumerals.add	42	9	1	✓	3.8
UnaryNumerals.distinct	66	4	1	✓	2.3
UnaryNumerals.mult	64	10	2	✓	4.4
BatchedQueue.dequeue	65	23	1		14.2
AddressBook.merge	99	11	2		6.2

Figure 2.16 – Automatically synthesized functions using our system. We consider a problem as synthesized if the solution generated is correct after manual inspection. For each generated function, the table lists the size of its syntax tree and the number of function calls it contains. ✓ indicates that the system also found a proof that the generated program matches the specification: in many cases proof and synthesis are done simultaneously, but in rare cases merely a large number of automatically generated inputs passed the specification. The final column shows the total time used for both synthesis and verification.

the synthesis succeeds sufficiently fast for a reasonable interactive experience.

2.7 Related Work

Our approach is similar in the spirit to deductive synthesis [MW71, MW80, Smi05], which incorporates transformation of specifications, inductive reasoning, recursion schemes and termination checking, but we extend it with modern SMT techniques, new search algorithms, and a new cost-based synthesis framework.

The origins of our deductive framework is in complete functional synthesis, which was used previously for integer linear arithmetic [KMPS12]. In this chapter, we do not use synthesis rules for linear integer arithmetic. Instead, here we use synthesis procedure rules for algebraic data types [Sut12a, JKS13], that were not reported in an implemented system before. This gives us building blocks for the synthesis of recursion-free code. To synthesize recursive code, we developed new algorithms, that build on and further advance the counterexample-guided approach to synthesis [SLTB⁺06], but apply it to the context of an SMT instead of SAT solver and use new approaches to control the search space.

Deductive Synthesis Frameworks

Early work on synthesis [MW71, MW80] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle.

Programming by refinement was popularized as a manual activity [Wir71, BvW98]. Interactive tools were developed to support such techniques in HOL [BGL⁺97]. A recent example of deductive synthesis and refinement is the Specware system from Kestrel [Smi05]. We were not able to use the system first-hand due to its availability policy, but it appears to favor expressive power and control, whereas we favor automation.

A combination of automated and interactive development is analogous to the use of automation in interactive theorem provers, such as Isabelle [NPW02]. However, in verification it is typically the case that the program is available, whereas the emphasis here is on constructing the program itself, starting from specifications.

Work on synthesis from specifications [SGF10] resolves some of these difficulties by decoupling the problem of inferring program-control structure and the problem of synthesizing the computation along the control edges. The work exploits verification techniques that use both approximation and lattice theoretic search along with decision procedures, but it appears to require detailed information about the structure of the expected solution more than our approach does.

Synthesis with Input/Output Examples

One of the first works that addressed synthesis with examples and put inductive synthesis on a firm theoretical foundation is the one by Summers [Sum77]. Subsequent work presents extensions of the classic approach to induction of functional Lisp-programs [KS06, Hof10]. These extensions include synthesizing a set of equations (instead of just one), multiple recursive calls, and a systematic introduction of parameters. Our current system lifts several restrictions of previous approaches by supporting reasoning about arbitrary datatypes, supporting multiple parameters in concrete and symbolic I/O examples, and by enabling nested recursive calls and user-defined declarations.

Inductive (logic) programming that explores the automatic synthesis of (usually recursive) programs from incomplete specifications, most often being input/output examples [FP01, MR94], influenced our work. Recent work in the area of programming by demonstration has shown that synthesis from examples can be effective in a variety of domains, such as spreadsheets [SG12]. Advances in the field of SAT and SMT solvers inspired counter-example guided iterative synthesis [SLTB⁺06, GJTV11], that can derive input and output examples from specifications. Our tool uses and advances these techniques through two new counterexample-guided synthesis approaches.

ESCHER, recently presented an inductive synthesis algorithm that is completely driven by input/output examples, focuses on synthesis of recursive procedures, and shares some similarities with some of our rules [AGK13]. By following the goal graph, which is similar in function as the AND/OR search tree, ESCHER tries to detect if two programs can be joined by a conditional. The split goal rule in ESCHER can speculatively split goals and is thus similar to our splitting rules. One of the differences is that ESCHER can split goals based on arbitrary choices of satisfied input/output example pairs, whereas our rules impose strictly predefined conditions that correspond to common branching found in programs. We found it difficult to compare the two frameworks because ESCHER needs to query the oracle (the user) for input/output examples each time a recursive call is encountered (in the SATURATE rule). We do not consider it practical to allow the synthesizer to perform such extensive querying, because the number of recursive calls during synthesis tends to be very large. Thus, ESCHER appears suitable for scenarios such as reverse-engineering a black-box implementation from its observable behavior more than for synthesis based on user's specification.

Our approach complements the use of SMT solvers, with additional techniques for the automatic generation of input/output examples. Our current approach is domain-agnostic, although in principle related to techniques such as Korat [BKM02] and UDITA [GGJ⁺10].

Recent works on synthesis of functional programs from input-output examples [FCD15] employ a deductive approach very similar to ours: they decompose the problem by making hypotheses about the structure of programs, and sub-problems are generated by applying a lens on input-output examples. Their approach relies on a fixed set of shapes of higher-order programs that they can combine arbitrarily to generate solutions. While their approach is

syntactically more restrictive, the higher-order combinators are expressive enough to cover a large set of applications.

Synthesis Based on Finitization Techniques

Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [SLTB⁺06, SLAT⁺07, SLJB08]. The algorithms employed in sketching are typically focused on appropriately guided searches over the syntax tree of the synthesized program. The tool we present shows one way to move the ideas of sketching towards infinite domains. In this generalization, we reason about equations as much as SAT techniques.

Reactive Synthesis

The synthesis of reactive systems generates programs that run forever and interact with the environment. Complete algorithms known for reactive synthesis work with finite-state systems [PR89] or timed systems [AMP94]. Finite-state synthesis techniques have applications for controlling the behavior of hardware and embedded systems or concurrent programs [VYY09]. These techniques usually take specifications in a fragment of temporal logic [PPS06] and result in tools that can synthesize useful hardware components [JB06]. Recently such synthesis techniques were extended to repair-tasks that preserves good behaviors [vEJ13], which is related to our notion of partial programs that have remaining **choose** statements. These techniques are applied to the component-based synthesis problem for finite-state components [LV09]. We focus on infinite domains but for simpler input/output computation model.

TRANSIT combines synthesis and model checking to bring a new model for programming distributed protocols [URD⁺13], which is a challenging case of a reactive system. The specification of a protocol is given with a finite-state-machine description augmented with snippets that can use concrete and symbolic values to capture intended behavior. Similarly to our STE rule, the main computational problem solving in TRANSIT is based on the counter-example guided inductive synthesis (CEGIS) approach, and the execution of concrete specification is used to prune the parts of the synthesis search space. Although similarities exist between the concept of progressive synthesis of guarded transitions in TRANSIT and inferring branches in our case splitting and condition abduction rules, the crucial difference is that our framework infers the entire implementation, including the control flow (with recursive calls), without the need of approximate control flow specification. However, the effectiveness of TRANSIT is increased by focusing on a particular application domain, which is the direction we will leave for the future.

Automated inference of program fixes and contracts

These areas share the common goal of inferring code and rely on specialized software synthesis techniques [WFKM11, PWF⁺11]. Inferred software fixes and contracts are usually snippets of code that are synthesized according to the information gathered about the analyzed program. The core of these techniques lies in the characterization of runtime behavior that is used to guide the generation of fixes and contracts. Such characterization is done by analyzing program state across the execution of tests; state can be defined using user-defined query operations [WFKM11], and additional expressions extracted from the code [PWF⁺11]. Generation of program fixes and contracts is done using heuristically guided injection of (sequences of) routine calls into predefined code templates.

Our synthesis approach works with purely functional programs and does not depend on characterization of program behavior. It is more general in the sense that it focuses on synthesizing whole correct functions from scratch and does not depend on already existing code. Moreover, rather than using execution of tests to define starting points for synthesis and SMT solvers just to guide the search, our approach utilizes SMT solvers to guarantee correctness of generated programs and uses execution of tests to speedup the search. Coupling of flexible program generators and the Leon verifier provides more expressive power of the synthesis than filling of predefined code schemas. In Chapter 3, we extend the synthesis framework presented here to perform repair. In that setting, exploiting existing code as well as runtime traces turns out to be crucial.

Further Related Work Synthesis remains an active research area as we write this. Interesting developments have been made since our results were first published.

The problem of finding a solution program amongst a space of candidates given by a grammar, as described in Section 2.4.1, has since been identified as a key synthesis challenge [ABJ⁺13]. A representation for such synthesis problems has been standardized (SyGuS) and competitions have been held. Sadly, the problems we target within Leon remains out of scope of most SyGuS solvers, because our specifications and solutions involve calls to recursive functions. However, we believe that this effort will drive the research towards efficient solutions for this challenging problem. While our framework is in essence more general than syntax-guided synthesis (since we only use it within certain deductive rules), most non-trivial synthesis problems are solved in part thanks to these rules. We thus believe that our approach will directly benefit from advances in syntax-guided synthesis.

The SYNAPSE[BTGC16] tool employs parallelization to simultaneously search for solutions within multiple sketches. By coordinating the intermediate results of individual searches, it is able to speed up the discovery of an optimal solution according to a user-provided cost-model. Even though Leon also follows a cost-model to reach potential solutions, its cost-model orients the search within the graph. Because it is not used within individual deduction rules, we thus cannot guarantee optimality of the end-result solution.

The increasing availability of large corpus of code made recent advances in statistical approaches for synthesis of code and specifications effective [RVY14, RVK15].

2.8 Conclusions

Software synthesis is a difficult problem, but we believe it can automate several development tasks. We have presented a new framework for synthesis: it combines transformational and counterexample-guided approaches. Our implemented system can synthesize and prove correct functional programs that manipulate unbounded data structures such as algebraic data types. We have used the system to synthesize algorithms that manipulate list and tree structures. Our approach uses the state-of-the-art SMT solving technology and an effective mechanism for solving certain classes of recursive functions. Due to this technology, we were able to synthesize programs over unbounded domains that are guaranteed to be correct for all inputs. Our automated system can be combined with manual transformations or run-time constraints-solving [KKS13], to cover the cases where static synthesis does not fully solve the problem. It can further be improved by having additional rules for manually verified refactoring and automatic synthesis steps [KMPS12], by being able to inform the search using statistical information from a corpus of code [GKKP13] and by being able to use domain-specific higher-order combinators [SNK⁺13], as well as by further improvements in decision procedures to enhance the class of verifiable programs.

3 Deductive Repair

We have seen in Chapter 2 a synthesis procedure that is able to generate implementations that satisfy a given specification. We build on these techniques and extend the synthesis framework to perform repair. We consider a function to be subject to repair if it does not satisfy its specification, expressed in the form of pre- and post-conditions. The task of repair consists in automatically generating an alternative implementation that meets the specification. The repair problem has been studied in the past for reactive and pushdown systems [JGB05, JSGB12, vEJ13, GBC06, SDE08, SOE14]. We view repair as generalizing, for example, the *choose* construct of complete functional synthesis [KMPS13], sketching [Sol13, SLTB⁺06], and program templates [SGF13], because the exact location and nature of expressions to be synthesized is left to the algorithm. Repair is thus related to localization of error causes. To speed up our approach, we do use *coarse-grained* error localization based on derived test inputs. Indeed, the repair identifies a particular change that makes the program correct. Using tests alone as a criterion for correctness is appealing for performance reasons [PWF⁺11, GNFW12, NQRC13], but this can lead to erroneous repairs. We therefore exploit our prior work on verifying and synthesizing recursive functional programs with *unbounded* data-types (trees, lists, integers) to provide strong correctness guarantees, while optimizing our technique to use automatically derived tests. By phrasing the problem of repair as one of synthesis and introducing tailored deduction rules that use the original implementation as guide, we enable the repair-oriented synthesis procedure to automatically find correct fixes, and in the worst case, to resort to re-synthesizing the desired function from scratch.

We view repair as a good example of a practical deployment of synthesis techniques. We implement repair as a particular use-case of the general synthesis procedure described in Chapter 2, to which we add the following key techniques:

Exploration of similar expressions. We present an algorithm for expression repair based on a grammar for generating expressions *similar* to a given expression (according to an error model we propose). We use such grammars within our new generic symbolic-term exploration routine that uses test inputs as well as an SMT solver, and efficiently explores the space of

expressions that contain recursive calls whose evaluation depends on the expression being synthesized.

Fault localization. To narrow down repair to a program fragment, we localize the error by doing dynamic analysis by using test inputs generated automatically from specifications. We combine two automatic sources of inputs: enumeration techniques and SMT-based techniques. We collect traces leading to erroneous executions and compute the common prefixes of branching decisions. We show that this localization is in practice sufficiently precise to repair sizeable functions efficiently.

Integration into a deductive synthesis and verification framework. Our repair system is part of a deductive verification system, so it can automatically produce new inputs from specification, prove correctness of code for all inputs ranging over an unbounded domain, and synthesize program fragments by using deductive synthesis rules that include common recursion schemas.

The repair approach offers significant improvements, compared with synthesis from scratch. Synthesis alone scales poorly when the expression to synthesize is large. Fault localization focuses synthesis on the smaller, invalid portions of the program and thus results in significant performance gains. The source code of our tool and additional details are available at <http://leon.epfl.ch>, as well as <https://github.com/epfl-lara/leon>.

3.1 Example

Consider the following functionality based on a part of a compiler. We want to transform (desugar) an abstract syntax-tree of a typed expression language into a simpler untyped language; simplifying some of the constructs and changing the representation of some of the types, preserving the semantics of the transformed expression. In Figure 3.1, the original syntax trees are represented by the class `Expr` and its subclasses, whereas the resulting untyped language trees are given by `SExpr`. A syntax tree of `Expr` either evaluates to an integer, to a Boolean, or to no value if it is not well typed. We capture this by defining a type-checking function `typeOf`, along with two separate semantic functions, `semI` and `semB`. Whereas, `SExpr` always evaluates to an integer, as defined by the `simSem` function. For brevity, most subclass definitions are omitted.

The `desugar` function translates a syntax tree of `Expr` into one of `SExpr`. We expect the function to ensure that the transformation preserves the semantics of the tree: originally integer-valued trees evaluate to the same value, Boolean-valued trees now evaluate to 0 and 1, representing **false** and **true**, respectively, and mistyped trees are left unconstrained. This is expressed in the postcondition of `desugar`.


```

abstract class Expr
case class Plus(lhs: Expr, rhs: Expr)
  extends Expr
... // 9 more subclasses

abstract class SExpr
case class SPlus(lhs: SExpr,
  rhs: SExpr) extends SExpr
... // 5 more subclasses

abstract class Type
case object IntType extends Type
case object BoolType extends Type

def typeOf(e: Expr): Option[Type] =
  ...

def seml(t: Expr): Int = {
  require(typeOf(t) == Some(IntType))
  ...
}
def semB(t : Expr) : Boolean = {
  require(typeOf(t) == Some(BoolType))
  ...
}
def simSem(e : SExpr) : Int = ...

def desugar(e: Expr) : SExpr = {
  e match {
    case Plus (lhs, rhs) =>
      SPlus(desugar(lhs), desugar(rhs))
    case Minus(lhs, rhs) =>
      SPlus(desugar(lhs), Neg(desugar(rhs)))
    case And(lhs, rhs) =>
      Slte(desugar(lhs), desugar(rhs), SLiteral(0))
    case Or(lhs, rhs) =>
      Slte(desugar(lhs), SLiteral(1), desugar(rhs))
    case Not(e) =>
      Slte(desugar(e), SLiteral(0), SLiteral(1))
    case lte(cond, thn, els) =>
      Slte(desugar(cond), desugar(els), desugar(thn))
    case IntLiteral(v) =>
      SLiteral(v)
    case BoolLiteral(b) =>
      SLiteral(if (b) 1 else 0)
    ...
  } ensuring { res => typeOf(e) match {
    case Some(IntType) =>
      simSem(res) == seml(e)
    case Some(BoolType) =>
      simSem(res) == if (semB(e)) 1 else 0
    case None() => true }
  }
}

```

Figure 3.1 – The syntax tree translation in function `desugar` has a strong **ensuring** clause, requiring the semantic equivalence of the transformed and the original tree, as defined by several recursive evaluation functions. `desugar` contains an error. Our system finds it, repairs the function, and proves the resulting program correct.

The implementation in Figure 3.1 contains a bug: the `thn` and `els` branches of the `lte` case have been accidentally switched. Using tests automatically generated by generic enumeration of small values, as well as from a verification attempt of `desugar`, our tool is able to find a coarse-grained location of the bug, as the body of the relevant case of the `match` statement. During repair, one of the rules performs a semantic exploration of expressions similar to the invalid one. It discovers that using the expression `Slte(desugar(cond), desugar(thn), desugar(els))` instead of the invalid one makes the discovered tests pass. The system can then formally verify that the repaired program meets the specification for all inputs. If we try to introduce similar bugs in the correct `desugar` function, or to replace the entire body of a case with a dummy value, the system successfully recovers the intended case of the transformation. In some cases our system can repair multiple simultaneous errors; the mechanism behind this is explained in Section 3.2.2. Note that the developer communicates with our system only by writing code and specifications, both of which are functions in an existing functional programming language. This illustrates the potential of repair as a scalable and developer-friendly deployment of synthesis in software development.

3.2 Deductive Guided Repair

We next describe our deductive-repair framework. The framework currently works under several assumptions, which we consider reasonable given the state of the art in repair of infinite-state programs. We consider the specifications of functions as correct; the code is assumed wrong if it cannot be proven correct with respect to this specification for all of the infinitely many inputs. If the specification includes input-output tests, it follows that the repaired function must have the same behavior on these tests. We do not guarantee that the output of the function is the same as the original on tests not covered by the specification, though the repair algorithm tends to preserve some of the existing behaviors due to the local nature of repair. It is the responsibility of the developer to sufficiently specify the function being repaired. Although under-specified benchmarks might produce unexpected expressions as repair solutions, we found that even partial specifications often yield the desired repairs. In our experience, a particularly effective specification style is to give a partial specification that depends on all components of the structure (for example, describes property of the set of stored elements), then additionally to provide a finite number of symbolic input-output tests. We assume that only one function of the program is invalid; the implementation of all other functions is considered valid as far as the repair of interest is concerned. Finally, we assume that all functions of the program, even the invalid one, terminate.

Stages of the Repair Algorithm. The function being repaired passes through the following stages, that we describe in the rest of the section:

- **Test generation and verification.** We combine enumeration- and SMT-based techniques to either verify the validity of the function or, if it is not valid, to discover coun-

terexamples (examples of misbehaviors).

- **Fault localization.** The incorrect implementation is provided as a hint to the synthesis problem. This enables localization rules to select smaller expressions that are executed in all failing tests, modulo recursion.
- **Synthesis of similar expressions.** After localizing the error to a set of independent branches, the synthesis procedure can use, amongst other techniques, the erroneous version as a hint to explore some of its variations.
- **Verification of the solution.** Lastly, the system attempts to prove the validity of the discovered solution. Our results, in Section 3.5 Figure 3.3, indicate in which cases the synthesized function passed the verification.

We perform repair using the framework described in Section 2.3. We show how this framework can be applied to program repair by introducing dedicated rules, as well as special predicates. We reuse the notation for synthesis tasks:

$$\llbracket \bar{a} \langle \Pi \triangleright \phi \rangle \bar{x} \rrbracket$$

To track the original (incorrect) implementation along instantiations of our deductive synthesis rules, we introduce a *guiding predicate* into the path-condition Π of the synthesis problem. We refer to this guiding predicate as $\odot[\text{expr}]$, where expr represents the original expression. Like the termination hint described in Section 2.4.1, this predicate does not have any logical meaning in the path-condition (it is equivalent to **true**), but it provides syntactic information that can be used by repair-dedicated rules. These rules are covered in detail in Section 3.2.1, Section 3.2.2 and Section 3.3.

3.2.1 Fault Localization

A contribution of our system is the ability to focus the repair problem on a small sub-part of the function’s body responsible for its erroneous behavior. The underlying hypothesis is that most of the original implementation is correct. This technique enables us to reuse as much of the original implementation as possible and minimizes the size of the expression given to subsequent more expensive techniques. Focusing also has the profitable side-effect of making repair more predictable, even in the presence of weak specifications: Repaired implementation tends to produce programs that preserve some of the existing branches, and thus have the same behavior on the executions that use only these preserved branches. To lead us to the source of the problem, we rely on the list of examples that fail the function specification: If all failing examples only use one branch of some branching expression in the program, then we assume that the error is contained in that branch. We define \mathcal{F} as the set of all inputs of collected failing tests (see Section 3.4). We describe focusing by using the following rules.

If-Focus

Given the input problem $\llbracket \bar{a} \langle \odot[\text{if}(c) \{t\} \text{ else } \{e\}] \triangleright \phi \rangle \bar{x} \rrbracket$ we first check if there is an alternative condition expression such that all failing tests succeed:

IF-FOCUS-CONDITION:

$$\frac{\begin{array}{c} \exists C. \forall \bar{i} \in \mathcal{F}. \phi[\bar{x} \mapsto \text{if}(C(\bar{a})) \{t\} \text{ else } \{e\}, \bar{a} \mapsto \bar{i}] \\ \llbracket \bar{a} \langle \odot[c] \wedge \Pi \triangleright \phi[\bar{x} \mapsto \text{if}(x') \{t\} \text{ else } \{e\}] x' \rrbracket \vdash \langle P \mid T \rangle \end{array}}{\llbracket \bar{a} \langle \odot[\text{if}(c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \text{if}(T) \{t\} \text{ else } \{e\} \rangle}$$

Instead of solving this higher-order hypothesis, we execute the function and non-deterministically consider both branches of the **if** (and do so within recursive invocations as well). If a valid execution exists for each failing test, the formula is considered satisfiable. This indicates that the if-condition might be the only source of error. We therefore focus on the condition alone.

Otherwise, we check whether c evaluates to either **true** or **false** for all failing inputs. In either case, we focus on the corresponding branch:

IF-FOCUS-THEN:

$$\frac{\begin{array}{c} \llbracket \bar{a} \langle \odot[t] \wedge c \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid T \rangle \quad \forall \bar{i} \in \mathcal{F}. c[\bar{a} \mapsto \bar{i}] \end{array}}{\llbracket \bar{a} \langle \odot[\text{if}(c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \text{if}(c) \{T\} \text{ else } \{e\} \rangle}$$

IF-FOCUS-ELSE:

$$\frac{\begin{array}{c} \llbracket \bar{a} \langle \odot[e] \wedge \neg c \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid T \rangle \quad \forall \bar{i} \in \mathcal{F}. \neg c[\bar{a} \mapsto \bar{i}] \end{array}}{\llbracket \bar{a} \langle \odot[\text{if}(c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P \mid \text{if}(c) \{t\} \text{ else } \{T\} \rangle}$$

We use analogous rules to repair **match** expressions; they are ubiquitous in our programs. For match expressions however, there might exist more than two branches. We thus focus on every branch exercised by at least one failing test:

MATCH-FOCUS:

$$\frac{\begin{array}{c} \exists \bar{i} \in \mathcal{F}. c_n[\bar{a} \mapsto \bar{i}] \quad \llbracket \bar{a} \langle \odot[b_n] \wedge c_n \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_n \mid \bar{T}_n \rangle \quad \dots \end{array}}{\llbracket \bar{a} \langle \odot[s \text{ match } \{\dots \text{ case } C_n \Rightarrow b_n \dots\}] \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle c_n \wedge P_n \mid s \text{ match } \{\dots \text{ case } C_n \Rightarrow T_n \dots\} \rangle}$$

where c_n is the condition required to reach C_n (and therefore execute b_n). In case some of the failing tests match none of the match conditions, we inject an additional “or-else” case (**case** $_ \Rightarrow ???$), that matches everything that was not matched by all the defined cases. This enables us to repair programs where the error is due to an incomplete match construct.

All the above rules use tests to locally approximate the validity of branches. They are sound only if the set of tests \mathcal{F} is sufficiently large and representative. Our system therefore performs an end-to-end verification for the complete solution, thus ensuring the overall soundness.

3.2.2 Guided Decompositions

In case the focusing rules fail to identify a single branch of an **if**-expression, we might still benefit from reusing the **if**-condition. To this end, we introduce a rule analogous to focus, that decomposes based on the guide.

$$\text{IF-SPLIT:} \quad \frac{\llbracket \bar{a} \langle \odot[t] \wedge c \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_1 \mid T_1 \rangle \quad \llbracket \bar{a} \langle \odot[e] \wedge \neg c \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle P_2 \mid T_2 \rangle}{\llbracket \bar{a} \langle \odot[\text{if}(c) \{t\} \text{ else } \{e\}] \wedge \Pi \triangleright \phi \rangle \bar{x} \rrbracket \vdash \langle (c \wedge P_1) \vee (\neg c \wedge P_2) \mid \text{if}(c) \{T_1\} \text{ else } \{T_2\} \rangle}$$

3.2.3 Synthesis within Repair

The repair-specific rules described earlier solve repair problems according to the error model. Due to integration into the Leon synthesis framework, general synthesis rules also apply, which enables the repair of more intricate errors. This achieves an appealing combination between fast repairs for predictable errors and expressive, albeit slower, repairs for more complicated errors.

3.3 Counterexample-Guided Similar-Term Exploration

After following the overall structure of the original problem, it is often the case that the remaining erroneous branches can be fixed by applying small changes to their implementations. For instance, an expression calling a function might be wrong only in one of its arguments or it might have two of its arguments swapped. We exploit this assumption by considering different variations to the original expression. Due to the lack of a large code base in the PureScala subset that Leon handles, we cannot use statistically informed techniques such as [GKKP13], so we define an error model following our intuition and experience from previous work.

We use the notation $G(\text{expr})$ to denote the space of variations of expr and define it in the form of a grammar as combination of three variations:

$$G(\text{expr}) ::= G_{\text{swap}}(\text{expr}) \mid G_{\text{arg}}(\text{expr}) \mid G_{|2|}(\text{expr})$$

with the following forms of variations.

Swapping arguments. We consider here all the variants of swapping two arguments that are compatible in terms of type. For instance, for an operation with three operands of the same type:

$$G_{\text{swap}}(\text{op}(a, b, c)) ::= \text{op}(b, a, c) \mid \text{op}(a, c, b) \mid \text{op}(c, b, a)$$

Generalizing one argument. This variation corresponds to making a mistake in only one argument of the operation we generalize:

$$G_{arg}(op(a,b,c)) ::= op(G(a),b,c) \mid op(a,G(b),c) \mid op(a,b,G(c))$$

Bounded arbitrary expression. We consider a grammar of interesting expressions of a given type and of limited depth. This grammar considers all operations in scope as well as all input variables, similar to the bounded grammars described in Section 2.4.1 Finally, it includes the guiding expression as a terminal, which corresponds to possibly wrapping the source expression in an operation.

Our grammars cover a range of variations that correspond to common errors. During synthesis, we instantiate STE with this specialized grammar. The input examples, collected for localizing the error, are also immediately available within STE. Even though this rule is inherently incomplete, it is able to fix common errors efficiently. Our deductive approach allows us to introduce such tailored rules without loss of generality: errors that go beyond this model could be repaired using more general, albeit slower synthesis rules. The prioritization of rules available in Section 2.3 enables us to first apply repair rules, which guarantees that we try to localize the problem before solving it.

3.4 Generating and Using Tests for Repair

Tests play an essential role during repair, allowing us to gather information about the valid and invalid parts of the function. In this section we elaborate on how we select, generate, and filter examples of inputs and possibly outputs. Several components of our system then make use of these examples. We distinguish two kinds of tests: input tests and input-output tests. Specifically, input tests provide valid inputs for the function according to its precondition, whereas input-output tests also specify the exact output that corresponds to each input.

3.4.1 Extraction and Generation of Tests

Our system relies on multiple sources for generating input tests that have been previously described in detail in Section 2.5:

1. *User-provided symbolic input-output tests:* Having partially symbolic input-output examples strikes a good balance between literal examples and full-functional specifications. They permit the specification of generic functions naturally and inline with the implementation, where traditional test frameworks would exercise specific instantiations of the function in a test harness located elsewhere.

This is the only source of tests that constraint the output. Indeed, we cannot rely on output values obtained by executing the function, without taking the risk of over-

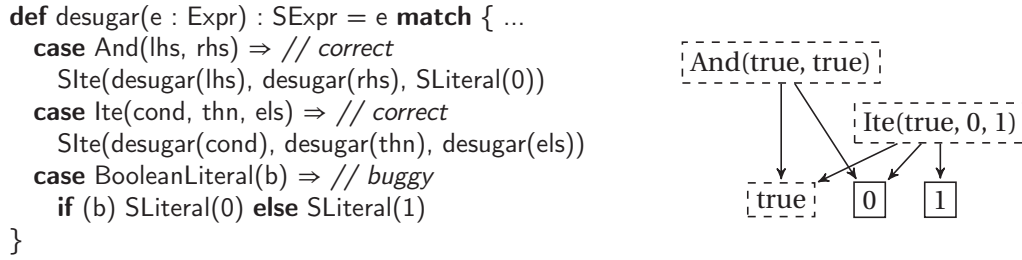


Figure 3.2 – Code and invocation graph for desugar. Solid borderlines stand for passing tests, dashed ones for failing ones. Type constructors for literals have been omitted.

committing: even if the output value satisfies the specification (making it a valid inputs-output pair), registering it as the only valid output value for the provided inputs is over-constraining the problem, and might prevent us from finding valid repairs.

2. *Generated Input Tests:* We rely on enumeration techniques to generate inputs that satisfy the precondition of the function. Using a generate and test approach, we gather up to 400 valid input tests in the first 1000 enumerated.
3. *Solver-Generated Tests:* We rely on the underlying solvers for recursive functions of Leon to generate verification counter-examples. Given that the function is invalid and that it terminates, the solver (which is complete for counter-examples) is guaranteed to eventually provide us with at least one failing test.

3.4.2 Classifying and Minimizing Traces

We partition the set of collected tests into passing and failing sets. A test is considered as failing if it violates a precondition, a postcondition, or emits one of various other kinds of runtime errors when the function to repair is executed on it. In the presence of recursive functions, a given test might fail within one of its recursive invocations. It is interesting in such scenarios to consider the arguments of this specific sub-invocation: they are typically smaller than the original and are better representatives of the failure. To clarify this, consider the example in Figure 3.2 (based on the program in Figure 3.1):

Assume the tests collected are

- And(BooleanLiteral(**true**), BooleanLiteral(**true**)),
- Itte(BooleanLiteral(**true**), IntLiteral(0), IntLiteral(1)), and
- BooleanLiteral(**true**)

When executed with these tests, the function produces the graph of eval invocations shown on the right of Figure 3.2. A trivial classification tactic would label all three tests as faulty, even

though it is obvious that all errors can be explained by the bug in `BooleanLiteral`, due to the dependencies between tests. More generally, *a failing test should also be blamed for the failure of all other tests that invoke it transitively*. Our framework deploys this classification. Thus, in our example, it would only label `BooleanLiteral(true)` as a failing example, which would lead to the correct localization of the problem on the faulty branch. Note that this process discovers new failing tests not present in the original test set if they occur as recursive sub-invocations.

Our experience with incorporating tests into the Leon system indicate that they prove time and again to be extremely important for the tool’s efficiency and complement well our existing verification approaches. In addition to allowing us to detect errors sooner and filter out wrong synthesis candidates, tests also allow us to quickly find the approximate error location.

3.5 Evaluation

We evaluate our implementation on a set of benchmarks in which we manually injected errors (Figure 3.3). The programs mainly focus on data structure implementations and syntax tree operations. Each benchmark is comprised of algebraic data-type definitions and recursive functions that manipulate them, specified using strong yet still partial preconditions and post-conditions. We manually introduced errors of different types in each copy of the benchmarks. We ran our tool unassisted until completion to obtain a repair, providing it only with the name of the file and the name of the function to repair (typically the choice of the function could also have been localized automatically by running the verification on the entire file). The experiments were run on an Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz with 16GB RAM, with 2GB given to the Java Virtual Machine. Although the deductive reasoning supports parallelism in principle, our implementation is currently single-threaded.

For each benchmark of Figure 3.3 we provide (1) the name of the benchmark and the broken operation (2) a short classification of the kind of error introduced. The error kinds include a small variation of the original program, a completely faulty **match**-case, a missing **match**-case, a missing necessary **if**-split, a missing function call, and finally, two separate variations in the same function. We describe the relevant sizes (counted in abstract syntax tree nodes) of (3) the overall benchmark, (4) the erroneous function, (5) the localized error, and (6) the repaired expression. The full size of the program is relevant because our repair algorithm can introduce calls to any function defined in the benchmark, and because the verification of a function depends on other functions in the file (recall Figure 3.1). We also include the time, in seconds, our tool took (7) to collect and classify tests, and (8) to repair the broken expression. Finally, we report (9) if the system could formally (and automatically) prove the validity of the repaired implementation. Our examples are challenging to verify, let alone repair. They contain both functional and test-based specifications to capture the intended behavior. Many rely on unfolding procedure of [SKK11, Sut12a] to handle contracts that contain other auxiliary recursive functions. The fast exponentiation algorithm of `Numerical.power` relies on the non-linear reasoning of the Z3 SMT solver [dMB08].

Operation	Error	Size				Time (sec)		Proof Success
		Prg	Fun	Err	Fix	Test	Repair	
Compiler.desugar1	full case	634	81	3	5	1.2	2.7	✓
Compiler.desugar2	full case	632	79	2	6	1.3	3.1	✓
Compiler.desugar3	variation	636	83	7	7	0.9	1.7	✓
Compiler.desugar4	variation	636	83	7	7	1.7	2.5	✓
Compiler.desugar5	2 variations	636	83	14	14	2.0	2.9	✓
Compiler.simplify1	variation	636	30	4	1	0.8	1.6	✓
Heap.merge1	if cond	314	36	3	3	2.8	7.1	✓
Heap.merge2	full case	314	36	1	1	1.3	1.9	✓
Heap.merge3	if cond	314	36	3	3	2.8	6.8	✓
Heap.merge4	variation	314	36	6	6	1.3	5.7	✓
Heap.merge5	if cond	316	38	5	7	1.7	10.6	✓
Heap.merge6	2 variations	314	36	2	2	2.1	2.6	✓
Heap.insert	variation	316	8	8	10	10.2	7.2	✓
Heap.makeNode	variation	316	16	7	7	2.6	6.6	✓
List.pad	variation	736	34	8	6	2.2	2.6	✓
List.++	variation	648	9	3	5	2.2	3.1	✓
List.:+	full case	679	11	1	3	1.8	1.7	✓
List.replace	if cond	721	25	4	3	1.3	20.1	✓
List.count	variation	681	22	4	3	2.1	6.1	✓
List.find1	variation	733	16	3	5	1.0	2.1	✓
List.find2	variation	655	21	2	4	2.9	4.0	✓
List.find3	if cond	708	23	4	6	2.9	3.8	✓
List.size	variation	709	24	4	3	4.7	14.5	✓
List.sum	variation	683	10	4	4	1.5	1.5	✓
List.delete	missing call	681	10	4	4	1.6	1.8	✓
List.drop1	if cond	681	16	1	3	1.3	2.2	✓
List.drop2	variation	721	22	3	5	1.3	4.9	✓
Numerical.power	variation	142	23	5	7	0.5	2.1	✓
Numerical.moddiv	variation	144	30	3	1	0.5	1.5	✓
MergeSort.split	full case	222	30	5	2	3.1	2.1	✓
MergeSort.merge1	variation	224	32	5	5	1.7	2.3	✓
MergeSort.merge2	if cond	224	32	3	3	1.7	3.8	✓
MergeSort.merge3	variation	222	30	3	5	1.7	2.1	✓
MergeSort.merge4	variation	224	32	1	1	1.3	1.6	✓

Figure 3.3 – Automatically repaired functions using our system. We provide for each operation a small description of the kind of error introduced, the overall program size, the size of the invalid function, the size of the erroneous expression we locate, and the size of the repaired version. We then provide the times our tool took to gather and classify tests, and to repair the erroneous expression. Finally, we mention if the resulting expression can be verified. The source of all benchmarks can be found in the appendix.

An immediate observation is that fault localization is often able to focus the repair to a small subset of the body. Combined with the symbolic term exploration, this translates to a fast repair if the error falls within the error model. Among the hardest benchmarks are those labeled as having “two variations”. For example, `Compiler.desugar5` is similar to one in Figure 3.1 but contains two errors. In those cases, localization returns the entire **match** as the invalid expression. Our guided repair uses the existing **match** as the guide and successfully resynthesizes code that repairs both erroneous branches. Another challenging example is `Heap.merge3`, for which the more elaborate *If-Focus-Condition* rule of Section 3.2.1 kicks in to resynthesize the condition of the **if** expression.

The repairs listed in evaluation are not only valid according to their specification, but were also manually validated by us to match the intended behavior. A failing proof thus does not indicate a wrong repair, but rather that our system was not able to automatically derive a proof of its correctness, often due to insufficient inductive invariants. We identify three scenarios under which repair itself might not succeed: (1) if the assumptions mentioned in Section 3.2 are violated, (2) when the necessary repair is either too big or outside of the scope of general synthesis, or (3) if test collection does not yield sufficiently many interesting failing tests to locate the error.

3.6 Further Related Work

Much of the prior work on diagnosis and repair focused on imperative programming, without native support for algebraic data types, making it typically infeasible to even automatically verify data structure properties of the kind that our benchmarks contain. The Syntax-guided synthesis format [ABJ⁺13, ABD⁺14] does not support algebraic data types, or a specific notion of repair (it could be used to specify some of the sub-problems that our system generates, such those of Section 3.3). Another approach aims at diagnosing errors in submissions from students [SGS13]. It also phrases the problem of repair as synthesis of program fixes, which in turn become feedbacks. They rely on sketching to explore the space of program fixes extracted from an error model. The error model is tailored for individual programs and requires expert knowledge about the solution as well as the kind of errors to be expected. Unfortunately, their approach does not seem to rely on runtime-execution to filter out invalid fixes, even though the setting allows for a large set of test-cases to be generated or provided.

GenProg [GNFW12] and SemFix [NQRC13] accept as input a C program, along with user-provided sets of passing and failing test cases, but no formal specifications. Our technique for fault localization is not applicable to a sequential program with side-effects, and these tools employ statistical fault localization techniques based on program executions. GenProg applies no code synthesis, but tries to repair the program by iteratively deleting, swapping, or duplicating program statements, according to a genetic algorithm. Whereas SemFix uses synthesis, but does not take into account the faulty expression while synthesizing. Staged Program Repair [LR15] repairs large imperative programs by instantiating parametrized transformations to

error locations that are determined by running positive and negative tests. For each tentative transformation, it then attempts to synthesize the appropriate parameters to validate the repair. Note that these parameters can be expressions (e.g. conditions). This approach is however limited by the parameterized transformation schemas provided to the system, and only allows one repair pattern to apply for each defect. AutoFix-E/E2 [PWF⁺11] operates on Eiffel programs equipped with formal contracts. Formal contracts are used to automatically generate a set of passing and failing test cases, but not to verify candidate solutions. AutoFix-E uses an elaborate mechanism for fault localization; it combines syntactic, control flow and statistical dynamic analysis. It follows a synthesis approach with repair schemas that reuse the faulty statement (e.g. as a branch of a conditional). Samanta et al. [SOE14] propose abstracting a C program with a Boolean constraint, repairing this constraint so that all assertions in the program are satisfied by repeatedly applying to it update schemas according to a cost model, then concretizing the Boolean constraint back to a repaired C program. Their approach needs developer intervention to define the cost model for each program, as well as for the concretization step. Logozzo et al. [LB12] present a repair suggestion framework based on static analysis provided by the CodeContracts static checker [FL11]; the properties checked are typically simpler than those in our case. In [GMK11], Gopinath et al. repair data structure operations by choosing an input that exposes a suspicious statement, then by using a SAT-solver to discover a corresponding concrete output that satisfies the specification. This concrete output is then abstracted to various possible expressions to yield candidate repairs that are filtered with bounded verification. In their approach, Chandra et al. [CTBB11] consider an expression as a candidate for repair if substituting it with some concrete value fixes a failing test.

Repair has also been studied in the context of reactive and pushdown systems with otherwise finite control [JGB05, JSGB12, vEJ13, GBC06, SDE08, SOE14]. In [vEJ13], the authors generate repairs that preserve explicitly subsets of traces of the original program, in a way strengthening the specification automatically. We deal with the case of functions from inputs to outputs equipped with contracts. In case of a weak contract we provide only heuristic guarantees that the existing behaviors are preserved, arising from the tendency of our algorithm to reuse existing parts of the program.

3.7 Conclusions

We have presented an approach to program repair of mutually recursive functional programs, building on top of a deductive synthesis framework. The starting point gives it the ability to verify functions, find counterexamples, and synthesize small fragments of code. When doing repair, it has proven fruitful to first localize the error and then perform synthesis on a small fragment. Tests prove to be very useful in performing such localization, as well as for generally accelerating synthesis and repair. In addition to deriving tests by enumeration and verification, we have introduced a specification construct that uses pattern matching to describe symbolic tests, from which we efficiently derive concrete tests without invoking a full-fledged verification. In the case of tests for recursive functions, we perform dependency

analyses and introduce new ones to better localize the cause of the error. Although localization of errors within conditional control flow can be done by analyzing test runs, the challenge remains to localize change inside large expressions with nested function calls. We have introduced the notion of *guided synthesis* that uses the previous version of the code as a guide when searching for a small change to an existing large expression. The use of a guide is very flexible, and enables us to repair multiple errors in some cases.

Our experiments with benchmarks of hundreds of syntax tree nodes, including tree transformations and data structure operations, confirm that repair is more tractable than synthesis for functional programs. The existing (incorrect) expression provides a hint about useful code fragments from which to build a correct solution. Compared to unguided synthesis, the common case of repair remains more predictable and scalable. At the same time, the developer needs not learn a notation for specifying holes or templates. Therefore, we believe that repair is a practical way to deploy synthesis in software development.

4 Effect Analysis for Programs with Callbacks

We have seen in previous chapters that programs written in PureScala have interesting properties that we can exploit to develop useful tools for developers. However, one of the most common programming style for Scala uses predominantly functional computation steps, including higher-order functions, with a disciplined use of side-effects. An opportunity for parallel execution further increases the potential of this style. Although higher-order functions have always been recognized as a pillar of functional programming, they have also become a standard feature of object-oriented languages such as C# (in the form of *delegates*), the 2011 standard of C++, and Java 8. Moreover, design patterns popular in the object-oriented programming community also rely on callbacks, for instance the *strategy pattern* and the *visitor pattern* [GHJV94].

Because PureScala is a purely functional language, it is unlikely that an existing Scala project can be fully handled by Leon. This hinders the applicability of Leon to entire applications. Leon can however be used on selected components of an application. For instance, Leon applies particularly well to immutable data-structures. A successful approach is thus to structure the application so that certain components are purely-functional. Leon would then be able to verify them, and the rest of the application could trust that these components behave as expected. However, we require that these components are self-contained, limiting the way they can interact with the application. In other words, for a given function to be Leon-compatible, its transitive call-graph must contain only Leon-compatible functions. This is often too restrictive, especially in the presence of higher-order functions. To loosen this requirement, we introduce *external functions*, which offers a way for Leon to interact with arbitrary Scala code.

```
def leonFun(a: BigInt, b: BigInt) = {  
  require(a > 0 && b > 0)  
  otherFun(a+b)  
} ensuring { _ > a }
```

```
@extern  
def otherFun(x: BigInt): BigInt = {
```

```
    require(x > 0)
    // arbitrary scala body
  } ensuring { _ > x }
```

An external function is a Scala function whose body cannot be fully translated to Leon. The `@extern` annotation tells Leon to ignore extraction errors, which are likely to occur since the function may use arbitrary Scala features. In Scala terms, the specification is provided as part of the function body. We however distinguish the expression computing the resulting value (that we refer to here as body) from the specification (pre- and post-conditions). When possible, our extraction procedure preserves the specification of external functions.

Due to the arbitrary nature of the body of external functions, Leon is unable to guarantee that these functions follow their specifications. However, the specification can be used to properly reason about the usage of external functions: We check that pre-conditions hold when calling an external function from a tractable fragment, and we assume that the post-condition holds for their return values.

Our procedure for satisfiability modulo recursive functions therefore treats external functions as uninterpreted. In contrast to normal functions, the solver may not unfold the function body to refine the formulas. But the theory of uninterpreted functions assumes referential transparency:

$$\bar{a}_1 = \bar{a}_2 \implies f(\bar{a}_1) = f(\bar{a}_2)$$

Although this mathematical behavior of functions is typically implied in functional programming languages, it does not hold for arbitrary external functions. This threatens the soundness of verification in Leon, because the behavior of the external function is not properly captured by the corresponding solver formulas.

In this chapter, we describe an analysis of side-effect that will be able to characterize external functions depending on their effects. As a result, we identify functions that maintain referential transparency, allowing them to be called from Leon functions without special handling.

A precise analysis of side-effects is essential for automated, as well as manual, reasoning about such programs. The combination of callbacks and mutation makes it difficult to design an analysis that is both scalable enough to handle realistic code bases and precise enough to handle common patterns such as local side-effects and initialization; these patterns arise both from manual programming practice and compilation of higher-level concepts. Among key challenges are flow-sensitivity and the precise handling of aliases, as well as the precise and scalable handling of method calls.

We support not only automated program analyses and transformations that rely on effect

information, but also program understanding tasks. We therefore generate readable effect-summaries that developers can compare to their intuition about what methods should and should not affect in a program heap. Such summaries must go beyond a pure/impure dichotomy, and should ideally capture the exact frame condition of the analyzed code-fragment – or at least an acceptable over-approximation.

In this chapter, we present the design, implementation, and evaluation of a new static analysis for method side-effects; this analysis is precise and scalable even in the presence of callbacks, including higher-order functions. The key design aspects of our analysis include the following:

- a relational analysis domain that computes summaries of code blocks and methods by tracking flow-sensitive side-effects and performing strong updates;
- a framework for relational analyses that compute higher-order relational summaries of method calls; the summaries are parameterized by the effects of the methods being called;
- an automated effect-classification and presentation of effect abstractions in terms of regular expressions to facilitate their understanding by developers.

Our static analyzer, called *Insane* (INterprocedural Static ANalysis of Effects) is publicly available from

<https://github.com/epfl-lara/insane>

We evaluate *Insane* on the full Scala standard library, that is widely used by all Scala programs, and is also publicly available. Our analysis works on a relatively low-level intermediate representation that is close to Java bytecodes. Despite this low-level representation, we were able to classify most method calls as not having any observational side-effects. Moreover, our analysis also detects conditionally pure methods, for which purity is guaranteed provided that a specified set of sub-calls are pure. We also demonstrate the precision of our analysis on a number of examples that use higher-order functions as control structures. We are not aware of any other fully automated static analyzers that achieve this precision while maintaining reasonable performance.

4.1 Overview of Challenges and Solutions

In this section, we present some of the challenges that arise when analyzing programs written in a higher-order style and how *Insane* can tackle them.

Effect attribution. The problem of correctly attributing heap effects is specific to higher-order programs. Consider a simple class and a (first-order) function:

Chapter 4. Effect Analysis for Programs with Callbacks

```
class Cell(var visited : Boolean)
```

```
@extern
def toggle(c : Cell) = {
  c.visited = !c.visited
}
```

Any reasonable analyses for effects would detect that `toggle` potentially alters the heap, as it contains a statement that writes to a field of an allocated object. This effect could informally be summarized as “*toggle may modify the .value field of its first argument*”. This information could in turn be retrieved whenever `toggle` is used. Consider now the function

```
def apply(c : Cell, f : Cell⇒Unit) = {
  f(c)
}
```

where $\text{Cell} \Rightarrow \text{Unit}$ denotes the type of a function that takes a `Cell` as argument and returns no value. What is the effect of `apply` on the heap? Surely, `apply` potentially has all the effects that `toggle` has, as the call `apply(c, toggle)` is equivalent to `toggle(c)`. It also potentially has no effect on the heap at all, e.g. if invoked as `apply(c, (cell⇒()))`. The situation can also be much worse, for instance in the presence of global objects that might be modified by `f`. In fact, in the absence of a dedicated technique, the only sound approximation of the effect of `apply` is to state that it can have any effects. This approximation is of course useless, both from the perspective of a programmer, who does not learn anything about the behaviour of `apply`, and for the perspective of a broader program analysis, where the effect cannot be reused modularly.

The solution we propose in this chapter is, intuitively, to define the effect of `apply` to be “*exactly the effect of calling its second argument with its first as a parameter*”. To support this, we extend the notion of effect to be expressive enough to represent *control-flow graphs* where edges can themselves be effects. In the context of `Insane` we apply this idea to a domain designed for tracking heap-effects (described in Section 4.2), although the technique applies to any relational analysis, as we show in Section 4.3.

Equipped with this extended notion of effects, we can classify methods as *pure*, *impure*, and *conditionally pure*. The `apply` function falls into this last category: it is pure, as long as the methods called from within it are pure as well (in this case, the invocation of `f`). Notable examples of conditionally pure functions include many of the standard higher-order operations on structures that are used extensively in functional programs (`map`, `fold`, `foreach`, etc.). As an example, a typical implementation of `foreach` on linked lists is the following:

```
class LinkedList[T](var hd : T, var tl : LinkedList[T]) {
  @extern
  def foreach(f : T ⇒ Unit) : Unit = {
    var p = this
    do {
```



```

    f(p.hd)
    p = p.tl
  } while(p != null)
}

```

Correctly characterizing the effects of such functions is essential to analyzing programs written in a language such as Scala.

Making sense of effects. Another challenge we address in this chapter is one of presentation: when a function is provably pure, this can be reported straightforwardly to the programmer. When, however, it can have effects on the heap, the pure/impure dichotomy falls short. Consider a function that updates all (mutable) elements stored in a linked list:

```

@extern
def update(es : LinkedList[Cell]) = {
  es.foreach(c => c.visited = true)
}

```

Because the closure passed to `foreach` has an effect, so does the overall function. Although a summary stating only that it is impure would be highly unsatisfactory: Crucially, it would not give any indication to the programmer that the structure of the list itself cannot be affected by the writes. As we will see, the precise internal representation of effects, even though suited to a compositional analysis, is impractical for humans, not the least because it is non-textual. We propose to bridge this representation gap by using an additional abstraction of effects in the form of regular expressions that describe sets of fields potentially affected by effects (see Section 4.4). This abstraction captures less information than the internal representation but can readily represent complex effect scenarios. For the example given above, the following regular expression is reported to the programmer:

```
es.tl)*.hd.visited
```

It shows that the fields affected are those *reachable* through the list (by following chains of `.tl`) but belonging to elements only, thus conveying the desired information. In Section 4.5.3, we further demonstrate this generation of human-readable effect summaries on a set of examples that use the standard Scala collections library.

4.2 Effect Analysis for Mutable Shared Structures

The starting point for our analysis is the effect analysis [Sal06, WR99]. We here present an adaptation to our setting, with the support for strong updates, which take into account statement ordering for mutable heap operations. In the next section, we lift this analysis to the case of programs with callbacks (higher-order programs), for which most existing analyses

Statement	Meaning
$v = w$	assign w to v
$v = o.f$	read field $o.f$ into v
$o.f = v$	update field $o.f$ with v
$v = \text{new } C$	allocate an object of class C and store the reference to it in v
$v = o.m(a_1, \dots, a_n)$	call method m of object o and store the result in v

Figure 4.1 – Program statements \mathcal{P} considered in the target language.

are imprecise. We thus obtain a unique combination of precision, both for field updates and for higher-order procedure invocations.

We start by describing a target language that is expressive enough to encode most of the intermediate representation of Scala programs that we analyze.

4.2.1 Intermediate Language Used for the Analysis

The language we target is a typical object-oriented language with dynamic dispatch. A program is made of a set of classes \mathcal{C} that implement methods. We uniquely identify methods by using the method name prefixed with its declaring class as in $C.m$ and denote the set of methods in a program \mathcal{M} . Our intermediate language has no ad-hoc method overloading, because the affected methods can always be renamed after type checking. We assume that, for each method, a standard control-flow graph is available, where edges are labeled with simple program statements. Each of these graphs contains a source node *entry* and a sink node *exit*. Figure 4.1 lists the statements in our intermediate language, along with their meaning.

Because of dynamic dispatch, a call statement can target multiple methods, depending on the runtime type of the receiver object. For each method call $o.m()$, we can compute a superset of targets $\text{targets}(o.m) \subseteq \mathcal{M} \cup \{?\}$ using the static type of the receiver. If the hierarchy is not bounded through **final** classes or methods, we also include the special "?" target to represent the arbitrary methods that could be defined in unknown extensions of the program. Hence, we do not always assume access to the entire program: this assumption is defined as a parameter of the analysis, and we will see in Section 4.3.2 how this parameter affects it.

4.2.2 Effects as Graph Transformers

We next outline our graph-based representation of compositional effects. Our approach is related to the representation originally used for escape analysis [SR05, Sal06]. The meaning of such an effect is a relation on program heaps, that over-approximates the behavior of a fragments of code (e.g. methods). Section 4.3 lifts this representation to a more general, higher-order settings, which gives our final analysis.

```

class List(var elem: Int,
           var nxt: List = null)

def prepend(lst: List, v: Int) {
  lst.nxt = new List(lst.elem, lst.nxt)
  lst.elem = v
}
    
```

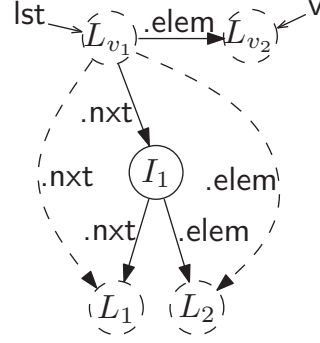


Figure 4.2 – Example of a graph representing the effects of `prepend`. Read edges lead to load nodes that represent unknown values, and solid edges represent field assignments.

Figure 4.2 shows an example of a simple function and its resulting graph-based effect. In this graph, L_{v_1} and L_{v_2} represent unknown local variables, here the parameters of the function. I_1 is an inside node corresponding to an object allocation. L_1 and L_2 are two load nodes that reference values for fields of L_{v_1} that are unknown at this time. Although read (dashed) edges do not strictly represent effects, they are necessary to resolve the meaning of nodes when composing this effect at call-sites.

In general, our effect-graphs are composed of nodes representing memory locations. We distinguish three kinds of nodes. *Inside nodes* are allocated objects. Because we use the allocation-site abstraction for these, we associate with them a flag indicating whether the node is a singleton or a summary node. *Load nodes* represent unknown fields. They model accesses to unknown parts of the heap; supporting them is a crucial requirement for modular effect-analyses. Graphs also contain special nodes for unresolved *local variables*, such as parameters.

We also define two types of edges labeled with fields. *Write edges*, represented by a plain (solid) edge in the graphical representation, and *read edges*, represented by dashed edges. Read edges provide an access path to past or intermediate values of fields, and are used to resolve *load* nodes. *Write edges* represent *must-write* modifications. Multiple write-edges for the same label with the same sources indicates that the field *must* be updated to one of the alternatives. Along with the graph, we also keep a mapping from local variables to sets of nodes.

Our analysis directly defines rules to compute the composition of any effect-graph with a state-ment that makes an individual heap modification. It is also possible to represent the meaning of each individual statement as an effect-graph itself; the result of executing statements on a current effect graph then corresponds to *composing* two effect-graphs. However, the main need for composition arises in the modular analysis of function calls.

4.2.3 Composing Effects

Composition is a key component of most modular analyses. It is typically required for inter-procedural reasoning. In our setting, it also plays an important role as a building block in our analysis framework for programs with callbacks, which we describe in Section 4.3. We now describe how composition applies to effect-graphs. This operation is done in a specific direction: we say that an *inner* effect-graph is applied to an *outer* effect-graph. The merging of graphs works by first constructing a map from inner nodes to equivalent outer nodes. This map, initially incomplete, expands during the merging process.

Importing inside nodes. The first step of the merging process is to import inside nodes from the inner graph to the outer graph. We specialize the labels that represent their allocation sites to include the label corresponding to the point at which we compose the graphs. This property is crucial for our analysis because case-classes, an ubiquitous feature of Scala, rewrite to factory methods. Once the refined label is determined, we check whether we import a singleton node into an environment in which it already exists. In such case, the node is imported as a summary node. In our example displayed in Figure 4.3, I_3 is imported as a singleton I'_3 .

Resolving load nodes. When merging two graphs, the next important operation is the resolution of load nodes from the inner graph to nodes in the outer graph. The procedure works as follows: for each inner load node we look at all its source nodes, by following read edges in the opposite direction. Note that the source node of a load node might be a load node itself, in which case we recursively invoke the resolution operation. Using this map, we then compute the nodes in the outer graph that correspond to the source nodes.

The resolution follows by performing a read operation from the corresponding source nodes in the outer graph. Once a load node is resolved to a set of nodes in the outer graph, the equivalence map is updated to reflect this. In our example, L_2 is identified as read from L_{v2} through $.g$. We consequently read $.g$ from the corresponding node L_{v1} , obtaining $\{I_1, L_1\}$.

Applying write effects. Given the map obtained by resolving load nodes, we apply write edges found in the inner graph to the corresponding edges in the outer graph. A strong update might not be able to remain strong in the outer-graph, depending on how the source node is resolved. In our example, L_2 is resolved to $\{I_1, L_1\}$. It would be incorrect to perform a strong update on both of these objects. The write effect becomes a weak update; hence the old value of $L_1.f$ is introduced. Our support for strong updates enables us to treat setter methods or constructors precisely without specific dedicated techniques. Imprecisions due to the abstract domain may however force us to treat a strong-update as a weak-update when merging the effect of a function call.

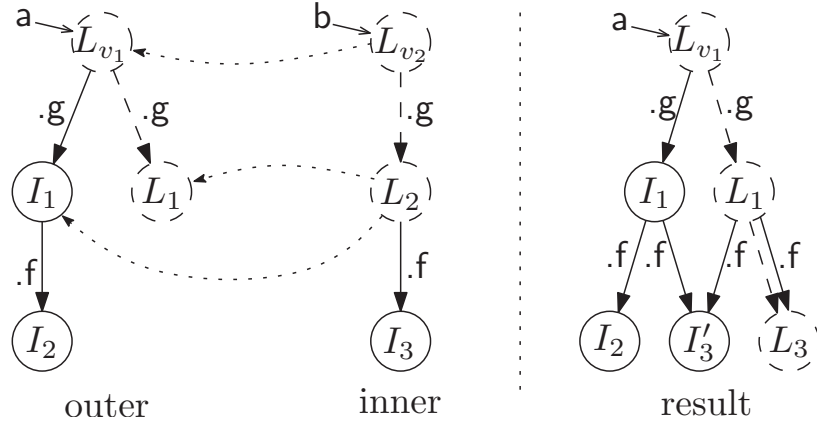


Figure 4.3 – Merging a graph with load nodes and strong updates in a context that does not permit a strong update. Inside nodes are imported after refining their label.

```

class A(var x: Int)
def f1(a1: A, a2: A) {
    val a = if(..) a1 else a2
    f2(a)
}

def f2(a: A) {
    a.x = 2
    a.x = 3
}
    
```

Figure 4.4 – Example of strong update being delayed as much as possible

The composition not only executes the last two steps, but repeats them until the convergence of the outer graph. Once a fixpoint is reached, we have successfully applied the full meaning of the inner graph to the outer graph. Such application until the fixpoint is crucial for correctness in the presence of unknown aliasing and strong updates. We illustrate this merging operation in Figure 4.3.

4.2.4 Local Strong Updates

When computing the effects of the function `f2` in Figure 4.4, we illustrate the need for local strong updates. Because the argument `a` is essentially unknown (i.e. `a` may point to many objects), a traditional conservative analysis would typically treat both writes to `a.x` as *weak updates*. As a result, the analysis would conclude that `a.x` can be equal to 2, 3, or its old value.

In our approach, these updates are treated as *strong*. The resulting effect summary will thus be that `a.x` is strongly updated to 3. Although this technique improves the precision of the analysis dramatically, it does require additional checks when applying an effect, because an update previously considered as strong can become weak. Indeed, when applying the effect of `f2` into `f1`, the write operation becomes weak, because it is applied on two distinct nodes, the nodes for `a1` and `a2`. We note however that, unlike the conservative approach, the write effect of `a.x = 2` has been fully overridden by `a.x = 3`.

4.2.5 Application to the Analysis of Real-World Scala Code

Analyzing real-world Scala code adds practical complications that were not directly embedded into the target language described in Section 4.2.1 and thus not originally accounted for. Here we list several of these complications and explain how we handled them.

Java dependencies. The Scala library depends heavily on the Java library, it is thus crucial for our analysis to handle Java code as well. Although the Scala compiler does not compile Java source-code, it does provide an utility to read Java byte-code into one of its intermediate representation used later in the pipeline. We were able to convert this stack-based intermediate representation into our control-flow graphs, which enables us to analyze the Java dependencies. However, due to technical limitations in this utility, some Java classes fail to parse and thus not all dependencies are available. In such cases, the calls to those unknown dependencies get delayed, just like imprecise method-calls, and eventually yield conditional summaries.

Native code. Having access to all Java dependencies in the available jar files is not sufficient to completely analyze the Scala library. Both the Java and Scala library rely on classes that are implemented as native (C) libraries, or by the JVM itself. This is the case, for instance, for arrays. In order to handle them correctly, *Insane* relies on custom, mock implementations for these native classes and methods: it intercepts the calls to native methods and redirects them to the stub implementations. Arrays, for example, are implemented as an instance of a class containing one field acting as a store. Writing to the array becomes a weak update on that store. (We thus do not distinguish between elements stored at different array indices.) This store field is notably apparent in Figure 4.11 as *store*. We do not introduce numerical domains to distinguish between array indices. We found this simplification to have limited impact on the precision of our analysis: developers tend to prefer Scala collections over arrays. When arrays are used, it is often as underlying storage to more complicated structures like heaps or hashtables for which simple numerical domains would anyway not be precise enough. In our analysis, we essentially represent arrays as instances of the *ArrayStub* class:

```
class ArrayStub[T](val length: Int) {  
  var store: T = _  
  def update(i: Int, v: T) = if (?) { store = v } else { }  
  def apply(i: Int) = store  
}
```

Exceptions. Accounting for exceptional flow is a challenging precision problem, as it typically clutters the control-flow graph with several edges to exception handlers. Scala has no checked exception and thus its compiler provides no information on which exceptions a method call might throw. We decided to ignore exceptional flow in this version of our analyzer.

Hence, the analysis is unsound in the presence of exceptions. Currently, throwing an exception in one program branch results in a bottom effect (indicating an impossible branch). This is consistent with the view that the results should be valid for non-exceptional executions.

Specialization for literal values. It is also worth noting that *Insane* specializes Integer and Boolean types, in order to distinguish between different literal values. Consequently, graphs can contain nodes representing *AnyInt* as well as *IntVal(n) $\forall n \in \text{Int}$* . This specialization is both at the node and type level, we thus have that

$$\text{type}(\text{BoolVal}(\text{true})) \sqcap \text{type}(\text{BoolVal}(\text{false})) = \perp$$

This is especially useful for boolean values, as it enables us to filter out branches protected by unsatisfiable boolean conditions.

4.3 Compositional Analysis of Higher-Order Code

The merge operation for effect-graphs presented in the previous section enables us to analyze programs without dynamic dispatch. Standard approaches to extend it to dynamic dispatch are either imprecise or lose modularity. In this section, we therefore extend the basic analysis to support dynamic dispatch (including higher-order functions and callbacks) in both a precise and rather modular way. The methodology by which we extend the core analysis to the higher-order case is independent of the particular domain of effect-graphs, so we present it in terms of a framework for precise interprocedural analysis of functions with callbacks.

Our framework works on top of any abstract interpretation-based analysis whose abstract domain R represents relations between program states. The abstract domain described in the previous section matches these requirements. Along with a set of control-flow graphs over statements \mathcal{P} previously described in Figure 4.1, we assume the existence of other usual components of such analyses: a concretization function $\gamma : R \rightarrow (S \times S)$ and a transfer function $T_f : (\mathcal{P} \times R) \rightarrow R$.

We now define a *composition operator* $\diamond : R \times R \rightarrow R$ for elements of the abstract domain, with the following property:

$$\forall e, f \in R . (\gamma(e) \circ \gamma(f)) \subseteq \gamma(e \diamond f)$$

that is

$$\forall s_0, s_1, s_2 . s_1 \in \gamma(e)(s_0) \wedge s_2 \in \gamma(f)(s_1) \implies s_2 \in \gamma(e \diamond f)(s_0)$$

In other words, \diamond must compose abstract relations in such a way that the result is a valid approximation of the corresponding composition in the concrete domain.

4.3.1 Control-Flow Graph Summarization

Summarization consists of replacing a part of the control-flow graph by a statement that over-approximates its effects. Concretely, we first augment the language with a special summary statement, characterized by a single abstract value:

$$\mathcal{P}_{ext} = \mathcal{P} \cup \{\text{Smr}(a \in R)\}$$

Consequently, we define $T_{f_{ext}}$ over \mathcal{P}_{ext} :

$$T_{f_{ext}}(s)(r) = \begin{cases} T_f(s)(r) & \text{if } s \in \mathcal{P} \\ r \diamond a & \text{if } s = \text{Smr}(a) \end{cases}$$

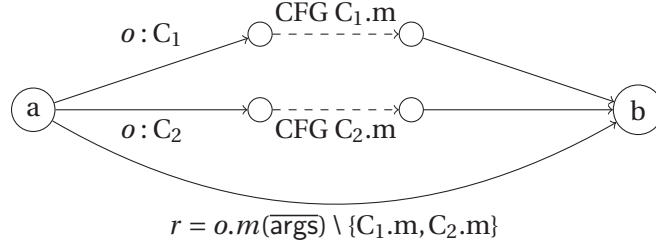
Let c be the control-flow graph of some procedure over \mathcal{P}_{ext} , and a and b two nodes of c such that a strictly dominates b and b post-dominates a . In such a situation, all paths from entry to b go through a and all paths from a to exit go through b . Let us consider the sub-graph between a and b , which we denote by $a \supset b$. This graph can be viewed as a control-flow graph with a as its source and b as its sink. The summarization consists of replacing $a \supset b$ by a single edge labelled with a summary statement obtained by analyzing the control-flow graph $a \supset b$ in isolation.

We observe that composition over the concrete domain is associative, whereas this is generally not the case for \diamond . Moreover, different orders of applications yield incomparable results. In fact, the order in which the summarizations are performed plays an important role in the overall result. When possible, left-associativity is preferred as it better encapsulates a forward top-down analysis and can exploit past information.

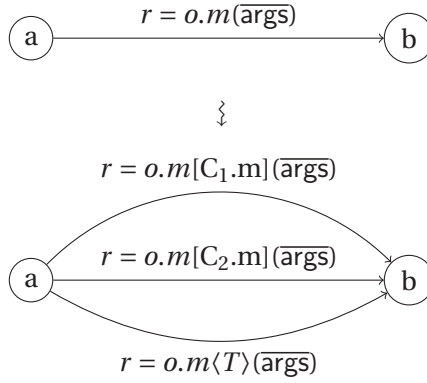
4.3.2 Partial Unfolding

The control-flow graph summarization presented above is one of the building blocks of our compositional framework. Another block is a mechanism for replacing method calls by summaries, or *unfolding*, which we present here.

When faced with a call statement $o.m(\overline{\text{args}})$, the analysis will extract information about o from the data-flow facts and compute the set of its potential static targets $T_{o.m} \subseteq \mathcal{M}$. The control-flow graphs corresponding to the targets are then included after a non-deterministic split. It is worth noting that the set of targets $T_{o.m}$ is generally not complete. Indeed, this process is performed during the fix-point computation, facts about o might still grow in the lattice during future iterations. The original call is therefore kept and annotated to exclude targets already unfolded, as pictured in Figure 4.5. In certain situations, we can conclude that all targets have been covered, rendering the alternative call edge infeasible and thus removable.


 Figure 4.5 – Example of unfolding with $T_{call} = \{C_1.m, C_2.m\}$.

To ensure the validity of the partial unfolding operation, we first apply a semantic-preserving rewrite: We split the call edge, thus making it explicit which fully determined (non-virtual) calls should be unfolded, and we record the set of unfolded targets in the original edge. This intuitively amounts to annotating call edges with the targets that have already been unfolded. For instance, for a set of targets $T = \{C_1.m, C_2.m\}$, we get



For an annotated call edge $r = o.m\langle T \rangle(\overline{args})$, we can compute its targets: $\text{targets}(o.m) \setminus T$. The fully determined call edges are displayed here for explanation purposes. In practice, we immediately replace them with their corresponding control-flow graphs, properly instantiated and surrounded with appropriate assign-statements for the arguments and return values:

If it is available, we can rely on abstract information on o to refine the number of potential targets. By keeping the call statement and by recording unfolded targets, we can perform this unfolded operation before we reach a fix-point. We argue this by making two observations:

- In the absence of precise information on the runtime type of o , we might have to unfold too many targets. Being an over-approximation, it remains sound.
- If during the abstract interpretation the facts at a increase in a way that include new potential targets for $o.m$, they will be included semantically in the alternative edge and become candidates for unfolding. Consequently, the partial unfolding does not prevent the discovery of other potential targets at a later stage.

```
sealed class A {                                     // .. continuing class A
  def m1() {
    val o = new A;
    this.m2(o)
  }

  def m2(o: A) {
    this.m3()
    o.f()
  }

  def m3() { }
  def f() { }
}

class B extends A {
  override def f() { .. }
}
```

Figure 4.6 – Example of a chain of method calls.

In certain situations, we can conclude that we have unfolded all potential targets. In such cases, the alternative call edge becomes infeasible and can be removed. This can happen either because the exact type is known for the receiver, some methods or classes are known to be final, or because we parametrize the analysis to assume a closed world.

4.3.3 Combining Unfolding and Summarization

We distinguish two main kinds of summaries. A summary that contains unanalyzed method calls is said to be *conditional*. In contrast, a *definite* summary is fully reduced down to a single edge with a summary statement.

We now illustrate the flexibility provided by our framework through a simple example displayed in Figure 4.6. In general, there are multiple ways to generate a definite summary from a control-flow graph, depending on the interleaving of summarization and unfolding operations.

For instance, one way to generate a summary for $A.m1$ would consist of the following steps: First, we fully summarize $A.m3$, $A.f$ and $B.f$, then we unfold their call in $A.m2$, summarize the result, unfold it in $A.m1$ and finally summarize it. This would represent a completely modular approach, where summaries are reused as much as possible. Being perhaps the most efficient way to compute a summary (as intermediate summaries for $A.m2$, $A.m3$, $A.f$ and $B.f$ are small, definite effects), it is also the least precise. Indeed, in this order, we have no precise information on o at the time of analyzing $o.f()$ and thus we have to consider every static targets— here $A.f$ and $B.f$, leading to an imprecise summary. We note that this approach, though generally used by traditional compositional analyses, falls short in the presence of callbacks where the number of static targets is typically large ($>1'000$ for the Scala library). In contrast, we could have waited to analyze $o.f()$ by generating a conditional summary for $A.m2$ where $this.m3()$ is unfolded but $o.f()$ remains unanalyzed. We refer to the decision of not analyzing a method call as *delaying*.

4.3.4 Controlled Delaying

We have seen through the examples above that choosing when to unfold a method call can have an important impact in terms of performance and precision. In our framework, we delegate this decision to a function $D(call, ctx)$. The precision and performance of the analysis are thus parametrized in D . We illustrate how this affects the analysis results by considering the two extremes:

- Fixing $D(...) = \text{false}$ ensures that every method is analyzed modularly, in a top-down fashion. This will enable summaries to be reused as much as possible. However, only relying on definite summaries hinders the precision of the overall analysis.
- In contrast, having $D(...) = \text{true}$ forces the analysis to delay every method call, hence leading to the analysis of a single, complete control-flow graph. This would in theory lead to the most precise result, but the construction of a single complete graph is not always possible. Recursive programs would, for instance, lead to an infinite control-flow graph. We specifically discuss the delaying of recursive functions in the following section. Even when available, the graphs obtained by delaying everything capture information that is mostly irrelevant for the overall effect computation. This lack of intermediate summarizations and simplifications results in a slow analysis.

We also note that the analysis must be able to conservatively reason about delayed method calls in order to proceed past them. A conservative approach is to assume that, by resetting the facts to the identity relation, we do not know anything anymore after the delayed call.

4.3.5 Handling Recursion

Assuming the underlying abstract interpretation-based analysis does terminate (which we ensure for effect-graphs), we still need to ensure that the control-flow graph does not keep changing due to unfoldings. For this reason, we need to take special measures for cycles in the call-graph.

Statically detecting recursion is non-trivial, especially in the presence of callbacks. An attempt to use a refined version of a standard class analysis proved to be overly imprecise: it flags every higher-order functions as recursive. Therefore, *Insane* lazily discovers recursive methods during the analysis when closing a loop in the progressively constructed call-graph. It then rewinds the analysis until the beginning of the loop in the lasso-shaped call-graph in order to handle the cycle safely. We handle recursion by ensuring that only definite summaries are generated for methods within the cycle. In fact, we enforce termination by requiring that $D(c, ctx)$ returns *false* for any call c within the call-graph cycle.

It is worth noting that $D(...)$ is constrained only for calls within the call-graph cycle: we are free to decide to delay the analysis when the call is at the boundaries of a cycle. For example, it

is critical for precision purposes to delay as much as possible the analysis of the entire cycle. When analyzing a set of mutually recursive functions, we start by assuming that all have a definite summary of identity, thus indicating no effect. The process then uses a standard fix-point iterative process and builds up summaries until convergence.

4.3.6 Instantiation for Effect Graphs

We now discuss the instantiation of this framework in the context of effect graphs presented in Section 4.2. We can quickly identify that our abstract domain is relational and thus a candidate for use in this framework. The original statements are thus extended with a summary statement characterized by an effect-graph:

$$\mathcal{P}_{ext} := \mathcal{P} \cup \{\text{Smr}(G)\}$$

We can also notice that the graph-merging operation acts as composition operator \diamond :

$$G_1 \diamond G_2 := \text{merge } G_2 \text{ in } G_1$$

For the delaying decision function D , we base our decision on a combination of multiple factors. One important factor is of course the number of targets a method currently has. We also check whether the receiver escapes the current function, which indicates that delaying might improve its precision. As expected, experiments indicate that this decision function dictates the trade-off between the performance and precision of the overall analysis.

In case the call at hand is recursive, we conservatively prevent its delaying. However, we also check whether the number of targets is not too high. In practice, we consider this upper limit to be 50. We argue that, without the ability to delay, the effects would become overly imprecise anyway if we exceed this many targets for a single call. In such cases, the analysis gives up and assigns \top as definite summary to all concerned functions.

4.3.7 Context Sensitivity

Compositional summaries already give us a powerful form of context sensitivity but it is not always sufficient in practice, specifically in the presence of recursive methods relying on callbacks. We thus had to introduce another form of context-sensitivity that specializes the analysis of the same method for multiple call signatures. We first introduce a notion of type signature that is recursively defined by:

$$\begin{aligned} \text{TypeSig} &:= \text{TypeInfo} \times 2^{\text{Fields} \times \text{TypeSig}} \\ \text{TypeInfo} &:= \text{Type} \times \text{Boolean} \end{aligned}$$

Type signatures represent type information about a particular object. They not only give information about the type of the object itself, but can also provide a type signature for some of its fields. Each type information is annotated with a flag that indicates whether subtypes should also be considered in order to distinguish $_ \sqsubseteq T$ and $_ = T$. We define the *depth* of a type signature to be the maximum depth until no field information is specified:

$$\text{depth}(s) := 1 + \max \{0\} \cup \{\text{depth}(s_f) \mid (f, s_f) \in s.\text{fields}\}$$

We also introduce $\text{TypeSig}_d \subset \text{TypeSig}$ to represent signatures of maximum depth d . The call context used to identify summaries is established by combining the type-estimates for the receiver, as well as each argument.

4.4 Producing Readable Effect Summaries

We have demonstrated that summaries based on control-flow graphs are a flexible and expressive representation of heap modifications. However, such graph-based summaries are often not directly usable as feedback to programmers, for several reasons. First, they capture both read and write effects, whereas users are likely to be interested primarily in write effects. Next, they can refer to internal memory cells that are allocated within a method and do not participate directly in an effect. Last but not least, they are not in textual form and can be difficult to interpret by developers who are used to textual representations.

To improve the usefulness of the analysis for program understanding purposes, we describe effect summaries of methods in a more concise and textual form. For this purpose, we adopt regular expressions because they are a common representation for the infinite sets of strings and can, therefore, characterize access paths [Deu92]. They also have a notable tradition of use for representing heap effects [LH88]. To generate an approximate textual representation of graph-based summaries for our analysis, we adopt the general idea of representing graphs by using sets of paths.

We first show how we construct a regular expression for a *definite* summary. For definite summaries, a graph-based effect is available for summarizing the method. The graph not only describes which fields can be modified, but also to which value they can be assigned. Whereas the corresponding regular expression only describes which fields could be written to. The task is therefore reduced to generating a conservative set of paths to fields that might be modified. We construct the following non-deterministic finite-state automaton $(Q, \Sigma, \delta, q_0, \{q_f\})$ based on a graph effect G :

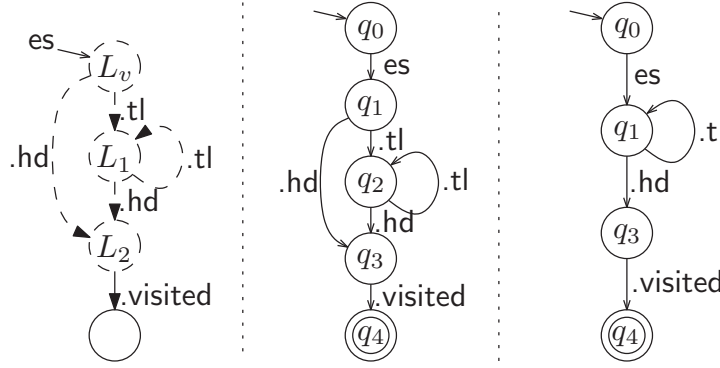


Figure 4.7 – Transformation steps from an effect-graph to a minimized DFA. The graph on the left is the *definite* effect of an impure list traversal. The center graph is the corresponding NFA whose accepting language represents paths to modified fields. The last graph is the minimized DFA to be translated to a regular expression.

$$\begin{aligned}
 Q &:= G.V \cup \{q_f, q_0\} \\
 \Sigma &:= \{f \mid v_1 \xrightarrow{f} v_2 \in G.E\} \\
 \delta &:= G.E \cup \{q_0 \xrightarrow{n} n \mid n \in G.V \wedge \text{connecting}(n)\} \\
 &\quad \cup \{v_1 \xrightarrow{f} q_f \mid v_1 \xrightarrow{f} v_2 \in G.IE \wedge v_1 \text{ is not an inside node}\}
 \end{aligned}$$

The automaton accepts strings of words where “letters” are names of the method arguments and field accesses. Given an access path, $o.f_1.f_2 \cdots f_{n-1}.f_n$, the automaton accepts it if f_n is modified on the set of objects reached via $o.f_1.f_2 \cdots f_{n-1}$. We exclude writes on inside nodes, because they represent writes that are not be observable from outside, as the node represents objects allocated within the function. Using the non-deterministic automaton, we produce a regular expression by first determinizing it, then minimizing the obtained deterministic automaton, and finally applying a standard transformation into a regular expression by successive state-elimination. Figure 4.7 shows the effect-graph and the corresponding automata (non-minimized and minimized) for the example from the end of Section 4.1. In general, we found the passage through determinization and minimization to have a significant positive impact on the conciseness of the final expression.

For a *conditional* summary, we extract the set of unanalyzed method calls, assuming that they are all pure, then compute a (definite) effect and present the corresponding regular expression along with the set of calls. The natural interpretation is that the regular expression captures all possible writes, under the assumption that no function in the set has a side-effect.

Section 4.5.3 and in particular Figure 4.11 below show some of the regular expressions that

were built from our analysis of collections in the standard Scala library.

4.5 Evaluation on Scala Library

We implemented the analysis described in the previous sections as part of a tool called *Insane*. *Insane* is a plugin for the official Scala compiler.

4.5.1 Overall Results

To evaluate the precision of our analysis, we ran it on the entire Scala library, composed of approximately 58000 methods. We believe this is a relevant benchmark: due to the functional paradigm encouraged in Scala, several methods are of higher-order nature. For instance, collection classes typically define traversal methods that take functions as arguments, such as *filter*, *fold*, *exists*, or *foreach* ([OM09]). It is worth noting that we assumed a closed-world in order to analyze the library. Indeed, as most classes of the library are fully extensible, analyzing it without this assumption would not yield interesting results. Given that even getters and setters can be extended, most of the effects would depend on future extensions, resulting in almost no definite summaries.

We proceeded as follows: For each method, we analyzed it using its declaration context and classified the resulting summary as a member of one of four categories; if the summary is definite, we look for observable effects. Depending on the presence of observable effects, the method is flagged either as *pure* or *impure*. If the summary is conditional, we check if the effect would be pure under the assumption that every remaining (delayed) method call is pure. In such cases, the effect is said to be *conditionally pure*. Otherwise, the effect is said to be *impure*. Lastly, an effect can be *top* if either the analysis timed out, or if more than 50 targets are unfolded in a situation where delaying is not available (e.g, recursive methods). We used a timeout of two minutes per function. We note that although these parameters are to some extent arbitrary, we estimate that they correspond to reasonable expectations for the analysis to be useful. The different categories of effects form a lattice:

$$\text{pure} \sqsubseteq \text{conditionally pure} \sqsubseteq \text{impure} \sqsubseteq \top$$

Figure 4.8 displays the number of summaries per category and per package. Observe that most methods are either *pure* or *conditionally pure*, which is what we would expect in a library that encourages functional programming.

Overall, the entire library takes short of twenty hours to be fully processed. This is mostly due to the fact that, in this scenario, we compute a summary for each method. Due to its modularity though, this analysis could be used in an incremental fashion, reanalyzing only modified code and new dependencies reusing past, unchanged results. Depending on the level of context sensitivity, past results can be efficiently reused in an incremental fashion and

Chapter 4. Effect Analysis for Programs with Callbacks

Package	Methods	Pure	Cond. Pure	Impure	T
scala	5721	79%	11%	10%	1%
scala.annotation	41	93%	2%	2%	2%
scala.beans	25	64%	8%	28%	0%
scala.collection	5182	52%	28%	17%	4%
scala.collection.concurrent	608	40%	19%	37%	4%
scala.collection.convert	1106	62%	23%	13%	1%
scala.collection.generic	649	61%	22%	12%	5%
scala.collection.immutable	6027	58%	13%	23%	6%
scala.collection.mutable	7263	48%	18%	29%	5%
scala.collection.parallel	13842	36%	13%	37%	14%
scala.collection.script	132	86%	1%	13%	0%
scala.compat	9	22%	33%	44%	0%
scala.io	546	47%	11%	40%	2%
scala.math	1847	67%	28%	5%	0%
scala.parallel	39	77%	23%	0%	0%
scala.ref	113	58%	3%	39%	0%
scala.reflect	5862	50%	9%	40%	1%
scala.runtime	1620	61%	25%	14%	1%
scala.sys	767	44%	22%	30%	4%
scala.testing	44	52%	2%	43%	2%
scala.text	115	87%	0%	11%	2%
scala.util	1786	51%	11%	32%	6%
scala.util.parsing	2206	56%	12%	27%	5%
scala.xml	2860	56%	11%	30%	3%
Total:	58410	52%	15%	27%	6%

Figure 4.8 – Decomposition of resulting summaries per package.

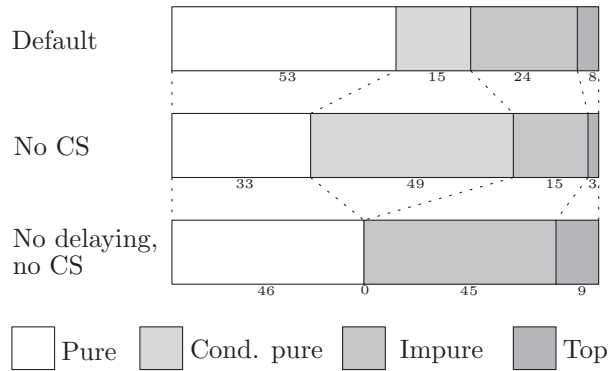


Figure 4.9 – Comparing strategies. Numbers below boxes are percentages. Running times are 166, 123 and 57 minutes respectively.

thus enables the analysis to scale well to large applications.

4.5.2 Comparative Analysis

To demonstrate the importance of some of the key features of *Insane*, we analyzed parts of the Scala library in three different settings: (1) the default configuration and strategy that ship with *Insane*, (2) disabled context-sensitivity, and (3) delaying and context-sensitivity both disabled. We analyzed a representative subset of the Scala library, specifically all mutable and immutable collections, under the three analysis configurations.

Figure 4.9 displays the decomposition of the effects for each of the analysis settings. First of all, we can see that the default configuration used by *Insane* is more precise than the alternatives. Indeed, it produces the largest number of purity guarantees. This precision, however, has a cost; it is also the slowest configuration. In the second configuration, *Insane* without context sensitivity, we first notice that although it produces a smaller number of definite-pure guarantees, it is also able to infer many conditionally pure methods. However, the quality of those conditional summaries is likely to be worse than for the first setting, resulting in assumptions that probably do not hold in practice. In comparison with the default *Insane*, we see that some of the conditionally pure are in fact most likely impure, but that additional precision is required to figure this out. The results for the third configuration need to be interpreted with care. In this setting, delaying is not permitted. As a result, most methods relying on higher-order functions are misclassified as impure. In fact, when running the analysis in this configuration, we see that more than 50% of the methods are considered to be impure.

4.5.3 Selected Examples

To demonstrate the precision of the analysis, we take a closer look at several methods that rely on the library. We targeted five collections: two immutable ones (*TreeSet* and *List*) and three

```
class Elem(val i: Int) {  
  var visited = false  
}  
  
def genTrav(es: TreeSet[Elem],  
           f: Elem ⇒ Unit) = {  
  es.foreach(f)  
}  
  
def grow(es: TreeSet[Elem], e: Elem) = {  
  es + e  
}  
  
def pureTrav(es: TreeSet[Elem]) = {  
  es.foreach {  
    e ⇒ ()  
  }  
}  
  
def impureTrav(es: TreeSet[Elem]) = {  
  es.foreach {  
    e ⇒ e.visited = true  
  }  
}
```

Figure 4.10 – The particular four operations applied on the `TreeSet` collection

mutable ones (`HashSet`, `LinkedList`, and `ArrayBuffer`). For each of these collections, we analyze code that performs four operations:

1. Generic Traversal: call `foreach` with an arbitrary closure,
2. Pure Traversal: call `foreach` with a pure closure,
3. Impure Traversal: call `foreach` with a closure modifying the collection elements,
4. Growing: build a larger collection, by copying and extending it for immutable ones, or modifying it in place for mutable ones. The method used for growing depends on what is available in the public interface of the collection, e.g, `add`, `append` or `prepend`.

Figure 4.10 shows functions corresponding to these four operations when applied to the `TreeSet` collection, and summarizes the general classes of operations.

The resulting effects are converted into a readable format, as described in Section 4.4 and displayed in Figure 4.11. We note that, in each case, producing these regular expressions takes under five seconds. First of all, we can see that all pure traversals are indeed proved pure and have no effect on the internal representation of the collections. Also, by assuming the closure passed is pure, we are often able to report that a generic traversal has no effect on the collection. The exceptions are the generic traversals of `TreeSet` and `ArrayBuffer`. In these two cases, the computed effect is \top , due to the fact that their respective traversal routines are implemented using a recursive function with a highly dynamic dispatch within its body. However, due to context sensitivity, We can see that we are able to obtain precise results when the closure is determined. For impure traversal of `TreeSet`, the analysis has to generate and combine no less than 27 method summaries. The fact that the resulting effect remains precise, despite the fundamental complexity of the library shows, that the analysis serves its purpose of combining precision and modularity through summaries, even in the case of higher-order programs.

<code>immutable.TreeSet:</code>	Generic trav.	Any
	Pure trav.	Pure
	Impure trav.	<code>es.tree(.right .left)*.key.visited</code>
	Grow	Pure
<code>immutable.List:</code>	Generic trav.	Pure (conditionally on the closure)
	Pure trav.	Pure
	Impure trav.	<code>es.tl*.hd.visited</code>
	Grow	Pure
<code>mutable.HashSet:</code>	Generic trav.	Pure (conditionally on the closure)
	Pure trav.	Pure
	Impure trav.	<code>es.table.store.visited</code>
	Grow	<code>es.tableSize es.table.store es.sizemap.store es.sizemap es.table</code>
<code>mutable.LinkedList:</code>	Generic trav.	Pure (conditionally on the closure)
	Pure trav.	Pure
	Impure trav.	<code>es.next*.elem.visited</code>
	Grow	<code>es.next.next*</code>
<code>mutable.ArrayBuffer:</code>	Generic trav.	Any
	Pure trav.	Pure
	Impure trav.	<code>es.array.store.visited</code>
	Grow	<code>es.size0 es.array.store es.array</code>

Figure 4.11 – Readable effect descriptions obtained from graph summaries from four operations performed on five kinds of collections.

In the cases of impure traversals, the effects correctly report that all elements of the collections can be modified. Additionally, they uncover the underlying implementation structures. For example, we can see that the `HashSet` class is implemented by using a flat-hash table (using open addressing) instead of the usual array of chained buckets. It is worth noting that `TreeSet` is implemented using red-black trees. For mutable collections, growing the collection indeed has an effect on the underlying implementation. Growing immutable collections remains pure because the modifications are applied only to the returned copy.

Overall, we believe such summaries are extremely useful, as they qualify the impurity. In almost all cases, the programmer can rely on the result produced by *Insane* to conclude that the functions have the intended effects.

4.6 Related Work

Our goals stand at the intersection of two long-standing fundamental problems:

1. effect and alias analysis for mutable linked structures; [CK88, CWZ90, JG91, MRR02, TD03, Rou04];

2. control-flow analysis [Shi88] for higher-order functional programs.

Because we considered the heap analysis to be the first-order challenge, we focused on adapting the ideas from the first category to higher-order settings.

The analysis domain presented in this chapter builds on the work of [SR05, Sal06] who used graphs to encode method-effect summaries, independently from aliasing relations. The elements of this abstract domain are best understood as state transformers, rather than sets of heaps. This observation, which is key to the applicability of the generic relational framework described in Section 4.3, is also made by Madhavan, Ramalingam, and Vaswani [MRV11] who formalize their analysis and applied it to C# code. The same authors extend their analysis to provide special support for higher-order procedures [MRV12]. In contrast with our work, [MRV12] summarizes higher-order functions by using only CFGs or a particular, fixed, normal form: a loop around the un-analyzed invocations. Because our analysis supports arbitrary conditional summaries, it is a strict generalization in terms of precision of summaries. Another distinctive feature of our analysis is its support for strong updates: it is crucial for obtaining a good approximation of many patterns commonly found in Scala code. In fact, the reduction of CFGs to a normal form in [MRV12] relies on graph transformers being monotonic, a property that is incompatible with strong updates. Finally, our tool also produces regular expression summaries, delivering results that can be immediately useful to programmers.

The idea of delaying parts of the analysis has been explored before in interprocedural analyses to improve context-sensitivity [CC02, YYC08] or to speed up bottom-up whole-program analyses [JMT10]. Our work shows that this approach also brings benefits to the analysis of programs with callbacks, and is in fact critical to its applicability.

Our analysis masks only effects that can be proved to be performed on fresh objects in given procedure call contexts. A more ambitious idea is to mask effects across method calls of an abstract data types, which resulted in a spectrum of techniques with different flexibility and annotation burden [JHS86, dRE98, CN02, CD02, BLS03, FL03, BDF⁺04, BN05]. Our analysis differentiates in that it is fully automated; but we do hope to benefit in the future from user hints that express encapsulation, information hiding, or representation independence.

Separation logic [DYDG⁺10, BCI11] and implicit dynamic frames [SJP09, PS11] are two popular paradigms for controlling modifications to heap regions. Nordio et al. describe an adaptation of dynamic frames [NCM⁺10] for the automated verification of programs with higher-order functions. We note that effect analysis is a separate analysis, whereas separation logic analyses need to perform shape and effect analyses at the same time. This coupling of shape and effect, through the notion of footprint, makes it harder to deploy separation logic-based analyses as lightweight components that are separate from subsequent analysis phases. Moreover, the state of the art in separation logic analyses is such that primarily linked list structures can be analyzed in a scalable way, whereas our analysis handles general graphs and is less sensitive to aliasing relationships.

The importance of conditional effects expressed as a function of arguments is identified in an effect system [ROH12] for Scala, which requires some type annotations and is higher-level, but provides more control over encapsulation and elegantly balances the expressive power with the simplicity of annotations. The resulting system is fully modular and supports, e.g, separate compilation.

4.7 Conclusion

Knowing the effects of program procedures is a fundamental activity for any reasoning task involving imperative code. We have presented an algorithm, a tool, and experiments showing that this task is feasible for programs written in Scala, a modern functional and object-oriented language. Our solution involves a general framework for relational effect analyses designed to support different automated reasoning strategies and to enable analysis designers to experiment with trade-offs between precision and time. Building on this framework we have introduced an abstract domain designed to track read and write effects on the heap. Combining the framework with the abstract domain, we have obtained an effect analysis for Scala. We have implemented and evaluated the analysis on the entire Scala standard library, producing a detailed breakdown of its 58000 functions by purity status. Finally, we have developed and implemented a technique to produce human-readable summaries of the effects in order to make them immediately useful to programmers. We have shown that these summaries can concisely and naturally describe heap regions, thus producing feedback that conveys much more information than a simple pure/impure dichotomy. *Insane* works on unannotated code and can thus readily be applied to existing code bases, facilitating program understanding, as well as subsequent deeper analyses and verification tasks.

Conclusions

In this thesis, we have investigated various ways of deploying advanced software-development techniques in a way that can be useful to a broad audience. We have presented in Chapter 2 a new framework for program synthesis; it combines transformational and counter-example-guided approaches. Our algorithm is able to synthesize interesting recursive functions that manipulates unbounded values. The search for solutions, however, is exponential and its scalability is thus limited. Therefore, the problem of synthesizing general-purpose software from scratch remains a difficult challenge.

We did however identify one of the benefits of such a framework for synthesis: To target particular problems or domain-specific formulas, it can be extended with additional deductive rules. In Chapter 3, we have implemented new rules to automatically repair invalid programs by first locating the error and then synthesizing correct alternatives. We were able to repair intricate errors within recursive functions manipulating unbounded structures. We believe that software repair is a very useful instantiation of synthesis, which scales beyond what is achievable by synthesis alone, due to its error localization.

In Chapter 4, we have presented ways to create hybrid programs, mixing a purely functional part with external functions implemented with arbitrary Scala code. We have described an analysis of side-effect that is able to characterize functions, depending on their effects. As a result, we identify functions that remain referentially transparent, enabling their use within Leon programs even though their body remains intractable for Leon.

Furthermore, we believe that, by targeting programs written mostly in PureScala, we are able to propose interesting techniques and algorithms that help developers produce trustworthy software. Because our techniques are applied during the development process, we made them available within an integrated development environment, deployed as a public web interface.

A Synthesis Benchmarks and Solutions

For each synthesis benchmark, we first give a *header* file that contains all the necessary definitions, including the data-structures and auxiliary abstraction functions. Then, we provide for each benchmark the problem that we give as input to Leon and the solution that Leon generates automatically. Benchmarks are listed in order, and we assume that all solutions to previous benchmarks are available when synthesizing the next one.

A.1 Compiler

A.1.1 Header

```
import leon.lang._
import leon.lang.synthesis._
import leon._

object Compiler {
  abstract class Expr
  case class Plus(lhs: Expr, rhs: Expr) extends Expr
  case class Minus(lhs: Expr, rhs: Expr) extends Expr
  case class UMinus(e: Expr) extends Expr
  case class LessThan(lhs: Expr, rhs: Expr) extends Expr
  case class And(lhs: Expr, rhs: Expr) extends Expr
  case class Implies(lhs: Expr, rhs: Expr) extends Expr
  case class Or(lhs: Expr, rhs: Expr) extends Expr
  case class Not(e: Expr) extends Expr
  case class Eq(lhs: Expr, rhs: Expr) extends Expr
  case class Ite(cond: Expr, thn: Expr, els: Expr) extends Expr
  case class BoolLiteral(b: Boolean) extends Expr
  case class IntLiteral(i: BigInt) extends Expr

  abstract class Value
  case class BoolValue(b: Boolean) extends Value
  case class IntValue(i: BigInt) extends Value
}
```

case object Error **extends** Value

```
def eval(e: Expr): Value = e match {  
  case Plus(l, r) ⇒  
    (eval(l), eval(r)) match {  
      case (IntValue(il), IntValue(ir)) ⇒ IntValue(il+ir)  
      case _ ⇒ Error  
    }  
  
  case Minus(l, r) ⇒  
    (eval(l), eval(r)) match {  
      case (IntValue(il), IntValue(ir)) ⇒ IntValue(il-ir)  
      case _ ⇒ Error  
    }  
  
  case UMinus(l) ⇒  
    eval(l) match {  
      case IntValue(b) ⇒ IntValue(-b)  
      case _ ⇒ Error  
    }  
  
  case LessThan(l, r) ⇒  
    (eval(l), eval(r)) match {  
      case (IntValue(il), IntValue(ir)) ⇒ BoolValue(il < ir)  
      case _ ⇒ Error  
    }  
  
  case And(l, r) ⇒  
    eval(l) match {  
      case b @ BoolValue(false) ⇒ b  
      case b: BoolValue ⇒  
        eval(r)  
      case _ ⇒  
        Error  
    }  
  
  case Or(l, r) ⇒  
    eval(l) match {  
      case b @ BoolValue(true) ⇒  
        b  
      case b: BoolValue ⇒  
        eval(r)  
      case _ ⇒  
        Error  
    }  
  
  case Implies(l, r) ⇒
```

```

    eval(l) match {
      case b @ BoolValue(true) ⇒
        eval(r)
      case b @ BoolValue(false) ⇒
        BoolValue(true)
      case _ ⇒ Error
    }

case Not(l) ⇒
  eval(l) match {
    case BoolValue(b) ⇒ BoolValue(!b)
    case _ ⇒ Error
  }

case Eq(l, r) ⇒
  (eval(l), eval(r)) match {
    case (IntValue(il), IntValue(ir)) ⇒ BoolValue(il == ir)
    case (BoolValue(il), BoolValue(ir)) ⇒ BoolValue(il == ir)
    case _ ⇒ Error
  }

case lte(c, t, e) ⇒
  eval(c) match {
    case BoolValue(true) ⇒ eval(t)
    case BoolValue(false) ⇒ eval(t)
    case _ ⇒ Error
  }

case IntLiteral(l) ⇒ IntValue(l)
case BoolLiteral(b) ⇒ BoolValue(b)
  }
}

```

A.1.2 Rewrite Implies

Problem

```

def rewritelmpies(in: Implies): Expr = {
  choose{ (out: Expr) ⇒
    eval(in) == eval(out) && !(out.isInstanceOf[Implies])
  }
}

```

Solution

```

def rewritelmpies(in : Implies): Expr = {
  require(in.isInstanceOf[Implies])
  Or(Not(in.lhs), in.rhs)
}

```

```
} ensuring {  
  (out : Expr) ⇒ eval(in) == eval(out) && !out.isInstanceOf[Implies]  
}
```

A.1.3 Rewrite Minus

Problem

```
def rewriteMinus(in: Minus): Expr = {  
  choose{ (out: Expr) ⇒  
    eval(in) == eval(out) && !(out.isInstanceOf[Minus])  
  }  
}
```

Solution

```
def rewriteMinus(in : Minus): Expr = {  
  require(in.isInstanceOf[Minus])  
  Plus(in.lhs, UMinus(in.rhs))  
} ensuring {  
  (out : Expr) ⇒ eval(in) == eval(out) && !out.isInstanceOf[Minus]  
}
```

A.2 List

A.2.1 Header

```
import leon.annotation._  
import leon.lang._  
import leon.lang.synthesis._  
  
object List {  
  sealed abstract class List  
  case class Cons(head: BigInt, tail: List) extends List  
  case object Nil extends List  
  
  def size(l: List) : BigInt = (l match {  
    case Nil ⇒ BigInt(0)  
    case Cons(_, t) ⇒ BigInt(1) + size(t)  
  }) ensuring(res ⇒ res ≥ 0)  
  
  def content(l: List): Set[BIGInt] = l match {  
    case Nil ⇒ Set.empty[BIGInt]  
    case Cons(i, t) ⇒ Set(i) ++ content(t)  
  }  
  
  def abs(i : BIGInt) : BIGInt = {  
    if(i < 0) -i else i  
  }
```

```

    } ensuring(_ ≥ 0)

    def dispatch(es: (BigInt, BigInt), rest: (List, List)): (List, List) = {
      (Cons(es._1, rest._1), Cons(es._2, rest._2))
    }
  }
}

```

A.2.2 Insert

Problem

```

def insert(in: List, v: BigInt) = choose {
  (out: List) ⇒
    content(out) == content(in) ++ Set(v)
}

```

Solution

```

def insert(in : List, v : BigInt): List = {
  Cons(v, in)
} ensuring {
  (out : List) ⇒ content(out) == content(in) ++ Set[BigInt](v)
}

```

A.2.3 Delete

Problem

```

def delete(in: List, v: BigInt) = choose {
  (out: List) ⇒
    content(out) == content(in) -- Set(v)
}

```

Solution

```

def delete(in : List, v : BigInt): List = {
  in match {
    case Cons(head, tail) ⇒
      if (head == v) {
        delete(tail, head)
      } else {
        Cons(head, delete(tail, v))
      }
    case Nil ⇒
      Nil
  }
} ensuring {
  (out : List) ⇒ content(out) == content(in) -- Set[BigInt](v)
}

```

A.2.4 Merge

Problem

```
def merge(in1: List, in2: List) = choose {  
  (out: List) ⇒  
    content(out) == content(in1) ++ content(in2)  
}
```

Solution

```
def merge(in1 : List, in2 : List): List = {  
  in1 match {  
    case Cons(head, tail) ⇒  
      Cons(head, merge(tail, in2))  
    case Nil ⇒  
      in2  
  }  
} ensuring {  
  (out : List) ⇒ content(out) == content(in1) ++ content(in2)  
}
```

A.2.5 Diff

Problem

```
def diff(in1: List, in2: List) = choose {  
  (out: List) ⇒  
    content(out) == content(in1) -- content(in2)  
}
```

Solution

```
def diff(in1 : List, in2 : List): List = {  
  in2 match {  
    case Cons(head, tail) ⇒  
      delete(diff(in1, tail), head)  
    case Nil ⇒  
      in1  
  }  
} ensuring {  
  (out : List) ⇒ content(out) == content(in1) -- content(in2)  
}
```

A.2.6 split

Problem

```
def split(in: List) : (List, List) = choose {
```

```

(out: (List, List)) ⇒
  val s1 = size(out._1)
  val s2 = size(out._2)

  abs(s1 - s2) ≤ 1 && s1 + s2 == size(in) &&
  content(out._1) ++ content(out._2) == content(in)
}

```

Solution

```

def split(in : List): (List, List) = {
  in match {
    case Cons(head, tail) ⇒
      tail match {
        case Cons(head1, tail1) ⇒
          dispatch((head, head1), split(tail1))
        case Nil ⇒
          (Nil, Cons(head, Nil))
      }
    case Nil ⇒
      (Nil, Nil)
  }
} ensuring {
  (out: (List, List)) ⇒
    val s1 = size(out._1)
    val s2 = size(out._2)

    abs(s1 - s2) ≤ 1 && s1 + s2 == size(in) &&
    content(out._1) ++ content(out._2) == content(in)
}

```

A.3 Sorted List

A.3.1 Header

```

import leon.annotation._
import leon.lang._
import leon.lang.synthesis._

object SortedListInsert {
  sealed abstract class List
  case class Cons(head: BigInt, tail: List) extends List
  case object Nil extends List

  def size(l: List): BigInt = (l match {
    case Nil ⇒ BigInt(0)
    case Cons(_, t) ⇒ BigInt(1) + size(t)
  }) ensuring(res ⇒ res ≥ 0)
}

```

```
def content(l: List): Set[BigInt] = l match {  
  case Nil => Set.empty[BigInt]  
  case Cons(i, t) => Set(i) ++ content(t)  
}  
  
def isSorted(list: List): Boolean = list match {  
  case Nil => true  
  case Cons(_, Nil) => true  
  case Cons(x1, Cons(x2, _)) if (x1 > x2) => false  
  case Cons(_, xs) => isSorted(xs)  
}  
}
```

A.3.2 Insert

Problem

```
def insert(in: List, v: BigInt): List = {  
  require(isSorted(in))  
  
  choose { (out : List) =>  
    (content(out) == content(in) ++ Set(v)) && isSorted(out)  
  }  
}
```

Solution

```
def insert(in : List, v : BigInt): List = {  
  require(isSorted(in))  
  in match {  
    case Cons(head, tail) =>  
      if (head < v) {  
        Cons(head, insert(tail, v))  
      } else if (head == v) {  
        insert(tail, head)  
      } else {  
        Cons(v, insert(tail, head))  
      }  
    case Nil =>  
      Cons(v, Nil)  
  }  
} ensuring {  
  (out : List) => content(out) == content(in) ++ Set[BigInt](v) && isSorted(out)  
}
```


A.3.3 Insert Always

Problem

```
def insertAlways(in: List, v: BigInt) = {
  require(isSorted(in))

  choose{ (out : List) =>
    (content(out) == content(in) ++ Set(v)) && isSorted(out) && size(out) == size(in) + 1
  }
}
```

Solution

```
def insertAlways(in : List, v : BigInt): List = {
  require(isSorted(in))
  in match {
    case Cons(head, tail) =>
      if (head < v) {
        Cons(head, insertAlways(tail, v))
      } else if (head == v) {
        Cons(head, insertAlways(tail, head))
      } else {
        Cons(v, insertAlways(tail, head))
      }
    case Nil =>
      Cons(v, Nil)
  }
} ensuring {
  (out : List) => content(out) == content(in) ++ Set[BigInt](v) && isSorted(out) && size(out) ==
    size(in) + BigInt(1)
}
```

A.3.4 Delete

Problem

```
def delete(in: List, v: BigInt) = {
  require(isSorted(in))

  choose( (res : List) =>
    (content(res) == content(in) -- Set(v)) && isSorted(res)
  )
}
```

Solution

```
def delete(in : List, v : BigInt): List = {
  require(isSorted(in))
```

```
in match {
  case Cons(head, tail) =>
    if (head == v) {
      delete(tail, head)
    } else {
      Cons(head, delete(tail, v))
    }
  case Nil =>
    Nil
}
} ensuring {
  (res : List) => content(res) == content(in) -- Set[BigInt](v) && isSorted(res)
}
```

A.3.5 Merge

Problem

```
def merge(in1: List, in2: List) = {
  require(isSorted(in1) && isSorted(in2))
  choose {
    (out : List) =>
      (content(out) == content(in1) ++ content(in2)) && isSorted(out)
  }
}
```

Solution

```
def merge(in1 : List, in2 : List): List = {
  require(isSorted(in1) && isSorted(in2))
  in1 match {
    case Cons(head, tail) =>
      insert(merge(tail, in2), head)
    case Nil =>
      in2
  }
} ensuring {
  (out : List) => content(out) == content(in1) ++ content(in2) && isSorted(out)
}
```

A.3.6 Diff

Problem

```
def diff(in1: List, in2: List) = {
  require(isSorted(in1) && isSorted(in2))

  choose {
```

```

    (out : List) =>
      (content(out) == content(in1) -- content(in2)) && isSorted(out)
  }
}

```

Solution

```

def diff(in1 : List, in2 : List): List = {
  require(isSorted(in1) && isSorted(in2))
  in2 match {
    case Cons(head, tail) =>
      diff(delete(in1, head), tail)
    case Nil =>
      in1
  }
} ensuring {
  (out : List) => content(out) == content(in1) -- content(in2) && isSorted(out)
}

```

A.3.7 Insertion Sort

Problem

```

def insertionSort(in: List): List = {
  choose { (out: List) =>
    content(out) == content(in) && isSorted(out)
  }
}

```

Solution

```

def insertionSort(in : List): List = {
  in match {
    case Cons(head, tail) =>
      insert(insertionSort(tail), head)
    case Nil =>
      Nil
  }
} ensuring {
  (out : List) => content(out) == content(in) && isSorted(out)
}

```

A.4 Strictly Sorted Lists

A.4.1 Header

```

import leon.annotation._
import leon.lang._

```

Appendix A. Synthesis Benchmarks and Solutions

```
import leon.lang.synthesis._

object StrictSortedList {
  sealed abstract class List
  case class Cons(head: BigInt, tail: List) extends List
  case object Nil extends List

  def size(l: List): BigInt = (l match {
    case Nil => BigInt(0)
    case Cons(_, t) => BigInt(1) + size(t)
  }) ensuring(res => res ≥ 0)

  def content(l: List): Set[BIGInt] = l match {
    case Nil => Set.empty[BIGInt]
    case Cons(i, t) => Set(i) ++ content(t)
  }

  def isSorted(list: List): Boolean = list match {
    case Nil => true
    case Cons(_, Nil) => true
    case Cons(x1, Cons(x2, _)) if (x1 ≥ x2) => false
    case Cons(_, xs) => isSorted(xs)
  }
}
```

A.4.2 Insert

Problem

```
def insert(in: List, v: BigInt): List = {
  require(isSorted(in))

  choose { (out : List) =>
    (content(out) == content(in) ++ Set(v)) && isSorted(out)
  }
}
```

Solution

```
def insert(in : List, v : BigInt): List = {
  require(isSorted(in))
  in match {
    case Cons(head, tail) =>
      if (head < v) {
        Cons(head, insert(tail, v))
      } else if (head == v) {
        insert(tail, head)
      } else {

```

```

        Cons(v, insert(tail, head))
      }
    case Nil =>
      Cons(v, Nil)
  }
} ensuring {
  (out : List) => content(out) == content(in) ++ Set[BigInt](v) && isSorted(out)
}

```

A.4.3 Delete

Problem

```

def delete(in: List, v: BigInt) = {
  require(isSorted(in))

  choose( (res : List) =>
    (content(res) == content(in) -- Set(v)) && isSorted(res)
  )
}

```

Solution

```

def delete(in : List, v : BigInt): List = {
  require(isSorted(in))
  in match {
    case Cons(head, tail) =>
      if (head == v) {
        tail
      } else {
        Cons(head, delete(tail, v))
      }
    case Nil =>
      Nil
  }
} ensuring {
  (res : List) => content(res) == content(in) -- Set[BigInt](v) && isSorted(res)
}

```

A.4.4 Merge

Problem

```

def merge(in1: List, in2: List) = {
  require(isSorted(in1) && isSorted(in2))
  choose {
    (out : List) =>
      (content(out) == content(in1) ++ content(in2)) && isSorted(out)
  }
}

```

```
}  
}
```

Solution

```
def merge(in1 : List, in2 : List): List = {  
  require(isSorted(in1) && isSorted(in2))  
  in1 match {  
    case Cons(head, tail) =>  
      merge(tail, insert(in2, head))  
    case Nil =>  
      in2  
  }  
} ensuring {  
  (out : List) => content(out) == content(in1) ++ content(in2) && isSorted(out)  
}
```

A.5 Unary Numerals

A.5.1 Header

```
import leon.lang._  
import leon.lang.synthesis._  
  
object UnaryNumerals {  
  sealed abstract class Num  
  case object Z extends Num  
  case class S(pred: Num) extends Num  
  
  def value(n: Num): BigInt = {  
    n match {  
      case Z => BigInt(0)  
      case S(p) => BigInt(1) + value(p)  
    }  
  } ensuring (_ ≥ 0)  
  
  def add(x: Num, y: Num): Num = {  
    choose { (r : Num) =>  
      value(r) == value(x) + value(y)  
    }  
  }  
}
```

A.5.2 Add

Problem

```
def add(x: Num, y: Num): Num = {
```

```
    choose { (r : Num) ⇒  
      value(r) == value(x) + value(y)  
    }  
  }
```

Solution

```
def add(x : Num, y : Num): Num = {  
  x match {  
    case S(pred) ⇒  
      add(pred, S(y))  
    case Z ⇒  
      y  
  }  
} ensuring {  
  (r : Num) ⇒ value(r) == value(x) + value(y)  
}
```

A.5.3 Distinct**Problem**

```
def distinct(x: Num, y: Num): Num = {  
  choose { (r : Num) ⇒  
    r != x && r != y  
  }  
}
```

Solution

```
def distinct(x : Num, y : Num): Num = {  
  S(add(y, x))  
} ensuring {  
  (r : Num) ⇒ r != x && r != y  
}
```

A.5.4 Mult**Problem**

```
def mult(x: Num, y: Num): Num = {  
  choose { (r : Num) ⇒  
    value(r) == value(x) * value(y)  
  }  
}
```

Solution

```
def mult(x : Num, y : Num): Num = {  
  x match {  
    case S(pred) =>  
      add(y, mult(pred, y))  
    case Z =>  
      Z  
  }  
} ensuring {  
  (r : Num) => value(r) == value(x) * value(y)  
}
```

A.6 Batched Queue

A.6.1 Header

```
import leon.lang._  
import leon.lang.synthesis._  
  
object BatchedQueue {  
  sealed abstract class List[T] {  
    def content: Set[T] = {  
      this match {  
        case Cons(h, t) => Set(h) ++ t.content  
        case Nil() => Set()  
      }  
    }  
  
    def size: BigInt = {  
      this match {  
        case Cons(h, t) => BigInt(1) + t.size  
        case Nil() => BigInt(0)  
      }  
    }  
  } ensuring { _ ≥ 0 }  
  
  def reverse: List[T] = {  
    this match {  
      case Cons(h, t) => t.reverse.append(Cons(h, Nil[T]() ))  
      case Nil() => Nil[T]()  
    }  
  } ensuring { res =>  
    this.content == res.content  
  }  
  
  def append(r: List[T]): List[T] = {  
    this match {  
      case Cons(h, t) => Cons(h, t.append(r))  
      case Nil() => r  
    }  
  }  
}
```



```

    }
  }

  def isEmpty: Boolean = {
    this == Nil[T]()
  }

  def tail: List[T] = {
    require(this != Nil[T]())
    this match {
      case Cons(h, t) => t
    }
  }

  def head: T = {
    require(this != Nil[T]())
    this match {
      case Cons(h, t) => h
    }
  }
}

case class Cons[T](h: T, t: List[T]) extends List[T]
case class Nil[T]() extends List[T]

case class Queue[T](f: List[T], r: List[T]) {
  def content: Set[T] = f.content ++ r.content
  def size: BigInt = f.size + r.size

  def isEmpty: Boolean = f.isEmpty && r.isEmpty

  def invariant: Boolean = {
    (f.isEmpty) ==> (r.isEmpty)
  }

  def toList: List[T] = f.append(r.reverse)
}

```

A.6.2 Dequeue

Problem

```

def dequeue[T](q: Queue[T]): Queue[T] = {
  require(q.invariant && !q.isEmpty)

  choose { (res: Queue[T]) =>
    res.size == q.size-1 && res.toList == q.toList.tail && res.invariant
  }
}

```

```
}
}
```

Solution

```
def dequeue[T](q : Queue[T]): Queue[T] = {
  require(q.invariant && !q.isEmpty)
  q.f match {
    case Cons(h, t) =>
      t match {
        case Cons(h1, t1) =>
          Queue[T](Cons[T](h1, t1), q.r)
        case Nil() =>
          Queue[T](q.r.reverse, Nil[T]())
      }
    case Nil() =>
      error[Queue[T]]("Impossible")
  }
} ensuring {
  (res : Queue[T]) => res.size == q.size - BigInt(1) && res.toList == q.toList.tail && res.invariant
}
```

A.7 Address Book

A.7.1 Header

```
import leon.annotation._
import leon.lang._
import leon.lang.synthesis._

object AddressBook {

  case class Address[A](info: A, priv: Boolean)

  sealed abstract class AddressList[A] {
    def size: BigInt = {
      this match {
        case Nil() => BigInt(0)
        case Cons(head, tail) => BigInt(1) + tail.size
      }
    }
    ensuring { res => res ≥ 0 }

    def content: Set[Address[A]] = this match {
      case Nil() => Set[Address[A]]()
      case Cons(addr, l1) => Set(addr) ++ l1.content
    }

    def ++(that: AddressList[A]): AddressList[A] = {
```

```

    this match {
      case Cons(h, t) ⇒ Cons(h, t ++ that)
      case Nil() ⇒ that
    }
  } ensuring {
    res ⇒ res.content == this.content ++ that.content
  }
}

case class Cons[A](a: Address[A], tail: AddressList[A]) extends AddressList[A]
case class Nil[A]() extends AddressList[A]

def allPersonal[A](l: AddressList[A]): Boolean = l match {
  case Nil() ⇒ true
  case Cons(a, l1) ⇒
    if (a.priv) allPersonal(l1)
    else false
}

def allBusiness[A](l: AddressList[A]): Boolean = l match {
  case Nil() ⇒ true
  case Cons(a, l1) ⇒
    if (a.priv) false
    else allBusiness(l1)
}

case class AddressBook[A](business: AddressList[A], personal: AddressList[A]) {
  def size: BigInt = business.size + personal.size

  def content: Set[Address[A]] = business.content ++ personal.content

  @inline
  def invariant = {
    allPersonal(personal) && allBusiness(business)
  }
}

```

A.7.2 Merge

Problem

```

def merge[A](a1: AddressBook[A], a2: AddressBook[A]): AddressBook[A] = {
  require(a1.invariant && a2.invariant)

  choose( (res: AddressBook[A]) ⇒
    res.personal.content == (a1.personal.content ++ a2.personal.content) &&
    res.business.content == (a1.business.content ++ a2.business.content) &&

```

Appendix A. Synthesis Benchmarks and Solutions

```
    res.invariant
  )
}
```

Solution

```
def merge[A](a1 : AddressBook[A], a2 : AddressBook[A]): AddressBook[A] = {
  require(a1.invariant && a2.invariant)
  AddressBook[A](a2.business ++ a1.business, a2.personal ++ a1.personal)
} ensuring {
  (res : AddressBook[A]) =>
    res.personal.content == a1.personal.content ++ a2.personal.content &&
    res.business.content == a1.business.content ++ a2.business.content &&
    res.invariant
}
```

B Repair Benchmarks and Solutions

For each repair benchmark, we first provide the correct program that contains all the necessary definitions, including the data-structures and auxiliary abstraction functions. Then, we provide for each benchmark the function in which we injected an error that we give as input to Leon. We add a comment indicating where the error is located as well as its nature. Leon repairs all benchmark in the way intended by the comment.

B.1 Compiler

B.1.1 Complete File

```
import leon.lang._
import leon.annotation._
import leon.collection._
import leon._

object Trees {
  abstract class Expr
  case class Plus(lhs: Expr, rhs: Expr) extends Expr
  case class Minus(lhs: Expr, rhs: Expr) extends Expr
  case class LessThan(lhs: Expr, rhs: Expr) extends Expr
  case class And(lhs: Expr, rhs: Expr) extends Expr
  case class Or(lhs: Expr, rhs: Expr) extends Expr
  case class Not(e : Expr) extends Expr
  case class Eq(lhs: Expr, rhs: Expr) extends Expr
  case class Ite(cond: Expr, thn: Expr, els: Expr) extends Expr
  case class IntLiteral(v: BigInt) extends Expr
  case class BoolLiteral(b : Boolean) extends Expr
}

object Types {
  abstract class Type
  case object IntType extends Type
}
```

Appendix B. Repair Benchmarks and Solutions

```
case object BoolType extends Type
}

object TypeChecker {
  import Trees._
  import Types._

  def typeOf(e : Expr) : Option[Type] = e match {
    case Plus(l,r) => (typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(IntType)
      case _ => None()
    }
    case Minus(l,r) => (typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(IntType)
      case _ => None()
    }
    case LessThan(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(IntType), Some(IntType)) => Some(BoolType)
      case _ => None()
    }
    case And(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(BoolType), Some(BoolType)) => Some(BoolType)
      case _ => None()
    }
    case Or(l,r) => ( typeOf(l), typeOf(r)) match {
      case (Some(BoolType), Some(BoolType)) => Some(BoolType)
      case _ => None()
    }
    case Not(e) => typeOf(e) match {
      case Some(BoolType) => Some(BoolType)
      case _ => None()
    }
    case Eq(lhs, rhs) => (typeOf(lhs), typeOf(rhs)) match {
      case (Some(t1), Some(t2)) if t1 == t2 => Some(BoolType)
      case _ => None()
    }
    case Lte(c, th, el) => (typeOf(c), typeOf(th), typeOf(el)) match {
      case (Some(BoolType), Some(t1), Some(t2)) if t1 == t2 => Some(t1)
      case _ => None()
    }
    case IntLiteral(_) => Some(IntType)
    case BoolLiteral(_) => Some(BoolType)
  }

  def typeChecks(e : Expr) = typeOf(e).isDefined
}
```

```

object Semantics {
  import Trees._
  import Types._
  import TypeChecker._

  def seml(t : Expr) : BigInt = {
    require( typeOf(t) == ( Some(IntType) : Option[Type] ))
    t match {
      case Plus(lhs , rhs) => seml(lhs) + seml(rhs)
      case Minus(lhs , rhs) => seml(lhs) - seml(rhs)
      case lte(cond, thn, els) =>
        if (semB(cond)) seml(thn) else seml(els)
      case IntLiteral(v) => v
    }
  }

  def semB(t : Expr) : Boolean = {
    require( (Some(BoolType): Option[Type]) == typeOf(t))
    t match {
      case And(lhs, rhs ) => semB(lhs) && semB(rhs)
      case Or(lhs , rhs ) => semB(lhs) || semB(rhs)
      case Not(e) => !semB(e)
      case LessThan(lhs, rhs) => seml(lhs) < seml(rhs)
      case lte(cond, thn, els) =>
        if (semB(cond)) semB(thn) else semB(els)
      case Eq(lhs, rhs) => (typeOf(lhs), typeOf(rhs)) match {
        case ( Some(IntType), Some(IntType) ) => seml(lhs) == seml(rhs)
        case ( Some(BoolType), Some(BoolType) ) => semB(lhs) == semB(rhs)
      }
      case BoolLiteral(b) => b
    }
  }

  def b2i(b : Boolean): BigInt = if (b) 1 else 0

  @induct
  def semUntyped( t : Expr) : BigInt = { t match {
    case Plus (lhs, rhs) => semUntyped(lhs) + semUntyped(rhs)
    case Minus(lhs, rhs) => semUntyped(lhs) - semUntyped(rhs)
    case And (lhs, rhs) => if (semUntyped(lhs)!=0) semUntyped(rhs) else BigInt(0)
    case Or(lhs, rhs ) =>
      if (semUntyped(lhs) == 0) semUntyped(rhs) else BigInt(1)
    case Not(e) =>
      b2i(semUntyped(e) == 0)
    case LessThan(lhs, rhs) =>

```

Appendix B. Repair Benchmarks and Solutions

```
    b2i(semUntyped(lhs) < semUntyped(rhs))
  case Eq(lhs, rhs) =>
    b2i(semUntyped(lhs) == semUntyped(rhs))
  case lte(cond, thn, els) =>
    if (semUntyped(cond) == 0) semUntyped(els) else semUntyped(thn)
  case IntLiteral(v) => v
  case BoolLiteral(b) => b2i(b)
}} ensuring { res => typeOf(t) match {
  case Some(IntType) => res == semI(t)
  case Some(BoolType) => res == b2i(semB(t))
  case None() => true
}}
}

object Desugar {
  import Types._
  import TypeChecker._
  import Semantics.b2i

  abstract class SimpleE
  case class Plus(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class Neg(arg : SimpleE) extends SimpleE
  case class lte(cond : SimpleE, thn : SimpleE, els : SimpleE) extends SimpleE
  case class Eq(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class LessThan(lhs : SimpleE, rhs : SimpleE) extends SimpleE
  case class Literal(i : BigInt) extends SimpleE

  @induct
  def desugar(e : Trees.Expr) : SimpleE = { e match {
    case Trees.Plus (lhs, rhs) => Plus(desugar(lhs), desugar(rhs))
    case Trees.Minus(lhs, rhs) => Plus(desugar(lhs), Neg(desugar(rhs)))
    case Trees.LessThan(lhs, rhs) => LessThan(desugar(lhs), desugar(rhs))
    case Trees.And (lhs, rhs) => lte(desugar(lhs), desugar(rhs), Literal(0))
    case Trees.Or (lhs, rhs) => lte(desugar(lhs), Literal(1), desugar(rhs))
    case Trees.Not(e) => lte(desugar(e), Literal(0), Literal(1))
    case Trees.Eq(lhs, rhs) =>
      Eq(desugar(lhs), desugar(rhs))
    case Trees.lte(cond, thn, els) => lte(desugar(cond), desugar(thn), desugar(els))
    case Trees.IntLiteral(v) => Literal(v)
    case Trees.BoolLiteral(b) => Literal(b2i(b))
  }} ensuring { res =>
    sem(res) == Semantics.semUntyped(e)
  }

  def sem(e : SimpleE) : BigInt = e match {
```



```

    case Plus(lhs, rhs) => sem(lhs) + sem(rhs)
    case Lte(cond, thn, els) => if (sem(cond) != 0) sem(thn) else sem(els)
    case Neg(arg) => -sem(arg)
    case Eq(lhs, rhs) => b2i(sem(lhs) == sem(rhs))
    case LessThan(lhs, rhs) => b2i(sem(lhs) < sem(rhs))
    case Literal(i) => i
  }
}

object Evaluator {
  import Trees._

  def bTob(b: Boolean): BigInt = if (b) 1 else 0
  def iTob(i: BigInt) = i == 1

  def eval(e: Expr): BigInt = {
    e match {
      case Plus(lhs, rhs) => eval(lhs) + eval(rhs)
      case Minus(lhs, rhs) => eval(lhs) - eval(rhs)
      case LessThan(lhs, rhs) => bTob(eval(lhs) < eval(rhs))
      case And(lhs, rhs) => bTob(iTob(eval(lhs)) && iTob(eval(rhs)))
      case Or(lhs, rhs) => bTob(iTob(eval(lhs)) || iTob(eval(rhs)))
      case Not(e) => bTob(!iTob(eval(e)))
      case Eq(lhs, rhs) => bTob(eval(lhs) == eval(rhs))
      case Lte(cond, thn, els) => if (iTob(eval(cond))) eval(thn) else eval(els)
      case IntLiteral(v) => v
      case BoolLiteral(b) => bTob(b)
    }
  }
}

object Simplifier {
  import Trees._
  import Evaluator._

  @induct
  def simplify(e: Expr): Expr = {
    e match {
      case And(BoolLiteral(false), _) => BoolLiteral(false)
      case Or(BoolLiteral(true), _) => BoolLiteral(true)
      case Plus(IntLiteral(a), IntLiteral(b)) => IntLiteral(a+b)
      case Not(Not(Not(a))) => Not(a)
      case e => e
    }
  }
  ensuring {
    res => eval(res) == eval(e)
  }
}

```

```
}  
}
```

B.1.2 Desugar 1

```
def desugar(e: Expr): SimpleE = {  
  e match {  
    case Plus(lhs, rhs) =>  
      Neg(desugar(lhs)) // FIXME: Should be Plus(desugar(lhs), desugar(rhs))  
    case Minus(lhs, rhs) =>  
      Plus(desugar(lhs), Neg(desugar(rhs)))  
    case LessThan(lhs, rhs) =>  
      LessThan(desugar(lhs), desugar(rhs))  
    case And(lhs, rhs) =>  
      lte(desugar(lhs), desugar(rhs), Literal(BigInt(0)))  
    case Or(lhs, rhs) =>  
      lte(desugar(lhs), Literal(BigInt(1)), desugar(rhs))  
    case Not(e) =>  
      lte(desugar(e), Literal(BigInt(0)), Literal(BigInt(1)))  
    case Eq(lhs, rhs) =>  
      Eq(desugar(lhs), desugar(rhs))  
    case lte(cond, thn, els) =>  
      lte(desugar(cond), desugar(thn), desugar(els))  
    case IntLiteral(v) =>  
      Literal(v)  
    case BoolLiteral(b) =>  
      Literal(b2i(b))  
  }  
}
```

B.1.3 Desugar 2

```
def desugar(e: Expr): SimpleE = {  
  e match {  
    case Plus(lhs, rhs) =>  
      Plus(desugar(lhs), desugar(rhs))  
    case Minus(lhs, rhs) =>  
      Literal(0) // FIXME: Should be Plus(desugar(lhs), Neg(desugar(rhs)))  
    case LessThan(lhs, rhs) =>  
      LessThan(desugar(lhs), desugar(rhs))  
    case And(lhs, rhs) =>  
      lte(desugar(lhs), desugar(rhs), Literal(BigInt(0)))  
    case Or(lhs, rhs) =>  
      lte(desugar(lhs), Literal(BigInt(1)), desugar(rhs))  
    case Not(e) =>  
      lte(desugar(e), Literal(BigInt(0)), Literal(BigInt(1)))  
    case Eq(lhs, rhs) =>  
      Eq(desugar(lhs), desugar(rhs))  
  }  
}
```

```

    case lte(cond, thn, els) =>
      lte(desugar(cond), desugar(thn), desugar(els))
    case IntLiteral(v) =>
      Literal(v)
    case BoolLiteral(b) =>
      Literal(b2i(b))
  }
}

```

B.1.4 Desugar 3

```

def desugar(e: Expr): SimpleE = {
  e match {
    case Plus(lhs, rhs) =>
      Plus(desugar(lhs), desugar(rhs))
    case Minus(lhs, rhs) =>
      Plus(desugar(lhs), Neg(desugar(rhs)))
    case LessThan(lhs, rhs) =>
      LessThan(desugar(lhs), desugar(rhs))
    case And(lhs, rhs) =>
      lte(desugar(lhs), desugar(rhs), Literal(BigInt(0)))
    case Or(lhs, rhs) =>
      lte(desugar(lhs), Literal(BigInt(1)), desugar(rhs))
    case Not(e) =>
      lte(desugar(e), Literal(BigInt(0)), Literal(BigInt(1)))
    case Eq(lhs, rhs) =>
      Eq(desugar(lhs), desugar(rhs))
    case lte(cond, thn, els) =>
      lte(desugar(cond), desugar(els), desugar(thn)) // FIXME: swap then/else
    case IntLiteral(v) =>
      Literal(v)
    case BoolLiteral(b) =>
      Literal(b2i(b))
  }
}

```

B.1.5 Desugar 4

Broken

```

def desugar(e: Expr): SimpleE = {
  e match {
    case Plus(lhs, rhs) =>
      Plus(desugar(lhs), desugar(rhs))
    case Minus(lhs, rhs) =>
      Plus(desugar(lhs), Neg(desugar(rhs)))
    case LessThan(lhs, rhs) =>
      LessThan(desugar(lhs), desugar(rhs))

```

Appendix B. Repair Benchmarks and Solutions

```
case And(lhs, rhs) =>
  lte(desugar(lhs), desugar(rhs), Literal(BigInt(0)))
case Or(lhs, rhs) =>
  lte(desugar(lhs), Literal(BigInt(1)), desugar(rhs))
case Not(e) =>
  lte(desugar(e), Literal(BigInt(1)), Literal(BigInt(1))) // FIXME should be 0
case Eq(lhs, rhs) =>
  Eq(desugar(lhs), desugar(rhs))
case lte(cond, thn, els) =>
  lte(desugar(cond), desugar(thn), desugar(els))
case IntLiteral(v) =>
  Literal(v)
case BoolLiteral(b) =>
  Literal(b2i(b))
}
```

B.1.6 Desugar 5

```
def desugar(e: Expr): SimpleE = {
  e match {
    case Plus(lhs, rhs) =>
      Plus(desugar(lhs), desugar(rhs))
    case Minus(lhs, rhs) =>
      Plus(desugar(lhs), Neg(desugar(rhs)))
    case LessThan(lhs, rhs) =>
      LessThan(desugar(lhs), desugar(rhs))
    case And(lhs, rhs) =>
      lte(desugar(lhs), desugar(rhs), Literal(BigInt(0)))
    case Or(lhs, rhs) =>
      lte(desugar(lhs), Literal(BigInt(1)), desugar(rhs))
    case Not(e) =>
      lte(desugar(e), Literal(BigInt(1)), Literal(BigInt(1))) // FIXME: error 1: Should be 0
    case Eq(lhs, rhs) =>
      Eq(desugar(lhs), desugar(rhs))
    case lte(cond, thn, els) =>
      lte(desugar(cond), desugar(els), desugar(thn)) // FIXME: error2: swapped then/else
    case IntLiteral(v) =>
      Literal(v)
    case BoolLiteral(b) =>
      Literal(b2i(b))
  }
}
```

B.1.7 Simplify

```
def simplify(e: Expr): Expr = {
  e match {
```

```

    case And(BoolLiteral(false), _) =>
      BoolLiteral(false)
    case Or(BoolLiteral(true), _) =>
      BoolLiteral(true)
    case Plus(IntLiteral(a), IntLiteral(b)) =>
      IntLiteral(a-b) // FIXME Should be IntLiteral(a+b). Leon fixes with 'e'
    case Not(Not(Not(a))) =>
      Not(a)
    case e =>
      e
  }
} ensuring {
  res => eval(res) == eval(e)
}

```

B.2 Heap

B.2.1 Complete File

```

import leon.lang._
import leon.collection._

object Heaps {

  sealed abstract class Heap {
    val rank : BigInt = this match {
      case Leaf() => 0
      case Node(_, l, r) =>
        1 + max(l.rank, r.rank)
    }
    def content : Set[BigInt] = this match {
      case Leaf() => Set[BigInt]()
      case Node(v,l,r) => l.content ++ Set(v) ++ r.content
    }
  }
  case class Leaf() extends Heap
  case class Node(value: BigInt, left: Heap, right: Heap) extends Heap

  def max(i1: BigInt, i2: BigInt) = if (i1 ≥ i2) i1 else i2

  def hasHeapProperty(h : Heap) : Boolean = h match {
    case Leaf() => true
    case Node(v, l, r) =>
      ( l match {
        case Leaf() => true
        case n@Node(v2,_,_) => v ≥ v2 && hasHeapProperty(n)
      }) &&

```

Appendix B. Repair Benchmarks and Solutions

```
( r match {
  case Leaf() => true
  case n@Node(v2,_,_) => v ≥ v2 && hasHeapProperty(n)
})
}

def hasLeftistProperty(h: Heap) : Boolean = h match {
  case Leaf() => true
  case Node(_,l,r) =>
    hasLeftistProperty(l) &&
    hasLeftistProperty(r) &&
    l.rank ≥ r.rank
}

def heapSize(t: Heap): BigInt = { t match {
  case Leaf() => BigInt(0)
  case Node(v, l, r) => heapSize(l) + 1 + heapSize(r)
}} ensuring(_ ≥ 0)

private def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1,h2) match {
    case (Leaf(), _) => h2
    case (_, Leaf()) => h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) =>
      if(v1 ≥ v2)
        makeN(v1, l1, merge(r1, h2))
      else
        makeN(v2, l2, merge(h1, r2))
  }
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(h1) + heapSize(h2) == heapSize(res) &&
  h1.content ++ h2.content == res.content
}

private def makeN(value: BigInt, left: Heap, right: Heap) : Heap = {
  require(
    hasLeftistProperty(left) && hasLeftistProperty(right)
  )
  if(left.rank ≥ right.rank)
    Node(value, left, right)
  else
    Node(value, right, left)
}
```

```

} ensuring { res ⇒
  hasLeftistProperty(res) }

def insert(element: BigInt, heap: Heap) : Heap = {
  require(hasLeftistProperty(heap) && hasHeapProperty(heap))

  merge(Node(element, Leaf(), Leaf()), heap)

} ensuring { res ⇒
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(res) == heapSize(heap) + 1 &&
  res.content == heap.content ++ Set(element)
}

def findMax(h: Heap) : Option[BiInt] = {
  h match {
    case Node(m,_,_) ⇒ Some(m)
    case Leaf() ⇒ None()
  }
}

def removeMax(h: Heap) : Heap = {
  require(hasLeftistProperty(h) && hasHeapProperty(h))
  h match {
    case Node(_,l,r) ⇒ merge(l, r)
    case l ⇒ l
  }
} ensuring { res ⇒
  hasLeftistProperty(res) && hasHeapProperty(res)
}

}

```

B.2.2 Merge 1

```

def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1,h2) match {
    case (Leaf(), _) ⇒ h2
    case (_, Leaf()) ⇒ h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
      if(v1 ≥ v2) // FIXME swapped the branches
        makeN(v2, l2, merge(h1, r2))
      else
        makeN(v1, l1, merge(r1, h2))
  }
}

```

```
}  
} ensuring { res ⇒  
  hasLeftistProperty(res) && hasHeapProperty(res) &&  
  heapSize(h1) + heapSize(h2) == heapSize(res) &&  
  h1.content ++ h2.content == res.content  
}
```

B.2.3 Merge 2

```
def merge(h1: Heap, h2: Heap) : Heap = {  
  require(  
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&  
    hasHeapProperty(h1) && hasHeapProperty(h2)  
  )  
  (h1, h2) match {  
    case (Leaf(), _) ⇒ h2  
    case (_, Leaf()) ⇒ h2 // FIXME should be h1  
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒  
      if(v1 ≥ v2)  
        makeN(v1, l1, merge(r1, h2))  
      else  
        makeN(v2, l2, merge(h1, r2))  
  }  
} ensuring { res ⇒  
  hasLeftistProperty(res) && hasHeapProperty(res) &&  
  heapSize(h1) + heapSize(h2) == heapSize(res) &&  
  h1.content ++ h2.content == res.content  
}
```

B.2.4 Merge 3

```
def merge(h1: Heap, h2: Heap) : Heap = {  
  require(  
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&  
    hasHeapProperty(h1) && hasHeapProperty(h2)  
  )  
  (h1, h2) match {  
    case (Leaf(), _) ⇒ h2  
    case (_, Leaf()) ⇒ h1  
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒  
      if(v1 ≤ v2) // FIXME should be ≥  
        makeN(v1, l1, merge(r1, h2))  
      else  
        makeN(v2, l2, merge(h1, r2))  
  }  
} ensuring { res ⇒  
  hasLeftistProperty(res) && hasHeapProperty(res) &&  
  heapSize(h1) + heapSize(h2) == heapSize(res) &&  
  h1.content ++ h2.content == res.content  
}
```



```

    h1.content ++ h2.content == res.content
}

```

B.2.5 Merge 4

```

def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1,h2) match {
    case (Leaf(), _) => h2
    case (_, Leaf()) => h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) =>
      if(v1 ≥ v2)
        makeN(v1, l1, merge(r1, h2))
      else
        makeN(v2, l1, merge(h1, r2)) // FIXME: l1 instead of l2
  }
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(h1) + heapSize(h2) == heapSize(res) &&
  h1.content ++ h2.content == res.content
}

```

B.2.6 Merge 5

```

def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1,h2) match {
    case (Leaf(), _) => h2
    case (_, Leaf()) => h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) =>
      if(v1 + v2 > 0) // FIXME Totally wrong, should be v1 ≥ v2
        makeN(v1, l1, merge(r1, h2))
      else
        makeN(v2, l2, merge(h1, r2))
  }
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(h1) + heapSize(h2) == heapSize(res) &&
  h1.content ++ h2.content == res.content
}

```

B.2.7 Merge 6

Appendix B. Repair Benchmarks and Solutions

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(
    hasLeftistProperty(h1) && hasLeftistProperty(h2) &&
    hasHeapProperty(h1) && hasHeapProperty(h2)
  )
  (h1,h2) match {
    case (Leaf(), _) => h1 // FIXME: swapped these cases
    case (_, Leaf()) => h2 // FIXME: swapped these cases
    case (Node(v1, l1, r1), Node(v2, l2, r2)) =>
      if(v1 ≥ v2)
        makeN(v1, l1, merge(r1, h2))
      else
        makeN(v2, l2, merge(h1, r2))
  }
} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(h1) + heapSize(h2) == heapSize(res) &&
  h1.content ++ h2.content == res.content
}
```

B.2.8 Insert

```
def insert(element: BigInt, heap: Heap) : Heap = {
  require(hasLeftistProperty(heap) && hasHeapProperty(heap))

  merge(Node(element + 1, Leaf(), Leaf()), heap) // FIXME: unneeded +1

} ensuring { res =>
  hasLeftistProperty(res) && hasHeapProperty(res) &&
  heapSize(res) == heapSize(heap) + 1 &&
  res.content == heap.content ++ Set(element)
}
```

B.2.9 Make Node

```
def makeN(value: BigInt, left: Heap, right: Heap) : Heap = {
  require(
    hasLeftistProperty(left) && hasLeftistProperty(right)
  )
  if(left.rank ≥ right.rank + 42) // FIXME unneeded constant
    Node(value, left, right)
  else
    Node(value, right, left)
} ensuring { res =>
  hasLeftistProperty(res)
}
```

B.3 List

B.3.1 Complete File

```

import leon._
import leon.lang._
import leon.collection._
import leon.annotation._

sealed abstract class List[T] {
  def size: BigInt = (this match {
    case Nil() ⇒ BigInt(0)
    case Cons(h, t) ⇒ BigInt(1) + t.size
  }) ensuring (_ ≥ 0)

  def content: Set[T] = this match {
    case Nil() ⇒ Set()
    case Cons(h, t) ⇒ Set(h) ++ t.content
  }

  def contains(v: T): Boolean = (this match {
    case Cons(h, t) if h == v ⇒ true
    case Cons(_, t) ⇒ t.contains(v)
    case Nil() ⇒ false
  }) ensuring { res ⇒ res == (content contains v) }

  def ++(that: List[T]): List[T] = (this match {
    case Nil() ⇒ that
    case Cons(x, xs) ⇒ Cons(x, xs ++ that)
  }) ensuring { res ⇒ (res.content == this.content ++ that.content) && (res.size == this.size + that.size) }

  def head: T = {
    require(this != Nil[T]())
    this match {
      case Cons(h, t) ⇒ h
    }
  }

  def tail: List[T] = {
    require(this != Nil[T]())
    this match {
      case Cons(h, t) ⇒ t
    }
  }

  def apply(index: BigInt): T = {
    require(0 ≤ index && index < size)

```

Appendix B. Repair Benchmarks and Solutions

```
    if (index == 0) {
      head
    } else {
      tail(index-1)
    }
  }
}

def ::(t:T): List[T] = Cons(t, this)

def :+(t:T): List[T] = {
  this match {
    case Nil() ⇒ Cons(t, this)
    case Cons(x, xs) ⇒ Cons(x, xs :+ (t))
  }
} ensuring (res ⇒ (res.size == size + 1) && (res.content == content ++ Set(t)))

def reverse: List[T] = {
  this match {
    case Nil() ⇒ this
    case Cons(x,xs) ⇒ xs.reverse :+ x
  }
} ensuring (res ⇒ (res.size == size) && (res.content == content))

def take(i: BigInt): List[T] = (this, i) match {
  case (Nil(), _) ⇒ Nil()
  case (Cons(h, t), i) ⇒
    if (i == 0) {
      Nil()
    } else {
      Cons(h, t.take(i-1))
    }
}

def drop(i: BigInt): List[T] = (this, i) match {
  case (Nil(), _) ⇒ Nil()
  case (Cons(h, t), i) ⇒
    if (i == 0) {
      Cons(h, t)
    } else {
      t.drop(i-1)
    }
}

def slice(from: BigInt, to: BigInt): List[T] = {
  require(from < to && to < size && from ≥ 0)
  drop(from).take(to-from)
}
```

```

def replace(from: T, to: T): List[T] = this match {
  case Nil() => Nil()
  case Cons(h, t) =>
    val r = t.replace(from, to)
    if (h == from) {
      Cons(to, r)
    } else {
      Cons(h, r)
    }
}

private def chunk0(s: BigInt, l: List[T], acc: List[T], res: List[List[T]], s0: BigInt): List[List[T]] =
  l match {
    case Nil() =>
      if (acc.size > 0) {
        res :+ acc
      } else {
        res
      }
    case Cons(h, t) =>
      if (s0 == 0) {
        chunk0(s, l, Nil(), res :+ acc, s)
      } else {
        chunk0(s, t, acc :+ h, res, s0-1)
      }
  }

def chunks(s: BigInt): List[List[T]] = {
  require(s > 0)

  chunk0(s, this, Nil(), Nil(), s)
}

def zip[B](that: List[B]): List[(T, B)] = (this, that) match {
  case (Cons(h1, t1), Cons(h2, t2)) =>
    Cons((h1, h2), t1.zip(t2))
  case (__, _) =>
    Nil()
}

def -(e: T): List[T] = this match {
  case Cons(h, t) =>
    if (e == h) {
      t - e
    } else {
      Cons(h, t - e)
    }
}

```

```

    }
    case Nil() =>
      Nil()
  }

def --(that: List[T]): List[T] = this match {
  case Cons(h, t) =>
    if (that.contains(h)) {
      t -- that
    } else {
      Cons(h, t -- that)
    }
  case Nil() =>
    Nil()
}

def &(that: List[T]): List[T] = this match {
  case Cons(h, t) =>
    if (that.contains(h)) {
      Cons(h, t & that)
    } else {
      t & that
    }
  case Nil() =>
    Nil()
}

def pad(s: BigInt, e: T): List[T] = { (this, s) match {
  case (_, s) if s ≤ 0 =>
    this
  case (Nil(), s) =>
    Cons(e, Nil().pad(s-1, e))
  case (Cons(h, t), s) =>
    Cons(h, t.pad(s, e))
}} ensuring { res =>
  (s > 0) ==> (res.size == this.size + s && res.contains(e))
}

def find(e: T): Option[BigInt] = this match {
  case Nil() => None()
  case Cons(h, t) =>
    if (h == e) {
      Some(0)
    } else {
      t.find(e) match {
        case None() => None()
        case Some(i) => Some(i+1)
      }
    }
}

```

```

    }
  }
}

def init: List[T] = (this match {
  case Cons(h, Nil()) =>
    Nil[T]()
  case Cons(h, t) =>
    Cons[T](h, t.init)
  case Nil() =>
    Nil[T]()
}) ensuring ( (r: List[T]) => ((r.size < this.size) || (this.size == 0)) )

def lastOption: Option[T] = this match {
  case Cons(h, t) =>
    t.lastOption.orElse(Some(h))
  case Nil() =>
    None()
}

def firstOption: Option[T] = this match {
  case Cons(h, t) =>
    Some(h)
  case Nil() =>
    None()
}

def unique: List[T] = this match {
  case Nil() => Nil()
  case Cons(h, t) =>
    Cons(h, t.unique - h)
}

def splitAt(e: T): List[List[T]] = split(Cons(e, Nil()))

def split(seps: List[T]): List[List[T]] = this match {
  case Cons(h, t) =>
    if (seps.contains(h)) {
      Cons(Nil(), t.split(seps))
    } else {
      val r = t.split(seps)
      Cons(Cons(h, r.head), r.tail)
    }
  case Nil() =>
    Cons(Nil(), Nil())
}

```

Appendix B. Repair Benchmarks and Solutions

```
def count(e: T): BigInt = this match {  
  case Cons(h, t) =>  
    if (h == e) {  
      1 + t.count(e)  
    } else {  
      t.count(e)  
    }  
  case Nil() =>  
    0  
}
```

```
def evenSplit: (List[T], List[T]) = {  
  val c = size/2  
  (take(c), drop(c))  
}
```

```
def insertAt(pos: BigInt, l: List[T]): List[T] = {  
  if(pos < 0) {  
    insertAt(size + pos, l)  
  } else if(pos == 0) {  
    l ++ this  
  } else {  
    this match {  
      case Cons(h, t) =>  
        Cons(h, t.insertAt(pos-1, l))  
      case Nil() =>  
        l  
    }  
  }  
}
```

```
def replaceAt(pos: BigInt, l: List[T]): List[T] = {  
  if(pos < 0) {  
    replaceAt(size + pos, l)  
  } else if(pos == 0) {  
    l ++ this.drop(l.size)  
  } else {  
    this match {  
      case Cons(h, t) =>  
        Cons(h, t.replaceAt(pos-1, l))  
      case Nil() =>  
        l  
    }  
  }  
}
```

```
def rotate(s: BigInt): List[T] = {
```



```

    if (s < 0) {
      rotate(size+s)
    } else {
      val s2 = s % size
      drop(s2) ++ take(s2)
    }
  }
}

def isEmpty = this match {
  case Nil() => true
  case _ => false
}

}

@ignore
object List {
  def apply[T](elems: T*): List[T] = ???
}

@library
object ListOps {
  def flatten[T](ls: List[List[T]]): List[T] = ls match {
    case Cons(h, t) => h ++ flatten(t)
    case Nil() => Nil()
  }

  def isSorted(ls: List[BigInt]): Boolean = ls match {
    case Nil() => true
    case Cons(_, Nil()) => true
    case Cons(h1, Cons(h2, _)) if (h1 > h2) => false
    case Cons(_, t) => isSorted(t)
  }

  def sorted(ls: List[BigInt]): List[BigInt] = ls match {
    case Cons(h, t) => insSort(sorted(t), h)
    case Nil() => Nil()
  }

  def insSort(ls: List[BigInt], v: BigInt): List[BigInt] = ls match {
    case Nil() => Cons(v, Nil())
    case Cons(h, t) =>
      if (v ≤ h) {
        Cons(v, t)
      } else {
        Cons(h, insSort(t, v))
      }
  }
}

```

```
}  
}
```

```
case class Cons[T](h: T, t: List[T]) extends List[T]  
case class Nil[T]() extends List[T]
```

B.3.2 Pad

```
def pad(s: BigInt, e: T): List[T] = {  
  (this, s) match {  
    case (_, s) if s ≤ 0 ⇒  
      this  
    case (Nil(), s) ⇒  
      Cons(e, Nil().pad(s-1, e))  
    case (Cons(h, t), s) ⇒  
      Cons(h, t.pad(s-1, e)) // FIXME should be s, not s-1  
  }  
} ensuring { res ⇒  
  (s > 0) ⇒ (res.size == this.size + s && res.contains(e))  
}
```

B.3.3 ++

```
def ++(that: List[T]): List[T] = (this match {  
  case Nil() ⇒ that  
  case Cons(x, xs) ⇒ xs ++ that // FIXME forgot Cons(x, ..)  
}) ensuring { res ⇒  
  (res.content == this.content ++ that.content) &&  
  (res.size == this.size + that.size)  
}
```

B.3.4 :+

```
def :+(t:T): List[T] = {  
  this match {  
    case Nil() ⇒ this // FIXME forgot t  
    case Cons(x, xs) ⇒ Cons(x, xs :+ (t))  
  }  
} ensuring { res ⇒  
  (res.size == size + 1) &&  
  (res.content == content ++ Set(t))  
}
```

B.3.5 Replace

```
def replace(from: T, to: T): List[T] = {  
  this match {
```

```

    case Nil() ⇒ Nil[T]()
    case Cons(h, t) ⇒
      val r = t.replace(from, to)
      if (h != from) { // FIXME should be ==
        Cons(to, r)
      } else {
        Cons(h, r)
      }
  }
}
}} ensuring { res ⇒
  (((this.content -- Set(from)) ++ (if (this.content contains from) Set(to) else Set[T]() == res.
    content) &&
    res.size == this.size
  })
}

```

B.3.6 Count

```

def count(e: T): BigInt = {
  this match {
    case Cons(h, t) ⇒
      if (h == e) {
        t.count(e) // FIXME missing +1
      } else {
        t.count(e)
      }
    case Nil() ⇒
      BigInt(0)
  }
} ensuring {((this, e), _) passes {
  case (Cons(a, Cons(b, Cons(a1, Cons(b2, Nil())))), a2) if a == a1 && a == a2 && b != a2 && b2
    != a2 ⇒ 2
  case (Cons(a, Cons(b, Nil())), c) if a != c && b != c ⇒ 0
}}
}

```

B.3.7 Find 1

```

def find(e: T): Option[BiInt] = {
  this match {
    case Nil() ⇒ None[BiInt]()
    case Cons(h, t) ⇒
      if (h == e) {
        Some(BiInt(0))
      } else {
        t.find(e) match {
          case None() ⇒ None[BiInt]()
          case Some(i) ⇒ Some(i) // FIXME forgot +1
        }
      }
  }
}

```

```
}  
} ensuring { res =>  
  if (this.content contains e) {  
    res.isDefined && this.size > res.get && this.apply(res.get) == e && res.get ≥ 0  
  } else {  
    res.isEmpty  
  }  
}
```

B.3.8 Find 2

Broken

```
def find(e: T): Option[BigInt] = {  
  this match {  
    case Nil() => None[BigInt]()  
    case Cons(h, t) =>  
      if (h == e) {  
        Some(BigInt(0))  
      } else {  
        t.find(e) match {  
          case None() => None[BigInt]()  
          case Some(i) => Some(i + 2) // FIXME +1  
        }  
      }  
  }  
}  
} ensuring { res =>  
  if (this.content contains e) {  
    res.isDefined && this.size > res.get && this.apply(res.get) == e && res.get ≥ 0  
  } else {  
    res.isEmpty  
  }  
}
```

B.3.9 Find 3

```
def find(e: T): Option[BigInt] = {  
  this match {  
    case Nil() => None[BigInt]()  
    case Cons(h, t) =>  
      if (h != e) { // FIXME should be ==  
        Some(BigInt(0))  
      } else {  
        t.find(e) match {  
          case None() => None[BigInt]()  
          case Some(i) => Some(i+1)  
        }  
      }  
  }  
}
```

```

    }
  } ensuring { res ⇒
    if (this.content contains e) {
      res.isDefined && this.size > res.get && this.apply(res.get) == e && res.get ≥ 0
    } else {
      res.isEmpty
    }
  }
}

```

B.3.10 Size

Broken

```

def size: BigInt = {
  this match {
    case Nil() ⇒ BigInt(0)
    case Cons(h, t) ⇒ BigInt(3) + t.size // FIXME 3 -> 1
  }
} ensuring { res ⇒
  res ≥ 0 &&
  ((this, res) passes {
    case Cons(_, Nil()) ⇒ 1
    case Nil() ⇒ 0
  })
}

```

B.3.11 Sum

```

def sum(l: List[BiInt]): BiInt = {
  l match {
    case Nil() ⇒ BiInt(0)
    case Cons(x, xs) ⇒ BiInt(1) + sum(xs) // FIXME x + sum(xs)
  }
} ensuring { res ⇒
  (l, res) passes {
    case Cons(a, Nil()) ⇒ a
    case Cons(a, Cons(b, Nil())) ⇒ a + b
  }
}

```

B.3.12 -

```

def -(e: T): List[T] = {
  this match {
    case Cons(h, t) ⇒
      if (e == h) {
        t // FIXME: t - e
      } else {

```

```

        Cons(h, t - e)
      }
    case Nil() =>
      Nil[T]()
  }
} ensuring { res =>
  res.content == this.content ++ Set(e)
}

```

B.3.13 Drop 1

```

def drop(i: BigInt): List[T] = {
  (this, i) match {
    case (Nil(), _) => Nil[T]()
    case (Cons(h, t), i) =>
      if (i != 0) { // FIXME: should be < 1
        Cons(h, t)
      } else {
        t.drop(i-1)
      }
  }
} ensuring { res =>
  ((this, i), res) passes {
    case (Cons(_, Nil()), BigInt(42)) => Nil()
    case (l@Cons(_, _), BigInt(0)) => l
    case (Cons(a, Cons(b, Nil())), BigInt(1)) => Cons(b, Nil())
  }
}

```

B.3.14 Drop 2

```

def drop(i: BigInt): List[T] = {
  require(i ≥ 0)
  (this, i) match {
    case (Nil(), _) => Nil[T]()
    case (Cons(h, t), i) =>
      if (i == 0) {
        Cons[T](h, t)
      } else {
        t.drop(i) // FIXME Should be -1
      }
  }
} ensuring { (res: List[T]) =>
  ((this, i), res) passes {
    case (Cons(a, Cons(b, Nil())), BigInt(1)) => Cons(b, Nil())
    case (Cons(a, Cons(b, Nil())), BigInt(2)) => Nil()
  }
}

```

B.4 Numerical

B.4.1 Complete File

```

import leon._
import leon.lang._
import leon.annotation._

object Numerical {
  def power(base: BigInt, p: BigInt): BigInt = {
    require(p ≥ BigInt(0))
    if (p == BigInt(0)) {
      BigInt(1)
    } else if (p%BigInt(2) == BigInt(0)) {
      power(base*base, p/BigInt(2))
    } else {
      power(base, p-BigInt(1))*base
    }
  }
  ensuring {
    res ⇒ ((base, p), res) passes {
      case (_, BigInt(0)) ⇒ BigInt(1)
      case (b, BigInt(1)) ⇒ b
      case (BigInt(2), BigInt(7)) ⇒ BigInt(128)
      case (BigInt(2), BigInt(10)) ⇒ BigInt(1024)
    }
  }
}

def moddiv(a: BigInt, b: BigInt): (BigInt, BigInt) = {
  require(a ≥ BigInt(0) && b > BigInt(0));
  if (b > a) {
    (a, BigInt(0))
  } else {
    val (r1, r2) = moddiv(a-b, b)
    (r1, r2+1)
  }
}
ensuring {
  res ⇒ b*res._2 + res._1 == a
}
}

```

B.4.2 power

```

def power(base: BigInt, p: BigInt): BigInt = {
  require(p ≥ BigInt(0))
  if (p == BigInt(0)) {
    BigInt(1)
  } else if (p%BigInt(2) == BigInt(0)) {
    power(base*base, p/BigInt(2))
  }
}

```

```
    } else {
      power(base, p-BigInt(1)) // FIXME: missing base*
    }
  } ensuring {
    res ⇒ ((base, p), res) passes {
      case (_, BigInt(0)) ⇒ BigInt(1)
      case (b, BigInt(1)) ⇒ b
      case (BigInt(2), BigInt(7)) ⇒ BigInt(128)
      case (BigInt(2), BigInt(10)) ⇒ BigInt(1024)
    }
  }
}
```

B.4.3 ModDiv

```
def moddiv(a: BigInt, b: BigInt): (BigInt, BigInt) = {
  require(a ≥ BigInt(0) && b > BigInt(0));
  if (b > a) {
    (BigInt(1), BigInt(0)) // FIXME: should be (a, 0)
  } else {
    val (r1, r2) = moddiv(a-b, b)
    (r1, r2+1)
  }
} ensuring {
  res ⇒ b*res._2 + res._1 == a
}
```

B.5 Merge Sort

B.5.1 Complete File

```
import leon.collection._

object MergeSort {

  def split(l : List[BigInt]) : (List[BigInt],List[BigInt]) = { l match {
    case Cons(a, Cons(b, t)) ⇒
      val (rec1, rec2) = split(t)
      (Cons(a, rec1), Cons(b, rec2))
    case other ⇒ (other, Nil[BigInt]())
  }} ensuring { res ⇒
    val (l1, l2) = res
    l1.size ≥ l2.size &&
    l1.size ≤ l2.size + 1 &&
    l1.size + l2.size == l.size &&
    l1.content ++ l2.content == l.content
  }

  def isSorted(l : List[BigInt]) : Boolean = l match {
```



```

    case Cons(x, t@Cons(y, _)) => x ≤ y && isSorted(t)
    case _ => true
  }

def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) =>
      if (h1 ≤ h2)
        Cons(h1, merge(t1, l2))
      else
        Cons(h2, merge(l1, t2))
    case (Nil(), _) => l2
    case (_, Nil()) => l1
  }
} ensuring { res =>
  isSorted(res) &&
  res.size == l1.size + l2.size &&
  res.content == l1.content ++ l2.content
}

def mergeSort(l : List[BigInt]) : List[BigInt] = { l match {
  case Nil() => l
  case Cons(_, Nil()) => l
  case other =>
    val (l1, l2) = split(other)
    merge(mergeSort(l1), mergeSort(l2))
}} ensuring { res =>
  isSorted(res) &&
  res.content == l.content &&
  res.size == l.size
}
}

```

B.5.2 Split

Broken

```

def split(l : List[BigInt]) : (List[BigInt], List[BigInt]) = { l match {
  case Cons(a, Cons(b, t)) =>
    val (rec1, rec2) = split(t)
    (rec1, Cons(b, rec2)) // FIXME: Forgot a
  case other => (other, Nil[BigInt]())
}} ensuring { res =>
  val (l1, l2) = res
  l1.size ≥ l2.size &&
  l1.size ≤ l2.size + 1 &&
  l1.size + l2.size == l.size &&

```

Appendix B. Repair Benchmarks and Solutions

```
l1.content ++ l2.content == l.content
}
```

Repaired

```
def split(l : List[BigInt]) : (List[BigInt], List[BigInt]) = { l match {
  case Cons(a, Cons(b, t)) =>
    val (rec1, rec2) = split(t)
    (Cons(a, rec1), Cons(b, rec2))
  case other => (other, Nil[BigInt]())
}} ensuring { res =>
  val (l1, l2) = res
  l1.size ≥ l2.size &&
  l1.size ≤ l2.size + 1 &&
  l1.size + l2.size == l.size &&
  l1.content ++ l2.content == l.content
}
```

B.5.3 Merge 1

```
def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) =>
      if (h1 ≤ h2)
        Cons(h1, merge(t1, l2))
      else
        Cons(h1, merge(l1, t2)) // FIXME: h1 -> h2
    case (Nil(), _) => l2
    case (_, Nil()) => l1
  }
} ensuring { res =>
  isSorted(res) &&
  res.size == l1.size + l2.size &&
  res.content == l1.content ++ l2.content
}
```

B.5.4 Merge 2

```
def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) =>
      if (h1 ≥ h2) // FIXME: Condition inverted
        Cons(h1, merge(t1, l2))
      else
        Cons(h2, merge(l1, t2))
    case (Nil(), _) => l2
  }
}
```

```

    case (_, Nil()) => l1
  }
} ensuring { res =>
  isSorted(res) &&
  res.size == l1.size + l2.size &&
  res.content == l1.content ++ l2.content
}

```

B.5.5 Merge 3

```

def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) =>
      if (h1 ≤ h2)
        Cons(h1, merge(t1, l2))
      else
        merge(l1, t2) // FIXME: missing h2
    case (Nil(), _) => l2
    case (_, Nil()) => l1
  }
} ensuring { res =>
  isSorted(res) &&
  res.size == l1.size + l2.size &&
  res.content == l1.content ++ l2.content
}

```

B.5.6 Merge 4

```

def merge(l1 : List[BigInt], l2 : List[BigInt]) : List[BigInt] = {
  require(isSorted(l1) && isSorted(l2))
  (l1, l2) match {
    case (Cons(h1, t1), Cons(h2, t2)) =>
      if (h1 ≤ h2)
        Cons(h1, merge(t1, l2))
      else
        Cons(h2, merge(l1, t2))
    case (Nil(), _) => l1 // FIXME should be l2
    case (_, Nil()) => l1
  }
} ensuring { res =>
  isSorted(res) &&
  res.size == l1.size + l2.size &&
  res.content == l1.content ++ l2.content
}

```


Bibliography

- [ABD⁺14] Rajeev Alur, Rastislav Bodik, Eric Dallar, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. To Appear in Marktoberdrof NATO proceedings, 2014. http://sygus.seas.upenn.edu/files/sygus_extended.pdf, retrieved 2015-02-06.
- [ABJ⁺13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
- [AGK13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, pages 934–950, 2013.
- [AMP94] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.
- [BCD⁺11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
- [BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.
- [BDF⁺04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [BGL⁺97] Michael Butler, Jim Grundy, Thomas Langbacka, Rimvydas Ruksenau, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In *Formal Methods Pacific*, 1997.

Bibliography

- [BKKS13] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Scala Workshop*, 2013.
- [BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.
- [BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proc. 30th ACM POPL*, 2003.
- [BN05] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *ECOOP*, 2005.
- [BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. *Principles of Programming Languages*, 2016.
- [BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, 1998.
- [CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis, invited paper. In R.N. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178, Grenoble, France, April 2002. LNCS 2304, Springer, Berlin.
- [CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.
- [CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, 1988.
- [CN02] Ana Cavalcanti and David A. Naumann. Forward simulation for data refinement of classes. In *Proceedings of Formal Methods Europe FME'2002*, volume 2391 of LNCS, pages 471–490, 2002.
- [CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic debugging. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *ICSE*, pages 121–130. ACM, 2011.
- [CWZ90] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.
- [Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In James R. Cordy and Mario Barbacci, editors, *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, USA, 20-23 Apr 1992*, pages 2–13. IEEE, 1992.

-
- [dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
- [dMB09] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, 2009.
- [dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge University Press, 1998.
- [DYDG⁺10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
- [FCD15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239. ACM, 2015.
- [FL03] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.
- [FL11] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, pages 10–30. Springer, 2011.
- [FP01] Pierre Flener and Derek Partridge. Inductive programming. *Autom. Softw. Eng.*, 8(2):131–137, 2001.
- [GBC06] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to C. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 358–371. Springer, 2006.
- [GGJ⁺10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994.
- [GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
- [GKKP13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

Bibliography

- [GMK11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 173–188. Springer, 2011.
- [GNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38(1):54–72, 2012.
- [Hof10] Martin Hofmann. IgorII - an analytical inductive functional programming system (tool demo). In *PEPM*, pages 29–32, 2010.
- [Jac95] Daniel Jackson. Structuring Z specifications with views. *ACM Trans. Softw. Eng. Methodol.*, 4(4):365–389, 1995.
- [JB06] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *EMCAD*, pages 117–124, 2006.
- [JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *POPL*, pages 303–310, 1991.
- [JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 226–238. Springer, 2005.
- [JHS86] He Jifeng, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *ESOP’86*, volume 213 of *LNCS*, 1986.
- [JKS13] Swen Jacobs, Viktor Kuncak, and Philippe Suter. Reductions for synthesis procedures. In *VMCAI*, pages 88–107, 2013.
- [JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL ’87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM.
- [JMT10] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, volume 6337 of *LNCS*, pages 320–339. Springer, 2010.
- [JSGB12] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *JCSS*, 78(2):441–460, 2012.
- [KAE⁺09] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrick, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.
- [KKS12] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

-
- [KKS13] Viktor Kuncak, Etienne Kneuss, and Philippe Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.
- [KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 316–329. ACM, 2010.
- [KMPS12] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *CACM*, 55(2):103–111, 2012.
- [KMPS13] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.
- [KS06] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*, 7:429–454, 2006.
- [LB12] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 133–146. ACM, 2012.
- [LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM PLDI*, Atlanta, GA, June 1988.
- [LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178. ACM, 2015.
- [LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM*, pages 806–809, 2009.
- [LV09] Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. In *FOSSACS*, pages 395–409, 2009.
- [MR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.
- [MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA*, pages 1–11, 2002.
- [MRV11] Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Purity analysis: An abstract interpretation formulation. In *SAS*, volume 6887 of *LNCS*, pages 7–24. Springer, 2011.

Bibliography

- [MRV12] Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *SAS*, volume 7460 of *LNCS*, pages 370–387. Springer, 2012.
- [MW71] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
- [MW80] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
- [NCM⁺10] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In Jan Vitek, editor, *TOOLS (48)*, volume 6141 of *LNCS*, pages 79–96. Springer, 2010.
- [NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
- [NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *ICSE*, pages 772–781. IEEE / ACM, 2013.
- [Ode10] Martin Odersky. Contracts for scala. In *RV*, pages 51–57, 2010.
- [Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
- [OM09] Martin Odersky and Adriaan Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS*, pages 427–451, 2009.
- [PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
- [PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, 1989.
- [PS11] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *ESOP*, pages 439–458, 2011.
- [PWF⁺11] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Code-based automated program fixing. *ArXiv e-prints*, 2011. arXiv:1102.1059.
- [ROH12] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, volume 7313 of *LNCS*, pages 258–282. Springer, 2012.
- [Rou04] Atanas Rountev. Precise identification of side-effect-free methods in java. In *ICSM*, pages 82–91, 2004.

- [RVK15] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 111–124. ACM, 2015.
- [RVY14] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In Michael F. P. O’Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 44. ACM, 2014.
- [Sal06] Alexandru D. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, Massachusetts Institute of Technology, 2006.
- [SDE08] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic generation of local repairs for boolean programs. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–10. IEEE, 2008.
- [SDK10] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210, 2010.
- [SG12] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.
- [SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.
- [SGF13] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
- [SGS13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26. ACM, 2013.
- [Shi88] Olin Shivers. Control-flow analysis in scheme. In *PLDI*, pages 164–174, 1988.
- [SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.
- [SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.
- [SLAT⁺07] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

Bibliography

- [SLJB08] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.
- [SLTB⁺06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
- [Smi05] Douglas R. Smith. Generating programs plus proofs by refinement. In *VSTTE*, pages 182–188, 2005.
- [SNK⁺13] Andrej Spielmann, Andres Nötzli, Christoph Koch, Viktor Kuncak, and Yannis Klonatos. Automatic synthesis of out-of-core algorithms. In *SIGMOD*, 2013.
- [SOE14] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. Cost-aware automatic program repair. In Markus Müller-Olm and Helmut Seidl, editors, *SAS*, volume 8723 of *LNCS*, pages 268–284. Springer, 2014.
- [Sol13] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
- [SR05] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer, 2005.
- [Sum77] Phillip D. Summers. A methodology for LISP program construction from examples. *JACM*, 24(1):161–175, 1977.
- [Sut12a] Philippe Suter. *Programming with Specifications*. PhD thesis, EPFL, December 2012.
- [Sut12b] Philippe Paul Henri Suter. *Programming with Specifications*. PhD thesis, IC, Lausanne, 2012.
- [TD03] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC / SIGSOFT FSE*, pages 188–197, 2003.
- [TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 253–262. ACM, 2005.
- [URD⁺13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.
- [vEJ13] Christian von Essen and Barbara Jobstmann. Program repair without regret. In *CAV*, pages 896–911, 2013.

- [VYY09] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS*, pages 139–154, 2009.
- [WFKM11] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *ICSE*, pages 191–200, 2011.
- [Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.
- [WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, November 1999.
- [YYC08] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234. ACM, 2008.

Etienne Kneuss

Curriculum Vitæ

Education

EPFL, Lausanne, Switzerland	PhD Computer Science	2011 – 2016
EPFL, Lausanne, Switzerland	MS Computer Science	2009 – 2011
EPFL, Lausanne, Switzerland	BS Computer Science	2006 – 2009

Work Experience

Research Assistant, EPFL 2011 – 2016

PhD studies under the supervision of Prof. Viktor Kuncak. My research interests are mainly divided in two research areas. The first one is about the automated reasoning on recursive purely-functional programs with high-level specifications. I have explored automated approaches for program synthesis [3] and repair [2]. In both cases, the goal is to automatically derive an executable implementation that satisfies an existing high-level relational specification. My second area of interest is static analysis techniques for functional and object oriented programs written in Scala. For instance, I investigated techniques to precisely and efficiently analyze memory side-effects in the presence of callbacks [4]. I have also worked on static reasoning techniques for highly-dynamic languages such as PHP [6, 5].

CTO, IMMOMIG Ltd 2005 –

I am responsible for the development and operation of internal tools for a company providing web-based management solutions for real-estate agencies. These tools cover operational aspects of any typical mid-sized company such as: ticketing and support, client-relations, financing computations and simulations, salaries computations, holidays and leaves, and invoicing and payments processing.

Publications

Conference proceedings and journal articles.

- [1] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. “Counter-example complete verification for higher-order functions”. In: *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala 2015, Portland, OR, USA, June 15-17, 2015*. Ed. by Philipp Haller and Heather Miller. ACM, 2015, pp. 18–29. DOI: 10.1145/2774975.2774978. URL: <http://doi.acm.org/10.1145/2774975.2774978>.
- [2] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. “Deductive Program Repair”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. LNCS. Springer, 2015, pp. 217–233. DOI: 10.1007/978-3-319-21668-3_13. URL: http://dx.doi.org/10.1007/978-3-319-21668-3_13.
- [3] Etienne Kneuss et al. “Synthesis modulo recursive functions”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 407–426. DOI: 10.1145/2509136.2509555. URL: <http://doi.acm.org/10.1145/2509136.2509555>.

- [4] Etienne Kneuss, Viktor Kuncak, and Philippe Suter. “Effect Analysis for Programs with Callbacks”. In: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*. Ed. by Ernie Cohen and Andrey Rybalchenko. Vol. 8164. LNCS. Springer, 2013, pp. 48–67. DOI: 10.1007/978-3-642-54108-7_3. URL: http://dx.doi.org/10.1007/978-3-642-54108-7_3.
- [5] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. “Runtime Instrumentation for Precise Flow-Sensitive Type Analysis”. In: *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. Ed. by Howard Barringer et al. Vol. 6418. LNCS. Springer, 2010, pp. 300–314. DOI: 10.1007/978-3-642-16612-9_23. URL: http://dx.doi.org/10.1007/978-3-642-16612-9_23.
- [6] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. “Phantm: PHP analyzer for type mismatch”. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by Gruia-Catalin Roman and Kevin J. Sullivan. ACM, 2010, pp. 373–374. DOI: 10.1145/1882291.1882355. URL: <http://doi.acm.org/10.1145/1882291.1882355>.

Teaching

Teaching Assistantship

- Computer Language Processing (2014)
- Compiler Construction (2010, 2011, 2012, 2013)
- Software Analysis & Verification (2013, 2015)
- Introduction à la programmation (C++) (2013)
- Introduction à la programmation (Java) (2013)
- Informatique I / II (2011)

