Search Techniques for Code Generation

THÈSE N^{O} 6378 (2015)

PRÉSENTÉE LE 30 JANVIER 2015

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

Tihomir GVERO

acceptée sur proposition du jury:

Prof. P. Dillenbourg, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. D. Marinov, rapporteur
Prof. M. Odersky, rapporteur
Prof. E. Yahav, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

To Gordana and Darko...

Acknowledgements

First and foremost, I would like to thank my advisor Viktor Kuncak for his support, without which this dissertation would not have been possible. Viktor is an outstanding person that posses great knowledge, kindness and modesty and I consider myself lucky and privileged for having the chance to work with him. I would also like to thank Viktor for teaching me many research and personal skills, that will help me in the years to come.

I am grateful to Pierre Dillenbourg, Darko Marinov, Martin Odersky and Eran Yahav for taking the time to serve on my thesis committee. I am aware of their busy schedules and I thank them for their time.

Without my co-authors, parts of this dissertation would not exist. I thank Darko Marinov for giving me opportunity to work on program testing and model checking. With Darko I made my first steps in research and those moments I will always keep in my memory with great gratitude, respect and happiness. Like Viktor, Darko is the person of great knowledge and kindness. I thank him for a valuable contribution on UDITA, described in Chapter 4. Next, I would like to thank Ruzica Piskac with whom I spent many hours solving synthesis problems and discussing formalism and algorithms. I will never forget her cheerful nature. I thank her for the contributions on InSynth and PolySynth, described in Chapter 2. I also thank Milos Gligoric, young, brilliant and industrious person, with whom I worked on many class projects during our bachelor and master studies. Our great collaboration culminated in UDITA paper. I would also like to thank Ivan Kuraj for a great help with the InSynth's implementation and for the numerous hours we spend discussing the main challenges and techniques which we can apply to make the implementation more efficient. Lastly, I would like to thank Marcelo d'Amorim, Brett Daniel, Danny Dig, Peli de Halleux, Vilas Jagannath, Sarfraz Khurshid, Aleksandar Milicevic, Sasa Misailovic and Nikolai Tillmann for helping me during my PhD studies.

I thank the members of the LARA group, past or present, who made my work more enjoyable in many different ways : Ali, Andrej, Andrew, Etienne, Eva, Emmanouil, Filip, Giuliano, Hossein, Ivan, Mikaël, Philippe, Ravi, Ruzica, Régis, Swen, Philipp and Pierre-Emmanuel. I thank the members of the LAMP group, including Alex, Heather, Hubert, Ingo, Iulian, Lukas, Miguel, Manohar, Nada, Sandro, Philipp and Vlad for their help in taming Scala and its compiler, and for the stimulating conversations. I thank Danielle Chamberlain and Yvette Gallay for their mastery of the bureaucratic ways. I thank Fabien Salvi for keeping our technical infrastructure

running, and for rescuing me whenever my urge to switch to the latest technology proved damaging.

I thank my friends Alex, Drazen and Pedja for the many great moments we have spent together. I know that without them this would have been much tougher road. I also thank my girlfriend Alba for giving me a lot of support and having a lot of understanding and patience with me and my busy work schedule.

Finally, I thank my parents for their continuous support. In particular, I thank my mother that always believed in me and my father for teaching me important life skills.

Lausanne, 14 January 2015 Tihomir Gvero

Preface

What are meaningful computer programs? What is the process by which developers transform software requirements into concrete pieces of software? The research presented in this dissertation develops algorithms that automatically explore the space of meaningful pieces of software. These tools take a description of requirements and offer to the developer a set of programs that satisfy these requirements.

A practical way to introduce such tools is to combine them with auto-completion facility of integrated development environments. In previous tools, auto-completion has been largely confined to listing the methods that can be invoked on an object of a given class. Chapter 2 shows that it is possible to do much more: it presents tools that suggest to the developers entire code fragments. A hard constraint on such code fragments is that they type check according to the type rules of a programming language. Exploring the space of well-typed expressions is closely related to the question of type inhabitation from type theory, and is therefore connected to intuitionistic theorem proving through Curry-Howard isomorphism. The algorithms presented in this dissertation can efficiently perform theorem proving, especially when the search space has a high degree of branching, as is the case in the presence of large programming libraries. Another important aspect of the presented tool is that it generates a representation of all type-correct expressions, then allows the developer to choose one of them to be inserted into the program. Because the set of possible well-typed expressions is often infinite, it is crucial to also automatically prioritize the expressions that are likely to be useful to the developers. This is where the presented techniques start to incorporate statistical and heuristic reasoning: expressions using more frequently appearing components, as well as those more local to the current scope should be prioritized.

Chapter 3 takes the statistical reasoning much further by building a statistical model of Java expressions, namely a probabilistic context-free grammar that refines the grammar of Java by lexicalizing it with concrete names of methods from Java libraries, and introduces probabilities derived from millions of files available from public software repositories. In addition to the use of a statistical model for Java code generation, Chapter 3 presents a unique approach to accept descriptions of the intended code fragment in the form of free-form text that mixes English sentences and Java constructs.

This combination results in a tool that bridges an unprecedentedly large range between free-form text input given by the user and executable Java code that the tool generates. Bridging this gap required modifying the natural language processing tools to accept input that includes programming language constructs. It also required robustly extracting useful information

from the generated parse trees and using this information to guide the generation of Java code. Examples shown in Chapter 3 illustrate remarkable performance of the tool in several cases. Among other aspects, the results illustrate that the tool looks beyond the literal spelling of words and finds methods that refer to synonyms and related words. The tool does so by leveraging semantic relationships of WordNet, adapted to the jargon of words used in the names of Java methods from a corpus of code.

In other cases, a cost-effective approach to generate code is to provide the developer or a test engineer with a language in which they can describe test inputs. Chapter 4 presents such a language, UDITA, formulated as a non-deterministic extension of Java. The work presents the implementation of UDITA as well its evaluation in finding real-world software bugs. Given a UDITA program, the set of all its executions is a concise description of a potentially large set of test inputs. The chapter presents several optimizations that exponentially speed up the enumeration of executions of UDITA programs. Among these optimizations is delaying non-deterministic choices and avoiding isomorphisms when generating graph structures. Thanks to these optimizations, UDITA was successful in finding real bugs in compilers, virtual machines, and integrated development environments.

Concrete examples of tools presented in the dissertation explore different possible divisions of tasks between the automated exploration of the space of programs and user input that directs this exploration. In each case, the message is that there is a higher-level approach to constructing software, with the potential to make developers more productive and focused more on high-level design decisions as opposed to low-level coding details.

Lausanne, 13 January 2015 Viktor Kuncak

Abstract

This dissertation explores techniques that synthesize and generate program fragments and test inputs. The main goal of these techniques is to improve and support automation in program synthesis and test input generation. This is important because performing those processes manually is often tedious, time consuming and error prone. The main challenge that these techniques face is exploring the search space in efficient and scalable ways.

In the first part of the dissertation, we present tools InSynth and PolySynth that interactively synthesize code fragments. They take as input a partial program and automatically extract type information, the desired type, and set of visible declarations. They use this input to synthesize ranked list of expressions with the desired type. Finally, they present the expressions to a developer in similar manner to code completions in modern IDEs. InSynth is the first tool that uses a complete algorithm to generate expressions with first class functions and higher order functions. We present the theoretical foundation of the InSynth problem, that is based on *type inhabitation*, and the type-based backward search algorithm that solves it. PolySynth uses type-driven, resolution based algorithm that considers polymorphic types (generics) to generate expressions. Furthermore, the uniqueness of both tools comes from the fact that their algorithms operate using corpus statistics. The statistics are used to steer the algorithms and the search space exploration towards the most relevant solutions. To show InSynth's practical usefulness we built 50 examples based on the real word code that demonstrate proper usage of API. To build the benchmark we randomly choose an expression in the example, remove it and try to recover it with InSynth. The results show that InSynth can recover the expected expression in 96% of the examples, in a short period of time.

In the second part of the dissertation we present the tool anyCode that uses natural language input to synthesize expressions. As input it accepts English words or Java program language constructs. This allows a developer to encode her intuition about the desired expression using words or the expression that approximates the desired structure. Thanks to this flexibility, anyCode can also repair broken expressions. It uses a pipeline of natural language and related-word tools to analyze the input. This helps anyCode to identify the set of the most relevant components and to reduce the size of search space. To further reduce the size of search space and to create the most relevant expressions, anyCode uses two statistical models: unigram and probabilistic context free grammar. To demonstrate the anyCode's generation power we write 60 examples with a text description as input, and an expected expression, that properly uses API, as output. The results show that anyCode generates 93% of expected expressions in

a short period of time and that it greatly benefits from both statistical models.

Finally, in the last part of the dissertation we present UDITA, a Java-like language with support for non-determinism, which allows a user to describe *test generation programs*. Test generation programs are run on a top of Java PathFinder (JPF), a popular explicit-state model checker, that has a built-in backtracking mechanism and supports non-determinism. Using UDITA programs, JPF generates test inputs. The first benefit of UDITA is that non-determinism empowers a user to describe *many* test inputs as easily as describing a *single* test input. The second benefit is that it gives a user more flexibility allowing her to describe test generation programs by arbitrarily combining filters and generators. UDITA reduces the size of search space using an algorithm that reduces the number of generated complex isomorphic structures and that delays non-deterministic choices. We demonstrate UDITA expressiveness and usefulnesses by generating test programs for Java compilers, refactoring engines (in Eclipse and NetBeans) and an early implementation of our UDITA algorithm. Using the generated programs we manage to discover numerous bugs in those tools, including some previously unknown.

Key words: Program Synthesis, Test Input Generation, Interactive Synthesis, Natural Language Processing

Résumé

Cette thèse explore les techniques qui synthétisent et génèrent les fragments de programmes et les entrées de test. Le principal objectif de ces techniques est d'améliorer et de supporter l'automation dans le programme de synthèses et la génération d'entrées de test. Cela est important parce que l'exécution de ces processus manuellement est souvent fastidieuse, c'est une perte de temps et source d'erreurs. Le principal défi que ces techniques doivent relever est celui d'explorer l'espace de recherche d'une manière efficace et paramétrable.

Dans la première partie de la dissertation, nous présentons les outils InSynth et PolySynth qui synthétisent interactivement les fragments du code. Comme entrée, ils prennent un programme partiel et extraient automatiquement des types d'informations, le type désiré, et l'ensemble des déclarations visibles. Ils utilisent cette entrée pour synthétiser la liste ordonnée des expressions avec le type désiré. À la fin, ils présentent les expressions à un concepteur d'une manière analogue pour coder les applications dans des IDE modernes. InSynth est le premier outil qui utilise un algorithme complet pour générer des expressions avec des fonctions de première classe et des fonctions d'ordre supérieur. Nous présentons le fondement théorique du problème InSynth, qui est basé sur le type inhabitation, et l'algorithme basé sur le type rétrogressif qui le résout. PolySynth utilise l'algorithme de guidage, basé sur la résolution, qui prend en compte les types polymorphiques (génériques) pour générer des expressions. De plus, l'unicité des deux outils provient du fait que leurs algorithmes opèrent en utilisant les statistiques du corpus. Les statistiques sont utilisées pour diriger les algorithmes et l'exploration de l'espace de recherche vers les solutions les plus pertinentes. Pour montrer l'utilité pratique de InSynth, nous avons construit 50 exemples basés sur de code tiré du monde réel qui démontrent l'utilisation appropriée d'API. Pour construire un étalon nous choisissons au hasard une expression dans l'exemple, nous la retirons et essayons de la recouvrer au moyen de InSynth. Les résultats démontrent que InSynth peut recouvrer l'expression attendue dans 96% des exemples, dans un laps de temps court.

Dans la deuxième partie de la dissertation, nous présentons l'outil anyCode qui utilise l'entrée du langage naturel pour synthétiser les expressions. Comme entrée, il accepte les mots anglais ou les constructions du langage du programme Java. Cela permet au concepteur d'encoder son intuition sur l'expression désirée en utilisant les mots ou l'expression qui se rapprochent de la structure désirée. Grâce à cette flexibilité, anyCode peut aussi réparer les expressions démontées. Il utilise le pipeline du langage naturel et les outils destinés aux mots pour analyser l'entrée. Cela aide anyCode à identifier l'ensemble des composantes les plus pertinentes et à

réduire la taille de l'espace de recherche. Pour réduire encore la taille de l'espace de recherche et pour créer les expressions les plus pertinentes, anyCode utilise les deux modèles statistiques : unigramme et la grammaire probabiliste hors contexte. Pour faire une démonstration du pouvoir de la génération anyCode, nous écrivons 60 étalons avec une description textuelle comme entrée, et une expression attendue, qui utilise adéquatement API, comme sortie. Les résultats montrent qu'anyCode génère 93% des expressions attendues dans un laps de temps court et qu'il tire grand profit des deux modèles statistiques.

Enfin, dans la dernière partie de la dissertation, nous présentons UDITA, un langage à la Java qui supporte le non-déterminisme, ce qui permet à l'utilisateur de décrire des générations de programmes de test. Des générations de programmes de test fonctionnent au-dessus de Java PathFinder (JPF), un model checker explicite, qui possède un mécanisme interne de retour en arrière et supporte le non-déterminisme. En utilisant les programmes UDITA, JPF génère les entrées de test. Le premier avantage de UDITA est que le non-déterminisme autorise un utilisateur à décrire de nombreuses entrées de test aussi facilement qu'à décrire une seule entrée de test. Le second avantage est que cela donne à l'utilisateur plus de flexibilité qui lui permet de décrire la génération des programmes de test en combinant arbitrairement les filtres et les générateurs. UDITA réduit la taille de l'espace de recherche en utilisant un algorithme qui réduit le nombre de structures générées isomorphiques complexes et qui retarde les choix non-déterministes. Nous prouvons l'expressivité et l'utilité de UDITA en générant les programmes de test pour les compilateurs Java, moteurs de refactoring (dans Eclipse et NetBeans) et une application précoce de notre algorithme UDITA. En utilisant les programmes générés nous réussissons à découvrir les nombreux bugs dans ces outils, y compris certains inconnus auparavant.

Mots clefs : Programme de Synthèses, Génération d'Entrée de Test, Synthèse Interactive, Traitement du Langage Naturel

Contents

List of Figures

List of Tables

1 Introduction

In this dissertation we present guided techniques that automatically construct and enumerate distinct structures and make program synthesis and test input generation more effective and efficient. Before we proceed, we will introduce a few terms. First, *program synthesis* is a technique that automatically generates a program or a program fragment from a given specification. Second, *automated test input generation* is a technique that automatically generates a set of test inputs from a given description. Finally, a *guided* technique is a technique that is directed by an external entity which can be either a user or a statistical model. The goal of the specification or the description is to describe the search space a system needs to explore to generate relevant programs, program fragments or test inputs. However, the search space is often very large and the main challenge that the systems face is *efficient search space exploration.* In this dissertation we describe our tools that use a specification or a description to automatically synthesize and generate test inputs, programs and program fragments. The tools employ techniques that efficiently explore the search space by narrowing, pruning and reducing it to generate the most interesting outputs.

Program synthesis is often defined as a method that automatically constructs a program from a high-level specification. The specification is usually written explicitly in a language that raises the level of abstraction closer to the language of requirements. Frequently, it is used to describe *what* the desired program should perform [55, 57, 84, 85], in contrast to imperative code that describes *how* the program is executed. This gives a power to a programmer to choose between an explicit specification and imperative code when encoding and solving a problem. This is important, because some programming problems are easier to express using a specification and others using imperative code. In contrast to explicitly written specification, the implicit specification is automatically extracted from the structure of the partial program. The implicit specification is very convenient for capturing the system properties that are important for synthesizing program fragments [47, 65, 80, 82, 88]. The program fragment synthesis generates code that fills in holes in a given partial program, using available information about the structure of the program. The information is automatically extracted using rules that depend on a domain where the synthesis is applied. For instance, if the domain is expression synthesis,

the goal of extraction is to collect all visible declarations in scope of a given program hole. The collected declarations represent the implicit specification and are used to define space of relevant expressions that can fill in the hole. Furthermore, the implicit specification can be combined (strengthened) with the explicit specification. This is done for two reasons: (1) to encode developers intuition, additional knowledge, about the desired expression, and (2) to narrow and speed up the search space exploration. For instance, to encode knowledge, a developer may want to explicitly specify a desired expression type, an approximate structure of the desired expression or a set of words that will appear in the expression.

In contrast to the goal of program synthesis, the goal of software testing is evaluation of a system or its components with the intent to find whether they satisfy the specified requirements or not. Software testing is a widely used technique in the industrial software development process. It includes system or component code execution to identify errors or missing requirements. However, executing the code is not simple because the system and the components often need input supply. Therefore, to execute and test them, one needs to create test inputs. Moreover, it is desirable to create test inputs that allow exploring different execution scenarios of the code. This is important because maximizing the number of different test inputs minimizes the chance of missing a potential bug in the component. Creating these test inputs manually is challenging, slow and error prone. Therefore, the tools that automate the test input generation are important. As input they take a description that describes a set of test inputs. The description can be explicit, written explicitly by a user, or implicit, automatically extracted from code under test. A user can write an explicit description using a declarative or an imperative style or the combination of the two. On the other hand, techniques like symbolic [24, 54] and concolic execution [20, 26, 32, 44, 75, 83, 90] are used to extract an implicit description. They collect conditions along execution paths, in code under test, to create the implicit description. The implicit description is a formula that captures relations among the inputs. The techniques automatically extract the formula and solve it for the inputs. However, sometime these techniques need to be guided by the explicit description [53], to generate complex test inputs. This is an example of the mixed explicit and implicit description. In this context an explicit declaration is used for the same two reasons as an explicit specification in program synthesis: (1) to encode additional knowledge about a domain, and (2) to narrow and speed up the search space exploration.

Test input generation can be used to generate code (to test programming language tools like compilers, interpreters and refactoring engines) and to generate other forms of structures data. Therefore, a *test input* is a more general term than a *program* and a *program fragment*. Also, test input generation systems often generate code fragments that initialize and set primitive or complex data test inputs.

The point where the two often differ is quantity, order and priority of generated output. As mentioned, the goal of test input generation is to systematically test a component and thus generate many test inputs that will achieve desired coverage. All test inputs work together to meet this goal. However, the challenge is to find those interesting test inputs given a fairly

small amount of time, because there are other parts of the software life cycle besides software testing that are equally important and also need to be executed. In contrast to test input generation, the goal of program synthesis is to generate a few programs or program fragments that satisfy a specification. Moreover, often one program or fragment is desired, although many can satisfy the specification. Therefore, a program synthesis tool needs to identify the best program or program fragment. To do so, it usually requires additional ranking models, for example, statistical. The tool then uses the model to estimate the quality of the generated code. Like test input generation tools, program synthesis tools often need to synthesize code in a short period of time. The requirement comes from the fact that a synthesis tool must produce desired code automatically before a developer can do it manually.

Program synthesis and test input generation techniques are the two topics of this dissertation, and in the following sections we will outline in more detail the contributions of each.

1.1 Synthesizing and Repairing Code Fragments

In this section we motivate and introduce three tools: InSynth, PolySynth and anyCode that generate code fragments. Moreover, anyCode is able to repair code fragments. We also introduce the related work and the techniques we implement and include in these tools.

To speed up production and build more reliable software a developer usually solves a task by using existing components. The attractiveness of using existing components comes from the fact that they are usually very well written by experienced developers, and are often very reliable, tested and verified. Among other important things, they often represent the best solution in terms of performance. Sometimes developers use them to quickly build prototypes and test the feasibility of new ideas. However, besides all these advantages, the main problem represents a familiarity with their protocol. Namely, components are often shipped as a part of a library and an application programming interface (API). Often, each library and API follow a different protocol. To use them properly, a developer needs to know what components are suitable for a given task and how to combine them such that they run in a desired manner. In other words, a user needs to be familiar with a component and a protocol specification. There are several ways to obtain the specification: (1) by searching and reading API documentation, (2) by searching, comprehending and using examples that demonstrate the proper component usage, and (3) by combining components into code, running the code and observing the code execution.

There are several tools that try to automate these approaches. Prospector [65] is a pioneering tool that employs a graph search to synthesize chains of declaration calls. As input it takes two types: a desired (expected) type and a receiver (object) type. The intuition is that a developer often knows the desired type of the expression she needs, as well as the receiver type. To efficiently traverse the search space, Prospector uses a precomputed graph, whose nodes are types. An edge connects two types if one is a return and the other a receiver type of an API declaration. Prospector uses a shortest path algorithm to find the best chains. It

generates the chain of declaration calls that can be directly fit into context, thanks to the context type information and the receiver object that is integrated into the chain. However, if the declarations, in the chain, need arguments, a developer needs to invoke Prospector a few times to complete the code.

Furthermore, we are today witnessing a massive popularity of the on-line repository host services such as GitHub [5], BitBucket [4], SourceForge [1]. These services allow developers to keep and share their projects. A lot of content in these repositories is publicly available and they are becoming an excellent sources of code examples that can help developers understand APIs.

PARSEWeb [88] and XSnippet [82] are synthesis tools that use statistics extracted from repositories of existing Java code. The tools use queries as input to search for code examples in the repositories. The returned examples are not cleaned and usually do not adapt to the context because they include unnecessary code. The tools use length and usage frequency of the code examples to rank them. In addition, they also rank the examples using the similarity of their enclosing context with the invocation context.

The three tools that we have briefly introduced come with advantages and drawbacks. Prospector synthesizes expressions that may directly fit into the context, but it often generates irrelevant expressions because it does not employ usage statistics when ranking the expressions. On the other hand, PARSEWeb and XSnippet use the statistics to rank solutions, but do not fit them into context. An excellent example of a tool that combines the two advantages is SLANG [80]. It uses repositories to build statistical model based on n-gram, a model often used in natural language tools. SLANG uses context information with the model to complete a partial program and properly fit the code fragments.

In this dissertation we propose the tool InSynth that, like SLANG, combines the two advantages: it uses the statistics from the code corpus and builds the expressions that fits into context. Moreover, it is the first tool with a complete algorithm that generates expressions with first class functions and higher order functions, features of the popular programming languages like Java, C#, Scala and Python. As input InSynth takes a desired type and a set of automatically extracted declarations in scope, visible from the place where InSynth was invoked. As output InSynth produces a ranked list of expressions with the desired type. In other words, the desired type and the declarations are the specification InSynth uses to synthesize the expressions. Unlike Prospector, that takes two types, InSynth requires simpler input and less intervention from a user. Moreover, the InSynth algorithm generates more complex expressions than the ones generated by Prospector. Namely, unlike Prospector that generates a chain of declaration calls, we generate a complete expression, with all declaration arguments. To rank solutions InSynth uses a declaration usage frequency and a novel metric that takes into account the declaration proximity to the invocation place. InSynth uses two metric to calculate declaration priorities. It uses a weights mechanism to implement the declaration prioritization. Because InSynth works in interactive environment it embeds an algorithm that effectively and efficiently generates

expressions in short period of time (usually in less that half a second). The algorithm consists of the two phases: (1) the reduction of the size of search space and (2) the expression construction. The reduction is done by combining a backward search strategy, a succinct type representation and a declaration priority search. The backward search reduces the size of search space by considering only declarations reachable from the desired type. Further, the succinct type representation treats the duplicate argument types as a single argument type. Consequently, this reduces the search space because we explore the space only once per all duplicates. Finally, the declaration priority search steers and accelerates the algorithm towards better solutions. The result of the reduction phase is the finite graph that encodes all possible expressions with the desired type that can be constructed in the search space. The construction phase uses the graph to produce `N` best expressions. As an intermediate result it produces partial expressions. It follows the graph to unfold the partial expression arguments until they become full expressions. While unfolding the arguments the construction chooses the declarations with the highest priority. This ensures that the first `N` synthesized expressions have the highest priority. In both phases, InSynth uses type-driven approach, meaning that it builds the graph and unfolds arguments using type information to select appropriate declarations.

We additionally propose the tool PolySynth that also takes as input a desired type and a set of visible declarations, and returns a list of ranked expressions. Unlike InSynth, it supports polymorphic types, but has limited support for function types. It does not construct new first class functions but uses only ones in scope, visible from the invocation place. PolySynth synthesis algorithm is based on resolution, whose core is unification that helps the algorithm to deal with polymorphic types. As additional input PolySynth can take a set of tests to further filter synthesized expression. For each synthesized expression the filtering is preformed as follows: (1) the expression is inserted into partial program, then (2) the tests are run, and (3) if all succeed, the expression is output to a user.

Another alternative and very appealing way of expressing developers intent is by using the natural language input. A typical example of synthesis tools that use such input is SNIFF [22]. It uses a natural language description as input and outputs a set of code examples. The code examples are selected from the previously prepared source corpus. To select the examples SNIFF uses an algorithm based on a keywords search. Each example is annotated with keywords from the corresponding documentation that describes declarations in the example. To select examples SNIFF uses the input matching score and the example usage frequency. SmartSynth [60] is another tool that uses a natural language text as input. It generates smartphone automation scripts from natural language descriptions. SmartSynth uses natural language processing (NLP) techniques to infer API components and their partial dataflow, from the description. Then it uses a type based synthesis algorithm to construct scripts. Unlike SNIFF, SmartSynth does not use any statistical data to select and sort the solutions. Additionally, G. Little and R. C. Miller [62] propose a tool that translates a small number of keywords into a valid expression. It first matches the API declarations with the keywords. Then, it selects the ones with the best matching score and combines them to produce the expressions. Like SmartSynth, it does not use any API statistics.

The usage of natural language input becomes more important knowing that a large obstacle for beginning developers is a strict structure of a programming language. This usually includes conforming to syntax and language typing rules. Frequently, this is tedious and frustrating because the developer is usually familiar with the concepts that help her solve the main task, but nevertheless she spends considerable amount of time implementing them due to the strict rules. Moreover, we observe that a developer usually knows the approximate structure of the desired code. When the expression that approximates the desired structure is written it often does not meet a programming language strictness requirements. Mostly, such an expression does not type-check, because it misses some declarations or the names of declarations are wrongly specified. However, it often contains enough information to make code repair feasible. This brings up the importance of the techniques that automatically repair code. If we have tools that fix broken code, which still reflects the intended behavior, we would manage to relieve a developer of a language bothersome rules.

Therefore, as the most flexible solution we finally propose anyCode, a tool that takes a free-form query as input and returns a list of ranked expressions. The free-form query is a textual input that represents a list of English words, a Java expression that approximates desired structure or a mixture of the two. Such input allows anyCode to treat two different problems, expression synthesis using text and code repair, as the same problem. Although, it seems that they are different, the same algorithm that uses text to search for the top expressions can be applied to both. The necessary step is to transform an expression that approximates a desired structure into a list of words (text). Once textual representation is obtained we apply a set of NLP and related-word tools to parse the text and identify a semantic structure. To reduce the search space anyCode first uses the semantic structure to select most popular and relevant declarations. This is done by matching the semantic structure with words that appear in declarations. Based on the matching score and the popularity of declarations anyCode selects the most likely ones. The popularity is determined by unigram model collected from a corpus. In the last phase, the selected declarations are unfolded using probabilistic context free grammar (PCFG) model that captures declaration composition information from the corpus. By unfolding declaration arguments anyCode builds partial and full expressions. The expressions with the highest score, based on the two statistical models, are output to a user. Therefore, the statistical models help to find the best solutions and to guide anyCode synthesis algorithms.

1.2 Test Input Generation

Unlike the synthesis tools and techniques we introduced in the previous section that require little input from a user, in this section we introduce UDITA technology that requires more input to generate complex test inputs. The reason is that synthesizing automatically complex test inputs, like programs, is often more challenging than synthesizing expressions. Therefore, it is desirable for a user to precisely describe the complex test inputs and thus help a system efficiently reduce the size of search space. For this reason, UDITA includes both: a language

that allows a user to describe test inputs, and a test generation algorithm. Although it is less automated, UDITA technology allows a user to generate many complex test inputs in short period of time, which makes it useful.

It is hard and tedious to manually write test cases that achieve high code coverage. This is mainly due to a large number of the test cases that a developer needs to write. However, testing is important. It helps identify bugs and build a regression suite that makes an application stable as it shifts through different versions and changes. Without testing, developing complex applications would be impossible. Therefore, an important problem in software engineering is automation of software testing and in particular test input generation.

Many techniques have been developed with the aim to reduce a burden of manual testing. The examples include tools that use an explicit description of tests [16,52] or systems that extract an implicit description using symbolic [24, 54] and concolic execution [20, 26, 32, 44, 53, 75, 83, 90]. The symbolic and concolic execution tools can handle advanced constructs of object-oriented programs, but still struggle with testing units that as input take complex structured data, like Java programs. Automatically handling programs of the *complexity of a compiler* remains challenging for such systematic approaches. Therefore, we aim to create a system that allows a user to encode her intuition and thus scale these systematic approaches, to overcome the challenge.

In this dissertation we propose UDITA, a Java-like language extended with non-deterministic constructs, that helps a user to specify *test generation programs*. The test generation program is an explicit description that the UDITA algorithm uses to generate test inputs. To describe a test generation program a user defines, uses and combines a set of generators and filters. This simultaneously allows him to describe a set of test inputs in a flexible way and to control the size of the search space. A UDITA program runs on a Java PathFinder (JPF), a popular Java explicit-state model checker. JPF is implemented as a Java virtual machine (JVM) with embedded backtracking mechanism, that supports non-determinism. We improve JPF's basic backtracking with a delayed choice technique that postpones a non-deterministic value assignment to a variable until the variable is read for the first time. In addition, we introduce an object pool abstraction that allows a user to describe complex linked data structures. We also implement mechanism that with respect to the abstraction and the test generation program reduces the number of isomorphic data structures. Together with the delayed choice technique, it represents the technique we use to prune the search space. The main advantage of UDITA is simple and concise way of describing the complex test inputs. Thanks to non-determinism, writing a test generation program, that describes *many* test inputs, is as simple as writing single Java code that describes *one* particular test input. In addition, UDITA can be used both for black box and white-box testing. We used UDITA, in a black-box manner, to generate input programs and test different versions of Java compilers, refactoring engines (in Eclipse and NetBeans) and our implementation of the delayed-choice algorithm. Using the input we discovered numerous bugs in the tools, including some previously unknown. We also used UDITA, in a white-box manner, to generate complex data structures, like red black

trees, and to reveal bugs in broken operations over the generated structures. The results show that UDITA manages to revel all the bugs in a short period of time.

1.3 Contributions and Outline

The rest of the dissertation is organized as follows:

- **Chapter 2** presents InSynth, an interactive code synthesis tool. InSynth uses a complete type-driven algorithm that as input takes a desired type and a set of visible declarations, and outputs a ranked list of expressions. The algorithm is complete in a sense that given enough time it can generate and enumerate all expressions with the desired type using the visible declarations. To support this claim, we provide the completeness and soundness proof of the algorithm. We build InSynth algorithm based on results regarding the well-known *type-inhabitation* problem. We make it practical using declaration priority to guide the synthesis and rank the expressions. The priority is estimated using declaration frequency in a corpus and a declaration proximity to InSynth invocation point. The priority of the final expressions is proportional to priorities of the containing declarations. Finally, on a number of real-world examples we show the effectiveness of InSynth. In addition, this chapter presents PolySynth, a tool similar to InSynth, that includes support for polymorphic types and limited support for functions types. To support polymorphic types PolySynth executes a resolution based algorithm that uses unification. Unlike InSynth, in addition to a desired type, PolySynth can accept as input a set of test cases. PolySynth uses the test cases to filter out irrelevant expressions.

- **Chapter 3** describes anyCode, a tool that takes a free-form query as input and returns a set of ranked expressions. The free-form query is a mix of a natural language input and Java programming language constructs. This simultaneously solves two problems: (1) synthesizing expressions from a natural language input, and (2) repairing a broken expression. The anyCode algorithm contains three key phases. In the first phase, anyCode employs an NLP tool pipeline to analyze the input and discover a semantic structure. This includes our custom built map of related words based on WordNet [36], a large lexical database of English. In the second phase, anyCode uses the semantic structure to find and select declarations. In the third and the last phase, anyCode uses the selected declarations to construct (partial) expressions by unfolding their arguments. To unfold the arguments effectively anyCode uses two statistical models: unigram and PCFG. Finally, anyCode uses the same models to rank the final expressions. In this chapter we also show a number of examples that demonstrate effectiveness of anyCode.

- **Chapter 4** describes UDITA, a Java-like language that helps a user to specify a set of test inputs. It extends Java with non-deterministic choice points and the abstraction used to support initialization and generation of complex linked data structures. A UDITA test input description is executed on top of an optimized version of JPF. JPF executes the

specification using built-in backtracking mechanism, that supports non-determinism, to generate a set of test inputs. We optimized JPF by implementing delayed choice execution that postpones the moment of a non-deterministic value assignment. We use UDITA to write generators and filters to create programs and test inputs for Java compilers, refactoring engines and JPF itself. Additionally, we use it to generate complex data structures, like red black trees, to test the different operations applied to those structures. We demonstrate that UDITA can be effectively and efficiently used for both black-box and white-box testing.

- **Chapter 5** concludes the dissertation and highlights selected future work directions.

2 Complete Completion using Types and Weights

Developing modern software typically involves composing functionality from existing libraries. This task is difficult because libraries may expose many methods to the developer. To help developers in such scenarios, we present tools and techniques that synthesize and suggest valid expressions of a given type at a given program point.

We first describe the tool InSynth that as a base technique uses type inhabitation for lambda calculus terms in long normal form. We introduce a succinct representation for type judgements that merges types into equivalence classes to reduce the search space, then reconstructs any desired number of solutions on demand. Furthermore, we introduce a method to rank solutions based on weights derived from a corpus of code. We implemented InSynth and deployed it as a plugin for the Eclipse IDE for Scala. We show that the techniques we incorporated greatly increase the effectiveness of the approach. Our evaluation benchmarks are code examples from programming practice; we make them available for future comparisons.

We additionally describe the tool PolySynth that applies theorem proving technology to synthesize code fragments that use given library functions. To determine candidate code fragments, our approach takes into account polymorphic type constraints as well as test cases. PolySynth interactively displays a ranked list of suggested code fragments that are appropriate for the current program point. We have found PolySynth to be useful for synthesizing code fragments for common programming tasks, and we believe it is a good platform for exploring software synthesis techniques.

2.1 Motivation

Libraries are one of the biggest assets for today's software developers. Useful libraries often evolve into complex application programming interfaces (APIs) with a large number of classes and methods. It can be difficult for a developer to start using such APIs productively, even for simple tasks. Existing Integrated Development Environments (IDEs) help developers to use APIs by providing *code completion* functionality. For example, an IDE can offer a list of

applicable members to a given receiver object, extracted by finding the declared type of the object. Eclipse [40] and IntelliJ [10] recommend methods applicable to an object, and allow the developer to fill in additional method arguments. Such completion typically considers one step of computation. IntelliJ can additionally compose simple method sequences to form a type-correct expression, but requires both the receiver object as well as assistance from the developer to fill in the arguments. These efforts suggest a general direction for improving modern IDEs: introduce the ability to synthesize entire type-correct code fragments and offer them as suggestions to the developer.

In this chapter we describe InSynth and PolySynth, tools for automated synthesis of code snippets. The tools generate and suggest a list of expressions that have a desired type. One observation behind our work is that, in addition to the forward-directed completion in existing tools, developers can benefit from a backward-directed completion. Indeed, when identifying a computation step, the developer often has the type of a desired object in mind. We therefore do not require the developer to indicate a starting value (such as a receiver object) explicitly. Instead, we follow a more ambitious approach that considers all values in the current scope as the candidate leaf values of expressions to be synthesized. Our approach therefore requires fewer inputs than the pioneering work on the Prospector tool [65], or than the recent work of Perelman et al. [77].

Finding a code snippet of the given type leads us directly to the type inhabitation problem: given a desired type T, and a type environment Γ (a map from identifiers to their types), find an expression e of this type T. Formally, find e such that $\Gamma \vdash e : T$. In our deployment, the tools compute Γ from the position of the cursor in the editor buffer. It similarly looks up T by examining the declared type appearing left of the cursor in the editor. The goal of the tools is to find an expression e, and insert it at the current program point, so that the overall program type checks. When there are multiple solutions, the tools prompt the developer to select one, much like in simpler code completion scenarios.

The type inhabitation in the simply typed lambda calculus corresponds to provability in propositional intuitionistic logic; it is decidable and PSPACE-complete [86, 93]. InSynth embeds a version of the algorithm that is complete in the lambda calculus sense (up to $\alpha\beta\eta$-conversion): it is guaranteed to synthesize a lambda expression of the given type, if such an expression exists. Moreover, if there are multiple solutions, it can enumerate all of them. If there are infinitely many solutions, then the algorithm can enumerate any desired finite prefix of the list of all solutions. Note also that each synthesized expression is a complete in that method calls have all of their arguments synthesized. Because of all these aspects of the algorithm we describe our technique as *complete completion*.

We present InSynth algorithm using a calculus of *succinct types*, which we tailored for efficiently solving type inhabitation queries. The calculus computes equivalence classes of types that reduce the search space in goal-directed search, without losing completeness. Moreover, InSynth algorithm generates a representation of *all* solutions, from which it can then extract

any desired finite subset of solutions.

Given a possibility of an infinite number of type inhabitants, it is natural to consider the problem of finding the *best* one. To solve this problem, we introduce *weights* to guide the search and rank the presented solutions. Initially we assign the weight to each type declaration. Those weights play a crucial role in the algorithm, since they guide the search and rank the presented solutions. The weight is defined in a way that a smaller weight indicates a more desirable formula. To estimate the initial weights of declarations we leverage 1) the lexical nesting structure, with closer declarations having lower weight, and 2) implicit statistical information from a corpus of code, with more frequently occurring declarations having smaller weight, and thus being preferred. In addition, we used a corpus of open-source Java and Scala projects as well as the standard Scala library to collect the usage statistics for the initial weights of declarations.

We implemented our tool, InSynth, within the Scala Eclipse plugin. Our experience shows fast response times as well as a high quality of the offered suggestions, even in the presence of thousands of candidate API calls. We evaluated InSynth on a set of 50 benchmarks constructed from examples found on the Web, written to illustrate API usage, as well as examples from larger projects. To estimate the interactive nature of InSynth, we measured the time needed to synthesize the expected snippet. The running times of InSynth were always a fraction of a second. In the great majority of cases we found that the expected snippets were returned among the top dozen solutions.

Furthermore, we evaluated a number of techniques deployed in final version of InSynth, and found that all of them are important for obtaining good results. We also observed that, even for checking existence of terms InSynth outperforms recent propositional intuitionistic provers [37, 71] on our benchmarks. Our overall experience suggests that InSynth is effective in providing help to developers.

Additionally we present the tool PolySynth, that similarly to InSynth, takes the desired type as input and returns a list of ranked expressions, using the algorithm that consider generic types. The support for generic types is a fundamental generalization compared to previous tools, which handled only ground types. With generic types, a finite set of declarations will generate an infinite set of possible values, and the synthesis of a value of a given type becomes undecidable. PolySynth therefore encodes the synthesis problem in first-order logic. This encoding has the property that a value of the desired type can be built from functions of given types iff there exists a proof for the corresponding theorem in first-order logic. It is therefore related to known connections between proof theory and type theory. In type-theoretic terms, PolySynth attempts to check whether there exists a term of a given type in a given polymorphic type environment. If such terms exist, the goal of PolySynth is to produce a finite subset of them, ranked according to some criterion.

PolySynth implements a custom resolution-based algorithm to find multiple proofs representing candidate code fragments. The use of resolution is related to the traditional deductive

program synthesis [66], but our approach attempts to derive code fragments by using type information instead of the code itself. Like InSynth, PolySynth uses declaration weights to guide the algorithm and to rank the expressions. As a post-processing step, PolySynth filters out the candidate code fragments that crash the program, or that violate assertions or postconditions. This functionality incorporates input/output behavior [50], but uses it mostly to improve the precision of the primary mechanism, the type-driven synthesis. We have found PolySynth to be fast enough for interactive use and helpful in synthesizing meaningful code fragments.

2.2 Motivating Examples

In this section we illustrate functionalities of InSynth and PolySynth using six examples. The first three examples illustrate usage of InSynth. The first example is taken from the online repository of Java API usage samples http://www.java2s.com/. The second example is a real-world fragment of the code base of the Scala IDE for Eclipse, http://scala-ide.org/, and requires invoking a higher-order function. For these two examples, the original code imports only declarations from a few specific classes; to make the problems more challenging and illustrate the task that a programmer faces, we import all declarations from packages where those classes reside. The third example illustrates that InSynth supports subtyping. The last three examples illustrate usage of PolySynth. Using those examples we demonstrate support for polymorphic types, and usage of test case and user preferences to filter out irrelevant expressions.

```
import java.io._
object Main {
  def main(args:Array[String]) = {
    var body = "email.txt"
    var sig = "signature.txt"

    var inStream:SequenceInputStream = |

    var eof:Boolean = false
    var byteCount:Int = 0
    while (!eof) {
      var c:Int = inStream.read()
      if (c == -1)
```

```
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(sig), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(sig))
new SequenceInputStream(new FileInputStream(body), new FileInputStream(body))
new SequenceInputStream(new FileInputStream(sig), System.in)
```

Figure 2.1: InSynth suggesting five highest-ranked well-typed expressions synthesized from declarations visible at a given program point

2.2.1 InSynth: Sequence of Streams

In this example the goal is to create a `SequenceInputStream` object, which is a concatenation of two streams. Suppose that the developer has the code shown in the Eclipse editor in Figure 2.1. If she invokes InSynth at the program point indicated by the cursor, in a fraction of a second InSynth displays the ranked list of five expressions. Seeing the list, the developer can decide that e.g. the second expression in the list matches their intention, and select it to be inserted

into the editor buffer.

This example illustrates that InSynth only needs the current program context, and does not require additional information from the developer. InSynth is able to use both imported values (such as the constructors in this example) and locally declared ones (such as body and sig). InSynth supports methods with multiple arguments and synthesizes expressions for each argument. In this example InSynth loads over 3000 declarations from the context, including local and imported values, and finds the expected solution in less than 250 milliseconds.

The effectiveness of InSynth is characterized by both scalability to many declarations and the quality of the returned suggestions. InSynth ranks the resulting expressions according to the weights and selects the ones with the lowest weight. The weights of expressions and types guide the final ranking and also make the search itself more goal-directed and effective. InSynth derives weights from a corpus of declarations, assigning lower weight to declarations appearing more frequently, and therefore favoring their appearance in the suggested fragments over more exotic declarations.

2.2.2 InSynth: Using Higher-Order Functions

We demonstrate the generation of expressions with higher-order functions on real code from the Scala IDE project:

```
import scala.tools.eclipse.javaelements._
import scala.collection.mutable._
trait TypeTreeTraverser {
  val global: tools.nsc.Global
  import global._
  class TreeWrapper(tree: Tree) {
    def filter(p: Tree => Boolean): List[Tree] = {
      val ft:FilterTypeTreeTraverser = ▮
      ft.traverse(tree)
      ft.hits.toList
    }
  }
}
```

The example shows how a developer should properly check if a Scala AST tree satisfies a given property. In the code, the tree is an argument of the class TreeWrapper, whereas the property p is an input of the method filter. The property p is a predicate function that takes the tree and returns **true** if the tree satisfies it. In order to properly use p inside filter, the developer first needs to create an object of the type FilterTypeTreeTraverser. If the developer calls InSynth at the place ▮, the tool offers several expressions, and the one ranked first turns out to be precisely the one found in the original code, namely

```
new FilterTypeTreeTraverser(var1 => p(var1))
```

The constructor FilterTypeTreeTraverser is a higher-order function that takes as input another function, in this case p. In this example, InSynth loads over 4000 initial declarations and finds the snippets in less than 300 milliseconds.

2.2.3 InSynth: Using Subtyping

The next example illustrates a situation often encountered when using java.awt: implementing a getter method that returns a layout of an object Panel stored in a class Drawing. To implement such a method, we use code of the following form.

```
import java.awt._
class Drawing(panel:Panel) {
  def getLayout:LayoutManager = ▮
}
```

Note that handling this example requires support for subtyping, because the type declarations are given by the following code.

```
class Panel extends Container with Accessible { ... }
class Container extends Component {
 ...
 def getLayout():LayoutManager = { ... }
}
```

The Scala compiler has access to the information about all supertypes of all types in a given scope. InSynth supports subtyping and, in 426 milliseconds, returns a number of solutions among which the second one is the desired expression panel.getLayout(). While doing so, it examines 4965 declarations.

For more experience with InSynth, we encourage the reader to download it from:

http://lara.epfl.ch/w/insynth

The rest of the chapter describes a formalization of the problem that InSynth solves as well as the algorithms we designed to solve it. We then describe the implementation and the evaluation, provide a survey of related efforts, and conclude.

2.2.4 PolySynth: Parametric polymorphism

We next illustrate the support of parametric polymorphism in PolySynth. Consider the standard higher-order function map that applies a given function to each element of the list. Assume that the map function is in the scope. Further assume that we wish to define a method that takes as arguments a function from integers to strings and a list of strings, and returns a list of strings.

```
def map[A,B](f:A ⇒ B, l:List[A]):List[B] = { ... }
def stringConcat(lst:List[String]):String = { ... }
def printInts(intList:List[Int], prn:Int ⇒ String):String = ▮
```

PolySynth returns stringConcat(map[Int,String](prn,intList)) as a result, instantiating polymorphic definition of map and composing it with stringConcat. PolySynth efficiently handles polymorphic types through resolution and unification.

2.2.5 PolySynth: Using code behavior

The next example shows how PolySynth applies testing to discard those snippets that would make code inconsistent.

```
class Mode(mode:String)          class File(name:String, val state:Mode)
object FileManager {
  private final val WRITE:Mode = new Mode("write")
  private final val READ:Mode = new Mode("read")
  def openForReading(name:String):File = ▮
    ensuring { result => result.state == READ}
}
object Tests { FileManager.openForReading("book.txt") }
```

The Scala object FileManager contains methods for opening files either for reading or for writing. If it were based only on types, PolySynth would return both **new** File(name,WRITE) and **new** File(name,READ). However, PolySynth also checks run-time method contracts (pre- and post-conditions) and verifies whether each of the returned snippets passes the test cases with them. Because of postconditions requiring that the file is open for reading, PolySynth discards the snippet **new** File(name,WRITE) and returns only **new** File(name,READ).

2.2.6 PolySynth: Applying user preferences

The last example demonstrates one way in which a developer can influence the ranking of the returned solutions. We consider the following functionality for managing calendar events.

```
private val events:List[Event] = List.empty[Event]
def reserve(user:User, date:Date):Event = { ... }
def getEvent(user:User, date:Date):Event = { ... }
def remove(user:User, date:Date):Event = ▮
```

Assume that a user wishes to obtain a code snippet for remove. In general, PolySynth ranks the results based on the weight function. We have found that the default computation of the weight is often adequate. Running the above example returns reserve(user,date) and getEvent(user,date), in this order. If this order is not the preferred one, the developer can modify it using elements of text search. To do so, the developer supplies a list of suggested strings indicating the names of some of the methods expected to appear in the code snippet.

For example, if the developer invokes PolySynth with "getEvent" as a suggestion, the ranking of returned snippets changes, and getEvent(user, date) appears first in the list.

In the sections that follow we will describe in more detail the techniques we embed in InSynth and PolySynth and the evaluation results they achieve. From Section 2.3 to Section 2.7 we present InSynth and in Sections 2.8 and 2.9 we present InSynth in more detail. Finally, in Section 2.10 we discuss related work and we conclude with Section 2.11.

2.3 Type Inhabitation Problem for Succinct Types

To answer whether there exists a code snippet of a given type, our starting point is the *type inhabitation problem*. In this section we establish a connection between type inhabitation and the synthesis of code snippets.

Let T be a set of types. A *type environment* Γ is a finite set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of pairs of the form $x_i : \tau_i$, where x_i is a variable of a type $\tau_i \in T$. We call the pair $x_i : \tau_i$ a *type declaration*.

The type judgment, denoted by $\Gamma \vdash e : \tau$, states that from the environment Γ, we can derive the type declaration $e : \tau$ by applying rules of some calculus. The type inhabitation problem for a given calculus is defined as follows: given a type τ and a type environment Γ, does there exist an expression e such that $\Gamma \vdash e : \tau$?

In the sequel we first describe type rules for the standard lambda calculus restricted to normal-form terms. We denote the corresponding type judgment relation $\vdash_\lambda$. We then introduce a new *succinct* representation of types and terms, with the corresponding type judgment relation $\vdash_c$.

2.3.1 Simply Typed Lambda Calculus for Deriving Terms in Long Normal Form

As background we present relevant rules for the simply typed lambda calculus, focusing on terms in long normal form (LNF). Let B be a set of basic types. Types are formed according to the following syntax:

$$\tau ::= \tau \to \tau \mid \upsilon, \quad \text{where } \upsilon \in B$$

We denote the set of all types as $\tau_\lambda(B)$.

Let V be a set of typed variables. Typed expressions are constructed according to the following syntax:

$$e ::= x \mid \lambda x.e \mid e\, e, \quad \text{where } x \in V$$

Figure 2.2 shows the type derivation rules used to derive terms in long normal form. This calculus is slightly more restrictive than the standard lambda calculus: the App rule requires that only those functions present in the original environment Γ_o can be applied on terms.

$$\textsc{App}\ \frac{(f:\tau_1 \to \ldots \to \tau_n \to \tau) \in \Gamma_o \qquad \Gamma_o \vdash_\lambda e_i:\tau_i,\ i=1..n \qquad \tau \in B}{\Gamma_o \vdash_\lambda f\, e_1 \ldots e_n : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma_o \cup \{x_1:\tau_1,\ldots,x_m:\tau_n\} \vdash_\lambda e:\tau \qquad \tau \in B}{\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_m.e : \tau_1 \to \ldots \to \tau_m \to \tau}$$

Figure 2.2: Rules for deriving lambda terms in long normal form

Definition 2.3.1 (Long Normal Form) *A judgement $\Gamma_o \vdash_\lambda e : \tau_e$ is in long normal form if the following holds:*

- $e \equiv \lambda x_1 \ldots x_m.f\, e_1 \ldots e_n$, *where* $m, n \geq 0$
- $(f : \rho_1 \to \ldots \to \rho_n \to \tau) \in \Gamma_o$, *where* $\tau \in B$
- $\tau_e \equiv \tau_1 \to \ldots \to \tau_m \to \tau$
- $\Gamma'_o \vdash_\lambda e_i : \rho_i$ *are in long normal form, where* $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_m : \tau_m\}$

Note that m can be zero. Then, $\tau_e \equiv \tau$ and Definition 2.3.1 reduces to the App rule. Otherwise, if $M \equiv f\, e_1 \ldots e_n$, then $M : \tau$ can be derived by App and $\lambda x_1 \ldots x_m.M : \tau_e$ by Abs rule.

In long normal form a variable f is followed by exactly the same number of sub-terms as the number of arguments indicated by the type of f. As an illustration, consider $f : \tau_1 \to \tau_2 \to \tau_3$ and $x : \tau_1$. There is no derivation resulting in a judgement $\Gamma_o \vdash_\lambda f\, x : \tau_2 \to \tau_3$ in long normal form, but $\lambda y.f\, x\, y : \tau_2 \to \tau_3$ has a long normal form derivation.

When solving the type inhabitation problem it suffices to derive only terms in long normal form, which restricts the search space. This does not affect the completeness of search, because each simply-typed term can be converted to its long normal form [33].

We define the depth $\mathscr{D}$ of a term from a long normal form judgement as follows:

$$\begin{aligned} \mathscr{D}(\lambda x_1 \ldots x_m.a) &= 1 \\ \mathscr{D}(\lambda x_1 \ldots x_m.f\, e_1, \ldots, e_n) &= \max(\mathscr{D}(e_1), \ldots, \mathscr{D}(e_n)) + 1 \end{aligned}$$

where a and f belong to V.

2.3.2 Succinct Types

To make the search more efficient we introduce *succinct types*, which are types modulo isomorphisms of products and currying, that is, according to the Curry-Howard correspondence, modulo commutativity, associativity, and idempotence of the intuitionistic conjunction.

Definition 2.3.2 (Succinct Types) *Let B be a set containing basic types. Succinct types t_s are constructed according to the grammar:*

$$t_s ::= \{t_s, \ldots, t_s\} \to v, \quad \text{where } v \in B$$

We denote the set of all succinct types with $t_s(B)$, sometimes also only with t_s.

A type declaration $f : \{t_1, \ldots, t_n\} \to t$ is a type declaration for a function that takes arguments of n different types and returns a value of type t. The type $\emptyset \to t$ plays a special role: it is a type of a function that takes no arguments and returns a value of type t, i.e. we consider types t and $\emptyset \to t$ equivalent.

Every type $\tau \in \tau_\lambda(B)$ can be converted into a succinct type in $t_s(B)$. With $\sigma : \tau_\lambda(B) \to t_s(B)$ we denote this conversion function. Every basic type $v \in B$ becomes an element of the set of basic succinct types, and $\sigma(v) = \emptyset \to v$. We also denote $\emptyset \to v$ only with v. Let A (arguments) and R (return type) be two functions defined on $t_s(B)$ as follows:

$$A(\{t_1, \ldots, t_n\} \to v) = \{t_1, \ldots, t_n\}$$

$$R(\{t_1, \ldots, t_n\} \to v) = v$$

Using A and R we define the σ function as follows:

$$\sigma(\tau_1 \to \tau_2) = \{\sigma(\tau_1)\} \cup A(\sigma(\tau_2)) \to R(\sigma(\tau_2))$$

In particular, for $v \in B$, a type of the form

$$\tau_1 \to \ldots \to \tau_n \to v$$

which often occurs in practice, has the succinct representation

$$\{\sigma(\tau_1), \ldots, \sigma(\tau_n)\} \to v$$

Given a type environment $\Gamma_o = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ where τ_i are types in the simply type lambda calculus, we define

$$\Gamma = \sigma(\Gamma_o) = \{\sigma(\tau_1), \ldots, \sigma(\tau_n)\}$$

It follows immediately that the conversion distributes over unions:

$$\sigma(\bigcup_{i \in I} \Gamma_o^i) = \bigcup_{i \in I} \sigma(\Gamma_o^i)$$

To demonstrate the power of the succinct representation, we provide the statistics from the example in Figure 2.1. In this example, the original type environment with 3356 declarations is reduced to the compact succinct environment with 1783 succinct types, after the σ

transformation. This drastically reduces the search space later explored by our main algorithm.

2.3.3 Succinct Patterns

Succinct patterns have the following structure:

$$\Gamma@\{t_1, \ldots, t_n\} : t$$

where $t_i \in t_s(B)$, $i = 1..n$, and $t \in B$.

A pattern $\Gamma@\{t_1, \ldots, t_n\} : t$ indicates that types $t_1, \ldots, t_n$ are inhabited in Γ and an inhabitant of type t can be computed from them also in Γ. They abstractly represent an application term in lambda calculus.

Our algorithm for finding all type inhabitants works in two phases. In the first phase we derive all succinct patterns. They can be seen as a generalization of terms, because they describe all the ways in which a term can be computed. In the second phase we do a term reconstruction based on the original type declarations (Γ_o) and the set of succinct patterns.

2.3.4 Succinct Calculus

Figure 2.3 describes the calculus for succinct types. Note that the patterns are derived only in the App rule. The rule Abs modifies Γ – it can either reduce Γ or enlarge it, depending on whether we are doing backward or forward reasoning.

$$\textsc{App}\ \frac{\{t_1, \ldots, t_n\} \to t \in \Gamma \qquad \Gamma \vdash_c t_i,\ i = 1..n \qquad t \in B}{\Gamma \vdash_c \Gamma@\{t_1, \ldots, t_n\} : t} \qquad \textsc{Abs}\ \frac{\Gamma \cup S \vdash_c (\Gamma \cup S)@\pi : t \qquad t \in B}{\Gamma \vdash_c S \to t}$$

Figure 2.3: Calculus rules for deriving succinct patterns

Consider the example given at the beginning of this section and its type environment $\Gamma_o = \{a : \text{Int}, f : \text{Int} \to \text{Int} \to \text{Int} \to \text{String}\}$. From the type environment Γ_o we compute $\Gamma = \{\emptyset \to \text{Int}, \{\emptyset \to \text{Int}\} \to \text{String}\} = \{\text{Int}, \{\text{Int}\} \to \text{String}\}$. By applying the APP rule on Int, we derive a succinct pattern $\Gamma@\emptyset : \text{Int}$ that we add to a set of derived patterns. Having a pattern for Int we apply the Abs rule. By setting $S = \emptyset$, we derive $\Gamma \vdash_c \emptyset \to \text{Int}$. Finally, by applying again the APP rule, we directly derive a pattern $\Gamma@\{\text{Int}\} : \text{String}$, for the String type and store it into the set of derived patterns.

2.3.5 Soundness and Completeness of Succinct Calculus

In this section we show that the calculus in Figure 2.3 is sound and complete with respect to synthesis of lambda terms in long normal form.

We are interested in generating any desired number of expressions of a given type without missing any expressions equivalent up to β reduction. To formulate a completeness that captures this ability, we introduce two functions, CL and RCN, shown in Figure 2.4. These functions describe the terms in long normal form of a desired type, up to a given depth d. They refer directly to $\vdash_c$ and are therefore not meant as algorithms, but as a way of expressing the completeness of succinct representation and as specifications for the algorithms we outline in Section 2.5.

```
fun CL(Γ, S→t) = {(Γ ∪ S)@S1 : t | (S1→t) ∈ (Γ ∪ S),   ∀t' ∈ S1.Γ ∪ S ⊢c t'}
fun Select(Γo, t) := {v:τ | v:τ ∈ Γo and σ(τ) = t}
fun RCN(Γo, τ1→···→τn→ν, d) :=
  if (d = 0) return ∅
  else
    S→ν := σ(τ1→···→τn→ν)
    Γ := σ(Γo)
    Γ'o := Γo ∪ {x1 : τ1, ..., xn : τn}                    //x1,...,xn are fresh
    TERMS := ∅
    foreach ((Γ ∪ S)@{t1, ..., tm'} : ν) ∈ CL(Γ, S→ν)
      foreach (f : τ) ∈ Select(Γ'o, {t1, ..., tm'}→ν)
        (ρ1→···→ρm→ν) := τ
        if (m=0) TERMS := TERMS ∪ {λx1...xn.f}
        else
          foreach i ← [1..m]
            Ti := RCN(Γ'o, ρi, d−1)
          foreach (e1,...,em) ← (T1 × ··· × Tm)
            TERMS := TERMS ∪ {λx1...xn.f e1...em}
    return TERMS
```

Figure 2.4: The function RCN constructs lambda terms in long normal form up to given depth d, invoking the auxiliary functions CL and Select.

The CL function in Figure 2.4 takes as arguments a succinct type environment Γ and a succinct type $S\to t$. It returns the set of all patterns $(\Gamma \cup S)@S_1 : t$ that describe the derivation of t. The function RCN uses the initial environment and the desired type to reconstruct lambda terms. Additionally, RCN takes a non-negative integer d to limit the reconstruction to terms with depth smaller or equal to d. It uses type $\tau_1\to\cdots\to\tau_n\to\nu$ to extend the environment and find all patterns that witness inhabitation of ν. We extend the environment with fresh variables $x_1 : \tau_1, \ldots, x_n : \tau_n$, and use CL to find the patterns. Further, we find all declarations f with a return type ν in the extended environment. If f has a function type $\rho_1\to\cdots\to\rho_m\to\nu$, we recursively generate corresponding sub-terms with types $\rho_1, \ldots, \rho_m$. Finally, we use $x_1, \ldots, x_n$, f and sub-terms to construct terms in long normal form.

Given the functions CL and RCN we can formalize the completeness theorem: each judgement in long normal form derived in the standard lambda calculus can also be derived by reconstruction using derivations (patterns) of the succinct calculus.

Theorem 2.3.3 (Soundness and Completeness) *Let Γ_o be an original environment, e an lambda expression, $\tau \in \tau_\lambda(B)$ and functions* RCN *and $\mathscr{D}$ defined as above, then:*

$$\Gamma_o \vdash_\lambda e : \tau \Leftrightarrow e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$$

We provide the proof of Theorem 2.3.3 in Appendix A.1.

2.4 Quantitative Type Inhabitation Problem

When answering the question of the type inhabitation problem, there might be many terms having the required type τ. A question that naturally arises is how to find the "best" term, for some adequate meaning of "best". For this purpose we assign a weight to every term. As in resolution-based theorem proving, a lower weight indicates a higher relevance of the term. Using weights we extend the type inhabitation problem to the *quantitative type inhabitation problem* – given a type environment Γ, a type τ and a weight function w, is τ inhabited and if it is, return a term that has the lowest weight.

Nature of Declaration or Literal	Weight
Lambda	1
Local	5
Coercion	10
Class	20
Package	25
Literal	200
Imported	$215 + \frac{785}{1+f(x)}$

Table 2.1: Weights for names appearing in declarations. We found these values to work well in practice, but the quality of results is not highly sensitive to the precise values of parameters.

Let w be a function that assigns a weight (a non-negative number) to each symbol primarily determined by:

1. the proximity to the point at which InSynth is invoked. We assume that the user prefers a code snippet composed from values and methods defined closer to the program point and assign the lower weight to the symbols which are declared closer. As shown in Table 2.1, we assign the lowest weight to local symbols declared in the same method. We assign a higher weight to symbols defined in the class where the query is initiated and the highest weight to symbols that are only in the same package.

2. the frequency with which the symbol appears in the training data corpus, as described in Section 2.7.3. For an imported symbol x, we determine its weight using the formula in Table 2.1. Here $f(x)$ is the number of occurrences of x in the corpus.

We also assign a low weight to a conversion function that witnesses the subtyping relation, as explained in Section 2.6. While we believe that our strategy is fairly reasonable, we arrived at the particular constants via trial and error, so further improvements are likely possible.

The function w also assigns a weight to a term such that the weight of $\lambda x_1 \dots x_m . f\, e_1 \dots e_n$ is equal to the sum of weights of all elements that occur in the expression:

$$w(\lambda x_1 \dots x_m . f\, e_1 \dots e_n) = \sum_{i=1}^{m} w(x_i) + w(f) + \sum_{i=1}^{n} w(e_i)$$

We use the weight of succinct types to guide the algorithm in Figure 2.5. Given Select in Figure 2.4, the weight of a succinct type t in Γ_o is defined as:

$$w(t, \Gamma_o) = min(\{w(f) \mid (f : \tau) \in \mathsf{Select}(\Gamma_o, t)\})$$

Defined this way, w has two important properties. The first property is that local variables and declarations that are defined in a user's project have higher priority than API declarations. The second property is that strictly non-negative weights and a weight of an expression, that is equal to the sum of its components' weights, allow us to build an algorithm that reaches and constructs every possible expression, given enough time.

2.5 Synthesis of All Terms in Long Normal Form

In this section we first motivate and introduce the backward search as the core mechanism of the algorithm, then we illustrate the algorithm and optimizations we implement in InSynth.

2.5.1 Backward Search

If we were to apply the rules in Figure 2.3 in a forward manner we could have started from any environment in the premise(s). However, there are infinitely many such environments. Moreover, rule Abs states that we can split Γ' in $3^{|\Gamma'|}$ possible ways into two subsets S and Γ, such that $\Gamma' = \Gamma \cup S$. However, only some environments and splittings will lead to the final conclusion $\Gamma_{init} \vdash_c S_{init} \rightarrow t_{init}$, where Γ_{init} and $S_{init} \rightarrow t_{init}$ are the initial environment and the desired type, respectively. This means we would have many unnecessary guesses and computations, leading to the wrong conclusions.

In contrast, if we use a backward search, then we start from the conclusion $\Gamma_{init} \vdash_c S_{init} \rightarrow t_{init}$ in Abs rule, and use a premise to create a hypothesis that a pattern $(\Gamma_{init} \cup S_{init})@\pi : t_{init}$ is derivable. Further, we need to check that it is indeed derivable by applying App rule. Now, unlike in the forward manner, thanks to the conclusion in App, we know the exact environment and the type t of the first premise. Selecting only types in Γ that have return types t introduce constraints on the other premises as well. The entire process is applied recursively until the initial hypothesis is proven or disproved. However, the constraints allow us to narrow the

search. This is the main advantage of the backward search. All this suggest that the backward search is more efficient, revealing only the search space reachable from the initial environment and the desired type, unlike the forward search.

To formalize the backward search we reformulate the earlier rules by splitting them into the five new rules shown in Figures 2.6 and 2.8. One should read and apply the new rules in the forward manner. In the following subsections we explain those rules in more detail.

2.5.2 Main Algorithm

In this section we present an algorithm based on the succinct ground calculus that we use for finding type inhabitants. This algorithm is further used as an interactive tool for synthesizing expression suggestions from which a developer can select a suitable expression. To be applicable, such an algorithm needs to 1) generate multiple solutions, and 2) rank these solutions to maximize the chances of returning relevant expressions to the developer.

The algorithm is illustrated in Figure 2.5. As input Synthesize takes a desired type τ_o, and an environment Γ_o and outputs at most N terms in long normal form with a type τ_o. We first transform Γ_o and τ_o by σ into a succinct environment and a type, respectively. Then we execute the algorithm in three phases. First, Explore takes the succinct type and the environment as input, and returns the discovered search space reachable from the desired type and the initial environment. Next, GenerateP takes the space as input and outputs a set of patterns. Finally, GenerateT takes patterns, Γ_o, τ_o and the integer N, and produces at most N ranked terms.

The Explore and GenerateP use succinct types to prune the search space in a light way. They leave only a portion where declaration argument-return types conform. This helps GenerateT to perform heavy reconstruct only when needed, returning compilable terms.

```
fun Synthesize(Γ_o, τ_o, N):= {
  space := Explore(σ(Γ_o), σ(τ_o))
  patterns := GenerateP(space)
  return GenerateT(patterns, Γ_o, τ_o, N)
}
```

Figure 2.5: The algorithm that generates all terms with a given type τ_o and an environment Γ_o

The algorithm Synthesize represents the imperative description of RCN, in Figure 2.4. It is the RCN version with a bound on the number of terms, N. Moreover, it uses weights to steer the search towards useful terms. We discuss this at the end of the section. The backward search and search driven by weights make the new algorithm effective, practical and interactive. Synthesize produces the same set of solutions as RCN, given the same input, if we remove bounds d and N or gradually increase them. Note that the set of solutions might be infinite.

2.5.3 Exploration phase

The goal of Explore is to start from the desired succinct type and environment and gradually explore the search space. We split the algorithm into three key steps:

1. **Type reachability.** Given a succinct type t and Γ we want to find all types reachable from t. We specify this request by $t \underset{\Gamma}{\rightsquigarrow}?$. We use the request to trigger the Match rule in Figure 2.3. By the rule, types in set S are reachable from type t, $A(t) = \emptyset$, in Γ, if $(S \to t) \in \Gamma$ holds. We denote this with *reachability* term, $t \underset{\Gamma}{\rightsquigarrow} (S, \Pi)$ (later we explain what the set Π is).

2. **Request propagation.** Once we discover that types S are reachable from t, we want to discover what types are reachable from any type $t' \in S$. Thus, we generate a new request $t' \underset{\Gamma}{\rightsquigarrow}?$. New requests are issued with the Prop rule. In other words, we use the Prop rule to propagate the search.

3. **Environment extension.** However, t can be a function type, i.e., $t \equiv S' \to t'$ and $S' \neq \emptyset$. Thus, we introduce the Strip rule that transforms a request $(S \to t) \underset{\Gamma}{\rightsquigarrow}?$ to a request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$. Now, we can further use the request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$ in the Match rule.

$$\textsc{Match}\ \frac{t \underset{\Gamma}{\rightsquigarrow}? \qquad (S \to t) \in \Gamma \qquad A(t) = \emptyset}{t \underset{\Gamma}{\rightsquigarrow} (S, \emptyset)}$$

$$\textsc{Prop}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (S, \emptyset) \qquad t' \in S}{t' \underset{\Gamma}{\rightsquigarrow}?} \qquad \textsc{Strip}\ \frac{(S \to t) \underset{\Gamma}{\rightsquigarrow}?}{t \underset{\Gamma \cup S}{\rightsquigarrow}?}$$

Figure 2.6: Type reachability rules.

A set of reachability terms keep the information about the explored search space. Thus our goal is to derive all such terms starting from the desired type and the environment. We next give the detailed description of the Explore algorithm.

To initiate the Explore algorithm in Figure 2.7 we create the request $(S \to t) \underset{\Gamma}{\rightsquigarrow}?$, where $S \to t$ is the desired type, and Γ is the initial type environment. We put the request into a working queue. In the loop we process one request at the time, until queue is empty. First, we use Strip to obtain a new request $t' \underset{\Gamma'}{\rightsquigarrow}?$ with an extended environment Γ'. Here, Strip is the function that implements the corresponding rule, Strip in Figure 2.3. It takes request $(S \to t) \underset{\Gamma}{\rightsquigarrow}?$ and returns new request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$. In the same fashion, we implement the other two functions, Match that returns a reachability term, and Prop that returns a request. Next, by applying Match to $t' \underset{\Gamma'}{\rightsquigarrow}?$ and every type in Γ' we find a set of reachability terms, found. We store the terms in the set space. The space set represents the entire search space discovered from the desired type and

```
fun Explore(Γ, S→t) := {
  queue := {S→t ⇝Γ ?}
  visited := ∅
  space := ∅
  while(queue ≠ ∅) {
    curr := queue.dequeue
    visited := visited ∪ {curr}
    t' ⇝Γ' ? := Strip(curr)
    found := {Match(t' ⇝Γ' ?, S'→t') | S'→t' ∈ Γ'}
    space := space ∪ found
    newr := {Prop(t_f ⇝Γ_f (S_f, ∅), t') | t_f ⇝Γ_f (S_f, ∅) ∈ found and t' ∈ S_f}
    queue := queue ∪ (newr \ visited)
  }
  return space
}
```

Figure 2.7: The algorithm that explores the search space.

the initial environment. Finally, we propagate the search by issuing newr requests using the Prop function onto found. We update queue with these requests. We additionally keep the set of all visited requests, to avoid cycles in the exploration.

2.5.4 Pattern generation phase

In this phase we use the space explored by the Explore algorithm to create patterns. We start from the reachability terms with inhabited types, and use them to produce patterns and new inhabited types. We repeat the process until no new types can be inhabited.

$$\textsc{Prod}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)}{\Gamma @ \Pi : t}$$

$$\textsc{Transfer}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S' \to t'\}, \Pi) \qquad t' \underset{\Gamma \cup S'}{\rightsquigarrow} (\emptyset, \Pi')}{t \underset{\Gamma}{\rightsquigarrow} (S, \Pi \cup \{S' \to t'\})}$$

Figure 2.8: Pattern synthesis rules.

Initially, we divide the search space, space, into two groups: 1) leaves that contains reachability terms in the form $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$, i.e., reachability terms with inhabited types, and 2) others that contains the remaining terms. The set Π collects succinct types that have an inhabitant. It is initialized by Match to an empty set. The Transfer rule, in Figure 2.8, turns $t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S' \to t'\}, \Pi)$

```
fun GenerateP(space) := {
  patterns := ∅
  visited := ∅
  leaves := {x | x = t ⇝_Γ (∅,∅) and x ∈ space}
  others := space \ leaves
  while (leaves ≠ ∅) {
    t ⇝_Γ (∅,Π) := leaves.dequeue
    visited := visited ∪ {t ⇝_Γ (∅,Π)}
    patterns := patterns ∪ {Prod(t ⇝_Γ (∅,Π))}
    compatible := {x | x = t' ⇝_Γ' (S∪{S' → t},Π') and Γ = Γ'∪S' and x ∈ others}
    newt := {Transfer(x, t ⇝_Γ (∅,Π)) | x ∈ compatible}
    newLeaves := {x | x = t' ⇝_Γ' (∅,Π') and x ∈ newt}
    others := (others \ compatible) ∪ (newt \ newLeaves)
    leaves := leaves ∪ (newLeaves \ visited)
  }
  return patterns
}
```

Figure 2.9: The algorithm that generates patterns.

into $t \underset{\Gamma}{\rightsquigarrow} (S, \Pi \cup \{S' \to t'\})$ if $\{S' \to t'\}$ is inhabited. We use Π to produce a pattern by the Prod rule.

In the loop we remove one term $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$ from leaves to generate: 1) a pattern $\Gamma@\Pi : t$ by the Prod function, and 2) the newt reachability terms by the Transfer function. To perform the latter we first calculate the set of compatible terms. Those are the reachability terms in form $t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S' \to t'\}, \Pi)$, such that $\Gamma = \Gamma' \cup S'$ holds. Every term in compatible can be resolved with $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$ by the Transfer rule. The result is the set of newt terms. These reachability terms can be split into two groups. The first group contains terms of the form $t' \underset{\Gamma'}{\rightsquigarrow} (\emptyset, \Pi')$, that we add to leaves. The second group contains the remaining terms and we add them to others. We also keep the set of visited leaves in order to avoid cycles in generation.

2.5.5 Term generation phase

In Figure 2.10 we illustrate the algorithm that finds at most N lambda expressions with the smallest weight. First, we introduce the notion of holes to define the partial expressions, and later we describe the algorithm GenerateT.

A typed hole $[\,]_h : \tau$ is a constant $[\,]$ with a name h and a type τ. Let V be a set of typed variables, and H a set of typed holes. Partial typed expressions are constructed according to the following syntax:

$$e ::= x \mid [\,]_h : \tau \mid \lambda x{:}\tau.e \mid e\,e, \quad \text{where } x \in V \text{ and } [\,]_h : \tau \in H$$

To derive the partial expressions in *long normal form* one can use the same APP and ABS rules in Figure 2.2, where e_i, $i = [1..m]$ and e are partial types expressions. Moreover, one can substitute all holes in a partial expression and get a new partial or complete expression, without holes. A hole $[\,]_h : \tau$, in a judgment $\Gamma_o \vdash [\,]_h : \tau$, can be substituted only with a partial expression $e : \tau$, where $\Gamma_o \vdash e : \tau$.

We next describe the GenerateT algorithm. We start from the desired type and original environment, follow patterns and gradually create and unfold partial expressions. During the process we keep partial expressions in the queue. Once a partial expression becomes complete, we store it in the set of snippets.

We use a priority queue to process partial expressions. The expressions are sorted by the weight in ascending order. We initiate the queue with $[\,]_x : \tau_{init}$, where τ_{init} is the desired type. In the loop we process one partial expression at a time. The loop terminates either when the queue is empty or we find N expression. First, we remove the highest ranked partial expression, expr_p, from the priority queue. Then, we call the function findFirstHole, that for a given judgment $\Gamma_{init} \vdash \mathsf{expr}_p$ finds (if it exists) a hole $[\,]_h : \tau_1 \to \cdots \to \tau_n \to \upsilon$ and its corresponding environment Γ_o. If expr_p has no holes, it is a complete lambda expression, i.e., a snippet that we will output to a user. Hence, we append it to snippets. If there is a hole in expr_p we build all partial expressions that can substitute the hole. We extend the environment and use patterns with the return type υ to find declarations f. If the declaration has function type, we build the expression filling all arguments with fresh holes. (Note that the new holes might be substituted in a later iteration.) For each expression expr_{newp}, we build a substitution that maps a name of the hole to the expressions. We apply the substitution to substitute the hole with the new expression. We use the function w to calculate expression weights and store them in the priority queue. The weight of a hole is equal to zero. We find the partial expressions that replace the hole using patterns. First we calculate a succinct type $S \to \upsilon$ and environment Γ. We expand the environment to $\Gamma \cup S$ and use it with type υ to find all patterns with form $(\Gamma \cup S)@S' : \upsilon$ in the pattern set. When we find all such sets S' we use them to select all type variables in Γ'_o whose type maps to a succinct type $S' \to \upsilon$. Once we have such a variable we use it to create the most general partial expression $\lambda x_1 \ldots x_n . f\,[\,]_{r_1} : \rho_1 \ldots [\,]_{r_m} : \rho_m$. Such an expression has holes at the places of f's arguments. In this way we gradually unfold a partial expression until it becomes complete.

2.5.6 Responsiveness

We use first two phases to synthesize patterns starting from the desired type and the initial environment. We referred to those phases as a $prover$. To be interactive we allow a user to specify a time limit for the prover. Due to time bound, we decide to interleave the two phases, such that whenever Explore discovers a new leaf, it immediately triggers GenerateP. Every time GenerateP is called it uses all discovered reachability terms to generate as many new patterns as possible. Moreover, to generate the best solutions, within a given time, we use a priority

```
fun GenerateT(patterns, Γ_init, τ_init, N) := {
  snippets := NIL
  pq := PriorityQueue.empty
  pq.put(0, [ ]_x : τ_init)
  while (pq.size > 0 and |snippets| < N){
    expr_p = pq.dequeue
    findFirstHole(Γ_init,expr_p) match {
      case None ⇒
        snippets.append(expr_p) //appends to the end
      case Some((Γ_o, [ ]_h : τ_1→⋯→τ_n→ν)) ⇒
        S→ν := σ(τ_1→⋯→τ_n→ν)
        Γ := σ(Γ_o)
        //x_1,...,x_n are fresh
        Γ'_o := Γ_o ∪ {x_1 : τ_1,...,x_n : τ_n}
        foreach ((Γ ∪ S)@S' : ν) ∈ patterns //S' is binder
         foreach (f : ρ_1→⋯→ρ_m→ν) ∈ Select(Γ'_o, S'→ν)
          expr_newp :=
           sub(expr_p, h ↣ (λx_1...x_n.f[ ]_r1 : ρ_1...[ ]_rm : ρ_m))
          //r_1,...,r_m are fresh names
          pq.put(w(expr_newp), expr_newp)
    }
  }
  return snippets
}

fun findFirstHole(Γ_o, exp):= exp match {
   case [ ]_x : τ ⇒ Some((Γ_o, [ ]_x : τ))
   case λx_1...x_n.f e_1...e_m ⇒
     Γ'_o := Γ_o ∪ {x_1 : τ_1,...,x_n : τ_n} //x_1,...,x_n are fresh
     for(i ∈ [1..m])
       findFirstHole(Γ'_o, e_i) match {
         case Some(hole) ⇒ return Some(hole)
         case None ⇒
       }
     None
 }

 fun sub(expr_1, y ↣ expr_2):= expr_1 match {
  case [ ]_x : τ ⇒ if (x = y) expr_2 else expr_1
  case λx_1...x_n.f e_1...e_m ⇒
     λx_1...x_n.f sub(e_1, y ↣ exp_2) ... sub(e_m, y ↣ exp_2)
 }
```

Figure 2.10: A function that constructs the best N lambda terms in long normal form.

queue in Explore instead of the regular queue. Requests in the priority queue are sorted by weights. A weight of a request $t \underset{\Gamma}{\rightsquigarrow}?$ is equal to a weight of type t in the initial environment. Additionally, we allow a user to specify a time limit for GenerateT.

2.5.7 Optimizations

To efficiently find the compatible set in GenerateP, we create a backward map that maps a term to its predecessor terms. Last reachability term_f that initiated creation of term, through propagation, is the predecessor of term. We build the map in Explore, that records all predecessors of a given term, by storing an entry (term, predecessors). By using the map, compatible becomes the predecessors set of $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$. This way we do not preform expensive calculation of compatible. However, whenever a new term x is generate by Transfer(y, z) in GenerateP, we need to update the map by substituting every occurrence of y with x in the map. To speed up this process, for every term in the map we keep the list of entries where the term occurs.

2.6 Subtyping using Coercion Functions

We use a simple method of coercion functions [17, 64, 81] to extend our approach to deal with subtyping. We found that this method works well in practice. On the given set of basic types, we model each subtyping relation $\nu_1 <: \nu_2$ by introducing into the environment a fresh coercion expression $c_{12} : \{\nu_1\} \to \nu_2$. If there is an expression $e : \tau$, and e was generated using the coercion functions, then while translating e into simply typed lambda terms, the coercion is removed. Up to η-conversion, this approach generates all terms of the desired type in a system with subtyping on primitive types with the usual subtyping rules on function types.

In the standard lambda calculus there are three additional rules to handle subtyping: transitivity ($\tau_1 <: \tau_2$ and $\tau_2 <: \tau_3$ imply $\tau_1 <: \tau_3$), subsumption (if $e : \tau_1$ and $\tau_1 <: \tau_2$ then $e : \tau_2$), and the cvariant rule ($\tau_1 <: \rho_1$ and $\rho_2 <: \tau_2$ imply $\rho_1 \to \rho_2 <: \tau_1 \to \tau_2$). We proved that even with those new rules the complexity of the problem does not change and the type inhabitation remains a PSPACE-complete problem. If subtyping constraints are present, then the coercion functions are used in the construction of succinct patterns. However, in the RCN function the coercion functions are omitted when deriving new lambda terms.

2.7 Evaluation of the Effectiveness of InSynth

This section discusses our implementation, a set of benchmarks we used to evaluate InSynth, and the experimental results.

Benchmarks		Size	#Initial	No weights		No corpus		All				Provers	
				Rank	Total	Rank	Total	Rank	Prove	Recon	Total	Imogen	fCube
1	AWTPermissionStringname	2/2	5615	>10	5157	1	101	**1**	8	125	133	127	20123
2	BufferedInputStreamFileInputStream	3/2	3364	>10	2235	1	45	**1**	7	46	53	44	5827
3	BufferedOutputStream	3/2	3367	>10	2009	1	18	**1**	7	11	19	44	5781
4	BufferedReaderFileReaderfileReader	4/2	3364	>10	2276	2	69	**1**	7	43	50	44	0176
5	BufferedReaderInputStreamReader	4/2	3364	>10	2481	2	66	**1**	7	42	49	44	0175
6	BufferedReaderReaderin	5/4	4094	>10	5185	>10	4760	**6**	7	237	244	61	0228
7	ByteArrayInputStreambytebuf	4/4	3366	>10	5146	**3**	94	>10	4	18	22	44	5836
8	ByteArrayOutputStreamintsize	2/2	3363	>10	2583	2	51	**2**	8	63	70	44	5204
9	DatagramSocket	1/1	3246	>10	5024	1	74	**1**	7	80	88	38	5555
10	DataInputStreamFileInput	3/2	3364	>10	2643	1	20	**1**	6	46	52	44	5791
11	DataOutputStreamFileOutput	3/2	3364	>10	5189	1	29	**1**	7	38	45	44	5839
12	DefaultBoundedRangeModel	1/1	6673	>10	3353	1	220	**1**	10	257	266	193	36337
13	DisplayModeintwidthintheightintbit	2/2	4999	>10	6116	1	136	**1**	6	147	154	99	10525
14	FileInputStreamFileDescriptorfdObj	2/2	3366	>10	3882	3	24	**2**	6	17	23	44	3929
15	FileInputStreamStringname	2/2	3363	>10	2870	1	125	**1**	9	100	109	44	4425
16	FileOutputStreamFilefile	2/2	3364	>10	4878	1	86	**1**	8	51	60	44	4415
17	FileReaderFilefile	2/2	3365	>10	3484	2	37	**2**	7	13	20	44	4495
18	FileStringname	2/2	3363	>10	3697	1	169	**1**	7	155	163	44	5859
19	FileWriterFilefile	2/2	3366	>10	4255	1	40	**1**	8	28	36	45	4515
20	FileWriterLPT1	2/2	3363	6	3884	1	139	**1**	7	89	96	44	4461
21	GridBagConstraints	1/1	8402	>10	3419	1	3241	**1**	19	323	342	290	0121
22	GridBagLayout	1/1	8401	>10	2	1	1	**1**	0	1	1	290	56553
23	GroupLayoutContainerhost	4/2	6436	>10	4055	1	24	**1**	10	26	36	190	29794
24	ImageIconStringfilename	2/2	8277	>10	3625	2	495	**1**	13	154	167	300	50576
25	InputStreamReaderInputStreamin	3/3	3363	>10	3558	8	90	**4**	7	177	184	44	4507

Table 2.2: Results of measuring overall effectiveness (part 1). The first 4 columns denote the ordinal and name of a benchmark, size of the desired snippet (in terms of number of declarations: with coercion function accounted/only visible) and the initial number of declarations seen at the invocation point. The subsequent columns denote the rank at which the desired snippet was found and (averaged) execution times in milliseconds for the algorithm with no weights, with weight but without use of input statistics, and with weights and input statistics (with the distribution of execution time between the engine and reconstruction parts). The last two columns show execution time for checking provability using the Imogen and fCube provers.

Benchmarks		Size	#Initial	No weights		No corpus		All				Provers	
				Rank	Total	Rank	Total	Rank	Prove	Recon	Total	Imogen	fCube
26	JButtonStringtext	2/2	6434	>10	3289	2	117	**1**	9	85	95	184	27828
27	JCheckBoxStringtext	2/2	8401	>10	3738	3	134	**2**	18	50	68	188	4946
28	JformattedTextFieldAbstractFormatter	3/2	10700	>10	3087	**2**	2048	4	21	101	122	520	99238
29	JFormattedTextFieldFormatterformatter	2/2	9783	>10	3404	2	67	**2**	15	85	100	419	74713
30	JTableObjectnameObjectdata	3/3	8280	>10	3676	2	109	**2**	13	129	142	300	46738
31	JTextAreaStringtext	2/2	6433	>10	2012	**2**	232	>10	9	293	302	183	29601
32	JToggleButtonStringtext	2/2	8277	>10	3171	2	177	**2**	12	123	135	299	5231
33	JTree	1/1	8278	2	3534	1	3162	**1**	16	2022	2039	298	52417
34	JViewport	1/1	8282	8	5017	**1**	20	8	12	7	19	298	22946
35	JWindow	1/1	6434	3	4274	1	296	**1**	10	425	434	194	2862
36	LineNumberReaderReaderin	5/4	3363	>10	2315	>10	3770	**9**	6	233	239	44	5876
37	ObjectInputStreamInputStreamin	3/2	3367	>10	3093	1	20	**1**	6	29	35	44	5849
38	ObjectOutputStreamOutputStreamout	3/2	3364	>10	4883	1	31	**1**	7	47	54	44	5438
39	PipedReaderPipedWritersrc	2/2	3364	>10	2762	2	54	**2**	8	60	68	44	262
40	PipedWriter	1/1	3359	>10	4801	1	107	**1**	6	133	139	44	5432
41	Pointintxinty	3/1	4997	>10	2068	5	133	**2**	6	96	103	101	8573
42	PrintStreamOutputStreamout	3/2	3365	>10	2100	6	16	**1**	7	20	27	44	5841
43	PrintWriterBufferedWriter	4/3	3365	>10	2521	4	135	**4**	8	36	44	44	448
44	SequenceInputStreamInputStreams	5/3	3365	>10	4777	2	35	**2**	8	20	28	44	5862
45	ServerSocketintport	2/2	4094	>10	2285	2	28	**1**	6	57	63	61	11123
46	StreamTokenizerFileReaderfileReader	3/2	3365	>10	2012	1	34	**1**	8	57	65	44	5782
47	StringReaderStrings	2/2	3363	>10	2006	1	35	**1**	6	37	43	45	5746
48	TimerintvalueActionListeneract	3/3	6665	>10	2051	1	123	**1**	10	189	199	186	34841
49	TransferHandlerStringproperty	2/2	8648	>10	3911	1	27	**1**	14	17	31	319	67997
50	URLStringspecthrows	3/3	4093	>10	3302	6	124	**1**	8	175	183	60	11197

Table 2.3: Results of measuring overall effectiveness (part 2). The first 4 columns denote the ordinal and name of a benchmark, size of the desired snippet (in terms of number of declarations: with coercion function accounted/only visible) and the initial number of declarations seen at the invocation point. The subsequent columns denote the rank at which the desired snippet was found and (averaged) execution times in milliseconds for the algorithm with no weights, with weight but without use of input statistics, and with weights and input statistics (with the distribution of execution time between the engine and reconstruction parts). The last two columns show execution time for checking provability using the Imogen and fCube provers.

2.7.1 Implementation in Eclipse

We implemented InSynth as an Eclipse plugin that extends the code completion feature. It enables developers to accomplish a complex action with only a few keystrokes: declare a type of a term, invoke InSynth, and select one of the suggested expressions.

InSynth provides its functionality in Eclipse as a contribution to the standard Eclipse content assist framework and contributes its results to the list of content assist proposals. These proposals can be returned by invoking the content assist feature when Scala source files are edited (usually with Ctrl + Space). If the code completion is invoked at any valid program point in the source code, InSynth attempts to synthesize and return code snippets of the desired type. Only the top specified number of snippets are displayed as proposals in the content assist proposal list, in the order corresponding to the weighted ranking. InSynth supports invocation at any location immediately following declaration of a typed value, variable or a method, i.e. in the place of its definition and also at the place of method parameters, if condition expressions, and similar (where the type can be inferred). InSynth uses the Scala presentation compiler to extract program declarations and imported API functions visible at a given point. InSynth can be easily configured though standard Eclipse preference pages, and the user can set maximum execution time of the synthesis process, desired number of synthesized solutions and code style of Scala snippets.

2.7.2 Creating Benchmarks

There is no standardized set of benchmarks for the problem that we examine, so we constructed our own benchmark suite. We collected examples primarily from http://www.java2s.com/. These examples illustrate correct usage of Java API functions and classes in various scenarios. We manually translated the examples from Java into equivalent Scala code. Since only single class imports are used in the original examples, we generalized the import statements for the benchmarks to include more declarations and thereby made the synthesis problem more difficult by increasing the size of the search space.

One idea of measuring the effectiveness of a synthesis tool is to estimate its ability to reconstruct certain expressions from existing code. We arbitrarily chose some expressions from the collected examples, removed them and marked them as goal expressions that needed to be synthesized (we replaced them with a fresh value definition if the place of the expression was not valid for InSynth invocation). The resulting benchmark is a partial program, similar to a program sketch [84]. We measure whether InSynth can reconstruct an expression equal to the one removed, modulo literal constants (of integer, string, or boolean type). Our benchmark suite is available for download from the InSynth web site.

2.7.3 Corpus for Computing Symbol Usage Frequencies

Our algorithm searches for typed terms that can be derived from the initial environment and that minimize the weight function. To compute initial declaration weights we follow the steps presented in Section 2.4. The key step is to derive declarations frequencies. Hence, we collected a code corpus which dictates those initial weights. The corpus contains code statistics from 18 Java and Scala open-source projects. Table 2.4 lists those projects together with their description.

Project	Description
Akka	Transactional actors
CCSTM	Software transactional memory
GooChaSca	Google Charts API for Scala
Kestrel	Tiny queue system based on starling
LiftWeb	Web framework
LiftTicket	Issue ticket system
O/R Broker	JDBC framework with support for externalized SQL
scala0.orm	O/R mapping tool
ScalaCheck	Unit test automation
Scala compiler	Compiles Scala source to Java bytecode
Scala Migrations	Database migrations
ScalaNLP	Natural language processing
ScalaQuery	Typesafe database query API
Scalaz	"Scala on steroidz" - scala extensions
simpledb-scala-binding	Bindings for Amazon's SimpleDB
smr	Map Reduce implementation
Specs	Behaviour Driven Development framework
Talking Puffin	Twitter client

Table 2.4: Scala open-source projects used for the corpus extraction.

One of the analyzed projects is the Scala compiler, which is mainly written in the Scala language itself. In addition to the projects listed in Table 2.4, we analyzed the Scala standard library, which mainly consists of wrappers around Java API calls. We extracted the relevant information only about Java and Scala APIs, and ignored information specific to the projects themselves. Overall, we extracted 7516 declarations and identified a total of 90422 uses of these declarations. 98% of declarations have less than 100 uses in the entire corpus, whereas the maximal number of occurrences of a single declaration is 5162 (for the symbol `&&`).

2.7.4 Platform for Experiments

We ran all experiments on a machine with a 3Ghz clock speed processor and 8MB of cache. We imposed a 2GB limit for allowed memory usage. Software configuration consisted of Ubuntu 12.04.1 LTS (64b) with Scala 2.9.3 (a nightly version), and Java(TM) Virtual Machine 1.6.0_24. The reconstruction part of InSynth is implemented sequentially and does not make use of multiple CPU cores.

2.7.5 Measuring Overall Effectiveness

In each benchmark, we invoked InSynth at the place where the goal expression was missing. We parametrized InSynth with N=10 and used a time limit of 0.5s seconds for *prover* (Section 2.5.6) and 7s for the reconstruction. By using a time limit, our goal was to evaluate the usability of InSynth in an interactive environment (which IDEs usually are).

We ran InSynth on the set of 50 benchmarks. Results are shown in Tables 2.2 and 2.3. The Size column represents the size of the goal expression in terms of number of declarations in its structure. It is illustrated in the form c/nc where c is the size with coercion functions and nc is the size without. Note that when c>nc holds, InSynth needs to deal with subtyping to synthesize the goal expression. The #Initial column represents the size of the initial environment, i.e. the number of initial type declarations that InSynth extracts at a given program point. The following columns are partitioned into three groups, one for each variant of the synthesis algorithm: 1) the algorithm without weights (the No weights column), 2) the algorithm with weights derived without the corpus (the No corpus column) and 3) finally, the full algorithm, with weights derived using the corpus (the All column).

In all groups, Rank represents the rank of the goal expression in the resulting list, and Total represents the total execution time of synthesis. The distribution of the execution time between *prover* and the reconstruction is shown in columns Prove and Recon, respectively. The last column group gives execution times of two state-of-the-art intuitionistic theorem provers (`Imogen` [71] and `fCube` [37]) employed for checking provability of inhabitation problems for the benchmarks.

Tables 2.2 and 2.3 show the differences in both effectiveness and execution time between the variants of the algorithm.

First, the table shows that the algorithm without weights does not perform well and finds the goal expressions in only 4 out of 50 cases and executes by more than an order of magnitude slower than the other variants. This is due to the fact that without the utilization of the weight function to guide the search, InSynth is driven into a wrong direction toward less important solutions, whose ranks are as low as the actual solutions.

Second, we can see that adding weights to terms helps the search drastically and the algorithm without corpus fails to find the goal expression in only 2 cases. Also, the running times are decreased substantially. In 33 cases, this variant finds the solution with the same rank as the variant which incorporates corpus, while on 4 of them it finds the solution of a higher rank. This suggests that in some cases, synthesis does not benefit from the derived corpus – initial weights defined by it are not biased favorably and do not direct the search toward the goal expression.

Third, we show the times for `Imogen` and `fCube` provers on the same set of benchmarks. We can see that our *prover* is up to 2 orders of magnitude faster than `Imogen` and up to 4 orders

$$\textsc{Axiom}\ \frac{x:\tau}{\Gamma \vdash x:\tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash f:\tau_1 \to \tau_2 \qquad \Gamma \vdash x:\tau_1'}{\Gamma \vdash f(x):\tau_2'} \quad \begin{array}{c}\sigma = \mathsf{mgu}(\tau_1,\tau_1') \\ \tau_2' = \sigma(\tau_2)\end{array}$$

$$\textsc{Compose}\ \frac{\Gamma \vdash f:\tau_1 \to \tau_2 \qquad \Gamma \vdash g:\tau_0 \to \tau_1'}{\Gamma \vdash (f \circ g):\tau_0' \to \tau_2'} \quad \begin{array}{c}\sigma = \mathsf{mgu}(\tau_1,\tau_1') \\ \tau_0' = \sigma(\tau_0) \\ \tau_2' = \sigma(\tau_2)\end{array}$$

Figure 2.11: Rules for Generic Types used by Our Algorithm

than `fCube`. Note that reconstruction of terms in `Imogen` was limited to 10 seconds and `Imogen` failed to reconstruct a proof within that time limit in all cases.

In the case of the full algorithm, the results show that the desired expressions appear in the top 10 suggested snippets in 48 benchmarks (96%). They appear as the top snippet (with rank 1) in 32 benchmarks (64%). Note that our corpus (Section 2.7.3) is derived from a source code base that is disjoint (and somewhat different in nature) from the one used for benchmarks. This suggests that even a knowledge corpus derived from unrelated code increases the effectiveness of the synthesis process; a specialized corpus would probably further increase the quality of results.

In summary, the expected snippets were found among the top 10 solutions in many benchmarks. Weights play an important role in finding and ranking those snippets high in a short period of time (on average around just 145ms). Finally, our *prover* outperforms two state of the art provers `Imogen` and `fCube`. These results suggest that InSynth is effective in quickly finding (i.e. synthesizing) desired expressions at various places in source code.

2.8 Quantitative Inhabitation for Generics

Unlike previous sections that describe InSynth techniques, in this section we present PolySynth that like InSynth takes a desired type as input and returns a list of ranked declarations. The main difference is that PolySynth, unlike InSynth, supports polymorphic types (generics).

We start by presenting PolySynth algorithm for type inhabitation in the presence of generic (parametric) types as in the Hindley-Milner type system, without nested type quantifiers. We represent type variables implicitly, as in resolution for logics with variables. Those types are also known under the names ML-style types or Hindley-Milner types.

Definitions:
$w(e{:}\tau) := w(e) + w(\tau)$
$\mathsf{bestT}(\tau,\Gamma) := \{(e{:}\tau) \in \Gamma \mid (\forall (e'{:}\tau) \in \Gamma.\ w(e) \leq w(e'))\}$
$\mathsf{bestT}(e'{:}\tau',\Gamma) := \mathsf{bestT}(\tau',\Gamma)$
$w(\mathsf{bestT}(b,\Gamma)) = w(b), \text{if } \exists b \in \mathsf{bestT}(b,\Gamma),\ +\infty \text{ otherwise}$
$\mathsf{best}(\mathsf{q}) := \{b \in \mathsf{q} \mid \forall b' \in \Gamma.\ w(b) \leq w(b')\}$
$\mathsf{cmpt}(\tau_1,\tau_2) :=$ an mgu **in** App or Compose rules

fun synth(Γ_0, τ_G) { // Γ_0 – *environment at program point*, τ_G – *desired type*
$\Gamma = \Gamma_0 \cup \Gamma_{\mathsf{Comb}} \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}$
$\mathsf{q} = \Gamma$
$\mathsf{res} = \emptyset$
while ($\neg$timeout $\wedge\ \mathsf{q} \neq \emptyset$) {
let $(e_1{:}\tau_1) \in \mathsf{best}(\mathsf{q})$
$\mathsf{q} = \mathsf{q} \setminus \{(e_1{:}\tau_1)\}$
foreach$((e_2{:}\tau_2) \in \{(e_2{:}\tau_2) \in \mathsf{bestT}(\tau_2,\Gamma) \mid \mathsf{cmpt}(\tau_1,\tau_2)\})$ {
$\mathsf{derived} = \mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2) \cup \mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$
$\mathsf{res} = \mathsf{res} \cup \{e' \mid (e : \bot_{fresh}) \in \mathsf{derived}, e[\mathsf{G} := \mathsf{I}] \overset{\mathsf{I}(t)\to t}{\rightsquigarrow^*} e'\}$
$\mathsf{q} = \mathsf{q} \cup \{b \in \mathsf{derived} \mid w(b) < w(\mathsf{bestT}(b,\Gamma))\}$
$\Gamma = \Gamma \cup \mathsf{derived}$
}
}
return res;
}

Figure 2.12: The Search Algorithm for Quantiative Inhabitation for Generic Types

Definition 2.8.1 (Generic Types) *Let C be a fixed finite set. For every $c \in C$, with $c/_n$ we denote the arity of the element. Let V be a set of type variables. The set of all generic types T is defined by the grammar:*

$$T_b ::= V \mid C(T_b, \ldots, Tb) \mid T_b \to T_b$$
$$T ::= T_b \mid \forall V.T$$

Figure 2.11 shows the axiom rule, the rules for application and the rule for composition (which we introduce to improve performance by making derivations shorter).

Description of the algorithm. Figure 2.12 shows the algorithm that systematically applies rules in Figure 2.11, while avoiding cycles due to repeated types whose terms have non-minimal weights. The algorithm maintains two sets of bindings (pairs of expressions and their types): Γ, which holds all initial and derived bindings, and q, which is a work list containing the bindings that still need to be processed. At the start of algorithm, Γ contains the initial declarations, as well as the goal encoded as $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$ where τ_G is the type for which the

user wishes to generate expressions. The work list initially contains all these declarations as well. The algorithm accumulates the expressions of the desired type in the set res. The main loop of the algorithm runs until the timeout is reached or the work list q becomes empty.

The body of the main loop of the algorithm selects a minimal (given by best(_)) binding $(e_1{:}\tau_1)$ from the work list q and attempts to combine it with all other bindings in Γ for which the types τ_1 and τ_2 can be unified to participate in one of the inference rules (we denote this condition using the $\mathsf{cmpt}(\tau_1,\tau_2)$ relation). Note, however, that there is no point in combining $(e_1{:}\tau_1)$ with a $(e_2{:}\tau_2)$ if there is another $(e_2'{:}\tau_2)$, with the same τ_2 but with a strictly smaller $w(e_2')$. Therefore, the algorithm restricts the choice of $(e_2{:}\tau_2)$ to those where $w(e_2)$ is minimal for a given τ_2. We formalize this using the function $\mathsf{bestT}(\tau_2,\Gamma)$ that finds a set of such bindings with minimal e_2. We also extend the function to accept a candidate e_2' (which is ignored in looking up the minimal e_2). Moreover, we define $w(\mathsf{bestT}(\tau_2,\Gamma))$ to denote the value of this minimum (if it exists).

The sets $\mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2)$ and $\mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$ are results of applying the rules from Figure 2.11. If no rule can be applied the result is the empty set. We use derived to denote the set of results of applying the inference rules to selected bindings. These results may need to be processed further and therefore the algorithm may need them into q. However, it avoids doing this if the derived binding has a type that already exists in Γ and the newly derived expression does not have a strictly smaller weight. This reduces the amount of search that the algorithm needs to perform.

Because of the declaration $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$, the algorithm detects expressions of type τ_G using the expressions e of fresh type $\bot_{fresh}$. To obtain the expression of the desired type, we replace in e every occurrence of G with the identity combinator I. This is justified because $\bot_{fresh}$ is a fresh constant, so replacing it with τ_G in a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}(e{:}\bot_{fresh})$ yields a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \tau_G)\} \vdash (e{:}\tau_G)$, in which we can use I instead of G. The algorithm also simplifies the accumulated expressions by reducing I where possible. In the presence of higher-order functions I may still remain in the expressions, which is not a problem because it is deducible from any complete set of combinators.

Finally, under the assumption that a linear weight function is given, and the weight of each expression symbol is strictly positive, it is straightforward to see that the algorithm finds the derivations for all types that can be obtained using the rules from Figure 2.11. Indeed, the weight of an expression strictly increases during the derivation, so an algorithm, if it runs long enough, reaches arbitrarily long value as the minimum of the work list. This shows that the algorithm is complete.

2.9 PolySynth Implementation and Evaluation

PolySynth is implemented in Scala and built on top of the Ensime plugin [21]. It can therefore directly use program information computed by the Scala compiler, including abstract syntax

Program	# Loaded Declarations	# Methods in Synthesized Snippets	Time [s]
FileReader	6	4	< 0.001
Map	4	4	< 0.001
FileManager	3	3	< 0.001
Calendar	7	3	< 0.001
FileWriter	320	6	0.093
SwingBorder	161	2	0.016
TcpService	89	2	< 0.001

Figure 2.13: Basic algorithm for synthesizing code snippets

trees and the inferred types. Furthermore, it can generate an appropriate pop-up window with suggested synthesized snippets and allow the user to interactively select the desired fragment. Our implementation consists of four key components: the loader that finds all visible declarations from a given point in code; the synthesis algorithm described before; and the filter that runs a user written tests and discards snippets that fail the tests. Although, the test cases can filter many irrelevant solutions, test execution is often slow and sometimes unfeasible due to several holes in the code under test.

Figure 2.13 gives an idea of the performance of the system. We ran all examples on Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM. The running times to find the first solution are usually bellow two milliseconds. Our experience suggests that the algorithm scales well. As an illustration, we were able to synthesize a snippet containing six methods in 0.093 seconds from the set of 320 declarations. Times to encode declarations into FOL formulas range from 0.015 (Calendar) to 0.046 (FileWriter) seconds. If the synthesized snippets need to use more methods from imported libraries, the synthesis typically takes longer, but is typically fast enough to be useful. The above examples and the system PolySynth are available on the following web site: http://lara.epfl.ch/w/insynth.

2.10 Related Work

Several tools including Prospector [65], XSnippet [82], Strathcona [47], PARSEWeb [88] and SNIFF [22] that generate or search for relevant code examples have been proposed. In contrast to all these tools we support expressions with higher order functions. Additionally, we synthesize snippets using all visible methods in a context, whereas the other existing tools build or present them only if they exist in a corpus. Prospector, Strathcona and PARSEWeb do not incorporate the extracted examples into the current program context; this requires additional effort on the part of the programmer. Moreover, Prospector does not solve queries with multiple argument methods unless the user initiates multiple queries. In contrast, we generate full expressions using just a single query. We could not effectively compare InSynth and PolySynth with those tools, since unfortunately, the authors did not report exact running times.

We next provide more detailed descriptions for some of the tools, and we compare their functionality to InSynth and PolySynth. InSynth and PolySynth are similar in operation to Eclipse content assist proposals [40] and it implements the same behaviour. More advanced solutions appeared recently, such as [18], that proposes declarations, and the Eclipse code recommenders [8], that suggests declarations and code templates. Both systems use API declaration call statistics from the existing code examples in order to offer suggestions to the developer with appropriate statistical confidence value. InSynth and PolySynth are fundamentally different from these approaches (they even subsume them) and can synthesize even code fragments that never previously occurred in code. Additionally, the system by Lee et al. [61] makes code completion and navigation aware of code evolution and enables them to operate on multiple versions at a time, without having to manually switch across these versions.

Prospector [65] uses a type graph and searches for the shortest path from a receiver type, $type_{in}$, to the desire type, $type_{out}$. The nodes of the graph are monomorphic types, and the edges are the names of the methods. The nodes are connected based on the method signature. Prospector also encodes subtypes and downcasts into the graph. The query is formulated through $type_{in}$ and $type_{out}$. The solution is a chain of method calls that starts at $type_{in}$ and ends at $type_{out}$. Prospector ranks solutions by the length, preferring shorter solutions. In contrast, we find solutions that have minimal weights. This potentially enables us to get solutions that have better quality, since the shortest solution may not be the most relevant. Furthermore, in order to fill in the method parameters, a user needs to initiate multiple queries in Prospector. In InSynth and PolySynth this is done automatically. Prospector uses a corpus for down-casting, whereas we use it to guide the search and rank the solutions. Moreover, Prospector has no knowledge of what methods are used most frequently. Unfortunately, we could not compare our implementation with Prospector, because it was not publicly available. XSnippet [82] offers a range of queries from generalized to specialized. The tool uses them to extract Java code from the sample repository. XSnippet ranks solutions based on their length, frequency, and context-sensitive as well as context-independent heuristics. In order to narrow the search the tool uses the parental structure of the class where the query is initiated to compare it with the parents of the classes in the corpus. The returned examples are not adjusted automatically into a context—the user needs to do this manually. Similar to Prospector the user needs to initiate additional queries to fill in the method parameters. In Strathcona [47], a query based on the structure of the code under development is automatically extracted. One cannot explicitly specify the desired type. Thus, the returned set of examples is often irrelevant. Moreover, in contrast to InSynth and PolySynth, those examples can not be fitted into the code without additional interventions. PARSEWeb [88] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. In InSynth and PolySynth the length of a returned snippet also plays an important role in ranking the snippets but InSynth and PolySynth also have an additional component by taking into account the proximity of derived snippets and the point where InSynth and PolySynth were invoked. The main idea behind the SNIFF [22] tool is to use natural language to search for code examples. The authors collected the corpus of examples and annotated them with

keywords, and attached them to corresponding method calls in the examples. The keywords are collected from the available API documentation. InSynth is based on a logical formalism, so it can overcome the gap between programming languages and natural language.

The synthesized code in our approach is extracted from the proof derivation. Similar ideas have been exploited in the context of sophisticated dependently typed languages and proof assistants [15]. Our goal is to apply it to simpler scenarios, where propositions are only partial specifications of the code, as in the current programming practice. Agda is a dependently typed programming language and proof assistant. Using Agda's Emacs interface, programs can be developed incrementally, leaving parts of the program unfinished. By type checking the unfinished program, the programmer can get useful information on how to fill in the missing parts. The Emacs interface also provides syntax highlighting and code navigation facilities. However, because it is a new language and lacks large examples, it is difficult to evaluate this functionality on larger numbers of declarations.

There are several tools for the Haskell API search. The Hoogle [7] search engine searches for a single function that has either a given type or a given name in Haskell, but it does not return a composed expression of the given type. The Hayoo [6] search engine does not use types for searching functions: its search is based on function names. The main difference between Djinn [3] and our system is that Djinn generates a Haskell expression of a given type, but unlike our system it does not use weights to guide the algorithm and rank solutions. Recently we have witnessed a renewed interest in semi-automated code completion [77]. The tool [77] generates partial expressions to help a programmer write code more easily. While their tool helps to guess the method name based on the given arguments, or it suggests arguments based on the method name, we generate complete expressions based only on the type constraints. In addition, our approach also supports higher order functions, and the returned code snippets can be arbitrarily nested and complex: there is no bound on the number and depth of arguments. This allows us to automatically synthesize larger pieces of code in practice, as our evaluation shows. In that sense, our result is a step further from simple completion to synthesis.

The use of type constraints was explored in interactive theorem provers, as well as in synthesis of code fragments. SearchIsos [31] uses type constraints to search for lemmas in Coq, but it does not use weights to guide the algorithm and rank the solutions. Having the type constraints, a natural step towards the construction of proofs is the use of the Curry-Howard isomorphism. The drawback of this approach is the lack of a mechanism that would automatically enumerate all the proofs. By representing proofs using graphs, the problem of their enumeration was shown to be theoretically solvable [97], but there is a large gap between a theoretical result and an effective tool. Furthermore, InSynth and PolySynth can not only enumerate terms but also rank them and return a desired number of best-ranked ones.

Having a witness term that a type is inhabited is a vital ingredient of our tools, so one could think of InSynth and PolySynth as provers for propositional intuitionistic logic (with substantial

additional functionality). Among recent modern provers are Imogen [71] and fCube [37]. These tools can reason about more expressive fragments of logic: they support not only implication but also intuitionistic counterparts for other propositional operators such as $\vee, \Rightarrow, \Leftrightarrow$, and Imogen also supports first-order and not only propositional fragment. Our results on fairly large benchmarks suggests that InSynth is faster for our purpose. This is not entirely surprising because these tools are not necessarily optimized for the task that we aim to solve (looking for shallow proofs from many assumptions), and often do not have efficient representation of large initial environments. The main purpose of our comparison is to show that our technique is no worse than the existing ones for our purpose, even when used to merely check the existence of proofs. What is more important than performance is that InSynth produces not only one proof, but a representation of *all* proofs, along with their ranking. This additional functionality of our algorithm is essential for the intended application: using type inhabitation as a generalization of code completion.

For a given type InSynth produces a finite representation of all the type inhabitants. In general, if an expression is an inhabitant of the given type, there is a derivation that proves that fact. Using Curry-Howard isomorphism for each proof derivation there is a lambda term representing it. The problem of enumerating all the proofs for a given formula is an important research topic, since it can be also used to answer other problems like provability or definability. We keep the system of patterns to represent all the type inhabitants, achieving this way finite representation of a possibly infinite set of the proofs. G. Dowek and Y. Jiang [34] used a semi-grammatically description of all proof-terms for minimal predicate logic and a positive sequent calculus. The use of grammars is an alternative to our use of graphs as the representation for all solutions; we therefore expect that grammars could similarly be used as the starting point for a practical system such as ours.

2.11 Conclusions

We have presented the design and implementation of InSynth, a code completion tool inspired by complete implementation of type inhabitation for the simply typed lambda calculus. InSynth algorithm uses succinct types, an efficient representation for types, terms, and environments that takes into account that the order of assumptions is unimportant. InSynth approach generates a representation of all solutions (a set of pattens), from which it can extract any desired number of solutions.

To further increase the usefulness of generated results, we introduce the ability to assign weights to terms and types. The resulting algorithm performs search for expressions of a given type in a type environment while minimizing the weight, and preserves the completeness. The presence of weights increases the quality of the generated results. To compute weights we use the proximity to the declaration point as well as weights mined from a corpus. We have deployed the algorithm in an IDE for Scala. InSynth evaluation on synthesis problems constructed from API usage indicate that the technique is practical and that several technical

ingredients had to come together to make it powerful enough to work in practice. InSynth and additional evaluation details are publicly available.

Our experience suggests that the idea of computing type inhabitats using succinct types and weights is useful by itself. Moreover, our subsequent exploration suggests that these techniques can also serve as the initial phase of semantic-based synthesis [58]. The idea is to generate a stream of type-correct solutions and then filter it to contain only expressions that meet given specifications, such as postconditions (or, in the special case, input/output examples).

Additionally, we have presented PolySynth that like InSynth takes the desire type and synthesizes a ranked list of expressions, considering declarations with polymorphic types (generics). It uses a resolution based algorithm and wights to synthesize and ranked candidate expressions. In addition, it takes test cases and user preferences to further filler undesired candidates. Our results show that PolySynth is useful for synthesizing code fragments that include declarations with polymorphic types.

Note that the approach based on the techniques we presented can also generate programs with various control patterns, because conditionals, loops, and recursion schemas can themselves be viewed as higher-order functions. Although we believe the current results to be a good starting point for such tasks, further techniques may be needed to control larger search spaces for more complex code correctness criteria and larger expected code sizes.

3 Synthesizing Code from Free-Form Queries

We present a new code assistance tool for integrated development environments. Our system accepts free-form queries allowing a mixture of English and Java as an input, and produces Java code fragments that take the query into account and respect syntax, types, and scoping rules of Java as well as statistical usage patterns. The returned results need not have the structure of any previously seen code fragment. As part of our system we have constructed a probabilistic context free grammar for Java constructs and library invocations, as well as an algorithm that uses a customized natural language processing tool chain to extract information from free-form text queries. We present the results on a number of examples showing that our technique can tolerate much of the flexibility present in natural language, and can also be used to repair incorrect Java expressions that contain useful information about the developer's intent.

3.1 Motivation

Application programming interfaces (APIs) are becoming more and more complex, presenting a bottleneck when solving simple tasks, especially for new developers. APIs contain many types and declarations, so it is difficult to know how to combine them to achieve a task of interest. Instead of focusing on more creative aspects of the development, a developer ends up spending a lot of time trying to understand informal documentation or adapt the API documentation examples. Integrated development environments (IDEs) help in this task by listing declarations that belong to a given type, but leave it to the developer to decide how to combine the declarations.

On the other hand, on-line repository host services such as GitHub [5], BitBucket [4], SourceForge [1] are becoming more and more popular, hosting a large number of freely accessible projects. Such repositories are an excellent sources of code examples that the developers can use to learn API usage. Moreover, their large size and variety suggests that they can be used by machine learning techniques to create more sophisticated IDE support. A natural first step is to perform code search [88], though this still leaves the user with the task of understanding

the context and adapting it to their needs. Several researchers have pursued the problem of generalizing from such examples in repositories, combining non-trivial program analysis and machine learning techniques [80].

In this chapter, we present a new approach that synthesizes code appropriate for a given program point, guided by hints given in *free-form* text. We have implemented our approach in a system *anyCode*, and have found it to be very useful in our experience (see Tables 3.1 and 3.2). Our approach builds a model of the Java language based on corpus of code in repositories, and adapts the model to a given text input. In that sense, our approach combines some of the advantages of statistical programming language models [80] but also of natural language processing of the input containing English phrases, previously done for restricted APIs [60].

Our approach builds on our past experience with the InSynth tool, already presented in Chapter 2, in which a user only indicates the desired API type; the tool InSynth then generates ranked expressions of a given type. With our new tool, *anyCode*, the input can be interpreted as a result type, but, more generally, it can be any text, referring to any part of an expected expression. We find this interface more convenient and expressive than in InSynth. Furthermore, InSynth uses only the unigram model [51, Chapter 4], which assigns a probability to a declaration based on its call frequency in a corpus. InSynth uses the model to synthesize and rank expressions. In *anyCode*, besides unigram, we use the more sophisticated probabilistic context free grammar (PCFG) model [51, Chapter 14], to synthesize and rank the expressions. We perform synthesis in three phases: (1) we use natural language processing (NLP) tools [29, 67, 92] to structure the input text and split the text input into chunks of words based on their relationships in the sentence parse tree; (2) we use the structured text and unigram model to select a set of most likely API declarations, using a set of scoring metrics and the Hungarian method [56] to solve the resulting assignment problem [19]; (3) we finally use the selected declarations, PCFG and unigram model to unfold the declaration arguments. The result is a list of ranked (partial) expressions, which *anyCode* offers to the developer using the familiar code completion interface of Eclipse.

By introducing a textual input interface, we aim to automatically reduce the gap between a natural and a programming language. *anyCode* allows the developer to formulate a query using a mixture of English and code fragments. *anyCode* takes into account English grammar when processing input text. To improve the input flexibility and expressiveness we also consider word synonyms and other related words (hypernyms and hyponyms). We build a related word map based on WordNet [36], a large lexical database of English. We present a technique to make WordNet usable in our context by automatically projecting it onto the API jargon. We use these techniques along with the NLP tools to support the natural language aspect in *anyCode*. The techniques we implement in *anyCode* are inspired by stochastic machine translation. However, we had to overcome the lack of a parallel corpus relating English and Java, as well as the gap between an informal medium such as English and the rigorous syntax and type rules of a programming language such as Java.

We aim to relieve the user of the strict structure of a programming language when describing their intention. From our perspective, IDE tools should allow a user to gloss over aspects such as the number and the order of arguments in method calls, or parenthesis usage. Instead, the developers should focus more on solving important higher-level software architecture and decomposition problems. Finally, we also hope to lower the entry for those who are learning to program, for whom syntax is often one of the first obstacles. To achieve this, we find that a short text input that approximately describes the structure of the desired expression is the most convenient. To make the input useful for programming, we also allow a user to explicitly write literals and local variable names in input. Using such input, *anyCode* manages to synthesize valid Java code fragments. It can do that because it does not impose any strict requirement on the input: it has the ability to generate likely expressions according to the Java language model, and uses as much of the information from the input as it can extract to steer the generation towards developer's intention.

In summary, this chapter makes the following contributions:

- A new technique that maps text to a list of ranked (partial) expressions. It combines NLP tools, a text-declaration matching, PCFG and unigram models to process input, synthesize and rank expressions.

- A new text and declaration models that encode input and a declaration as a set of words. They prioritize words based on their importance and position, both in text and a declaration.

- Efficiently performing text-declaration matching thanks in part to the creation of appropriate indices and the use of the algorithms for the assignment problem, in particular the Hungarian method.

- A fast corpus analysis and extraction algorithm. The algorithm extracts method compositions and frequencies and builds PCFG and unigram models. The implementation combines the Eclipse JDT parser, our symbol table and type-checker.

- A customized related word-map that maps a word to its related words. We use relations in WordNet and a novel scoring technique that for a given word ranks and finds the closest related words. We use a set of API words to build the score and filter out irrelevant words.

- A benchmark set of 60 text-expression (input-output) benchmarks and experimental result that shows the effectiveness of our techniques and can also be used to calibrate future open ended tool that map free-form text into Java API invocations.

3.2 Examples

In this section we use five examples to illustrate main functionality of *anyCode*. The first example demonstrates *anyCode*'s interactive deployment, the text interface, and the use of program context to guide the synthesis.

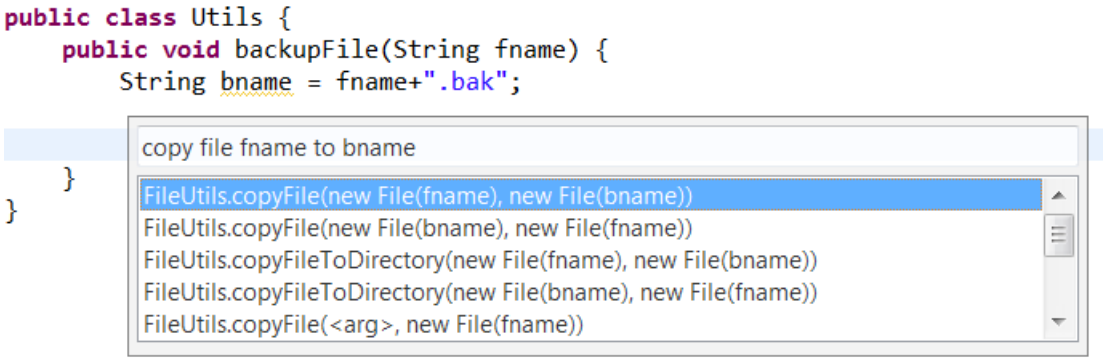

Figure 3.1: After the user inserts text input, *anyCode* suggests five highest-ranked well-typed expressions that it synthesized for this input.

3.2.1 Making a Backup of a File

Suppose that a user wishes to create a method that backs up the content of a file. The method should take a file name as a parameter and copy the content of the file to a new file with an appropriately modified name. To implement such a backup method, the user needs to identify the appropriate API, select the set of its declarations (typically method calls) and combine them into an expression. In practice, to perform this, a user might follow these steps manually: (1) search the Internet, or API documentation (if it exists) to find the examples of API use, (2) select the most suitable example, (3) copy-paste it into the working editor with code, and (4) edit the example such that it fits into the context, using appropriate values in the program scope. *anyCode* offers an automatic approach that combines all the mentioned operations, and more. Suppose that a user writes an incomplete piece of code of the method that takes the parameter `fname` that stores the file name, as shown in Figure 3.1. She also creates a variable `bname` that stores the backup file name. Here, the backup name is obtained by adding ''.bak'' extension to `bname`. In the next line the user invokes *anyCode*. A pop-up text field appears where she can insert the text, such as `copy file fname to bname` that specifies her desire to copy the file content. *anyCode* automatically extracts the program context from Eclipse and identifies words `fname` and `bname` in the input as values referring to a parameter and a local variable. *anyCode* then uses this information to generate and present several ranked expressions to the user. When the user makes her choice, the tool inserts the chosen expression at the invocation point. In this example, *anyCode* works for less than 50 milliseconds and then

presents five solutions of which the first one copies the file fname content to a file with name bname:

FileUtils.copyFile(**new** File(fname), **new** File(bname))

This is a valid solution; it uses the method FileUtils.copyFile from the popular "Commons IO" library.

3.2.2 Invoking the Class Loader

Suppose that a user intends to load a class with a name ''MyClas.class''. She invokes the tool with the free-form input

load class ''MyClas.class''

anyCode automatically synthesizes and suggests the following (partial) expressions:

1 Thread.currentThread().getContextClassLoader().loadClass(''MyClas.class'').getClass()
2 Thread.currentThread().getContextClassLoader().loadClass(''MyClas.class'')
3 Thread.currentThread().getContextClassLoader().loadClass(⟨arg⟩).getClass()
4 ''MyClas.class''.getClass()
5 Thread.currentThread().getContextClassLoader().loadClass(⟨arg⟩)

In this example *anyCode* generates suggestions in less than 40 milliseconds. The second suggestion turns out to be the desired one.

The suggestions 1, 2, and 4 represent complete expressions. On the other hand, suggestions 3 and 5 represent templates that include the symbol ⟨arg⟩ that marks the places where local variables are often used. The main reason why we present templates is that a user often inserts incomplete input and for an incomplete input the best solution is an incomplete output, i.e., a template. If we have insisted only on completed expressions, we would miss many interesting solutions that are more convenient for such an incomplete input. Thus, *anyCode* treats a textual input both as complete and incomplete, and tries to find both complete and incomplete solutions.

Note that the complete expressions 1 and 2 include declarations whose selection and integration does not directly depend on the textual input. For instance, method loadClass contains both input words load and class, whereas currentThread does not. To reach the currentThread from loadClass we use probabilistic language model for Java and its API calls, derived from a corpus of code. Without such a model we would not be able to construct complex expressions such as the above one.

3.2.3 Creating a Temporary File

In the third example we demonstrate the use of semantically related words. For instance, if a user wants to discover templates that make a new file, she may insert 'make file'. In a less than

80 milliseconds, *anyCode* generates the following output:

```
1  new File(⟨arg⟩).createNewFile()
2  new File(⟨arg⟩).isFile()
3  new File(⟨arg⟩)
4  new FileInputStream(⟨arg⟩)
5  new FileOutputStream(⟨arg⟩)
```

Note that word make does not appear among the solutions, because API designers used the word create. *anyCode* succeeds in finding the solution because it considers, in addition to the words such as make appearing in the input, its related words, which includes create. *anyCode* uses a custom related-word map to compute the relevant words. We built this map by automatically processing and adapting WordNet, a large lexical semantic network of English words.

3.2.4 Writing to a File

Consider next the following input of a developer:

```
write ''hello'' to file ''text.txt''
```

For this input, in less than 50 milliseconds *anyCode* outputs:

```
1  FileUtils.writeStringToFile(new File(''text.txt''), ''hello'')
2  FileUtils.writeStringToFile(new File(''hello''), ''text.txt'')
3  FileUtils.writeStringToFile(new File(''hello''), ''hello'')
4  FileUtils.writeStringToFile(new File(''text.txt''), ''text.txt'')
5  FileUtils.writeStringToFile(new File(⟨arg⟩), ''hello'')
```

The expressions under 1 and 2 are ranked higher than, e.g., the solution 3 because they have a higher usage of the input text elements. Indeed, solution 3 does not refer to one of the string literals in the input. The synthesis algorithm and our scoring techniques favor solutions with the greater input coverage. In this example, the first expression performs the desired task.

3.2.5 Reading from a File

In the final example we show that our input interface may also accept an approximate expression. For instance, if a user attempts to write an expression that reads the file, in the first iteration she may write the following expression

```
readFile(''text.txt'',''UTF−8'')
```

Unfortunately, this expression is not well-typed according to common Java APIs. Nevertheless, if *anyCode* takes such a broken expression, it pulls it apart and recomposes it into a correct one, suggesting (again in less than 40 milliseconds) the following solutions:

```
1  FileUtils.readFileToString(new File("text.txt"))
2  FileUtils.readFileToString(new File("UTF−8"))
3  FileUtils.readFileToString(⟨arg⟩)
4  FileUtils.readFileToString(new File(⟨arg⟩))
5  FileUtils.readFileToString(new File("text.txt"), "UTF−8")
```

anyCode first transforms the input by ignoring the language specific symbols (e.g., parenthesis and commas). It then slices complex identifiers, so called *k-words*, into single words. Here, readFile is a 2-word that gets sliced into read and file. Despite the loss of some structure in treating the input, our language model gives us the power to recover meaningful expressions from such an input. This shows that *anyCode* can be used as a simple expression repair system. The desired solution is ranked fifth because it uses a version of readFileToString method with two arguments, which appears less frequently in the corpus than the simpler versions of the method.

We have evaluated our system on a number of examples; Tables 3.1 and 3.2 show 60 text queries and the code that we expected to obtain in return. The "All" column indicates the rank on which the expression was found with all features of our system turned on, as discussed in Section 3.4.

3.3 System Overview

In this section we give a high-level picture of the main components of our system. Input to our system consists of i) a textual description, explicitly typed by the developer and ii) a partial Java program with a position of the cursor, which *anyCode* extracts automatically from the Eclipse IDE. *anyCode* uses the input to generate, rank and present (possibly partial) expressions to the user. As Figure 3.2 shows, the key components of *anyCode* are the input text parser, the declaration search engine, and the expression synthesizer. The method getExpressions is the main method that performs these steps, as outlined in Figure 3.3.

The first goal of parsing is to identify structure of the input text using a set of natural language processing tools. *anyCode* uses the structure to group input words into WordGroups. The intuition is that the input text corresponds to several declarations, and grouping according to the rules of English helps to identify these declarations from multiple input words. Moreover, the system uses a map of related words to complete the words given in the input with some of the related meanings computed from a modified version of WordNet [36]. To complement natural language input, the system uses program context from the IDE to mark local variables and literals in the input text. Section 3.5 describes our parsing and related word completion techniques in more depth.

In the declaration search engine, the system uses the groups, WordGroups, to find a subset of API declarations that are most likely to form the final expressions. The system tries to match word groups against declarations in our API collection. To perform matching, the system

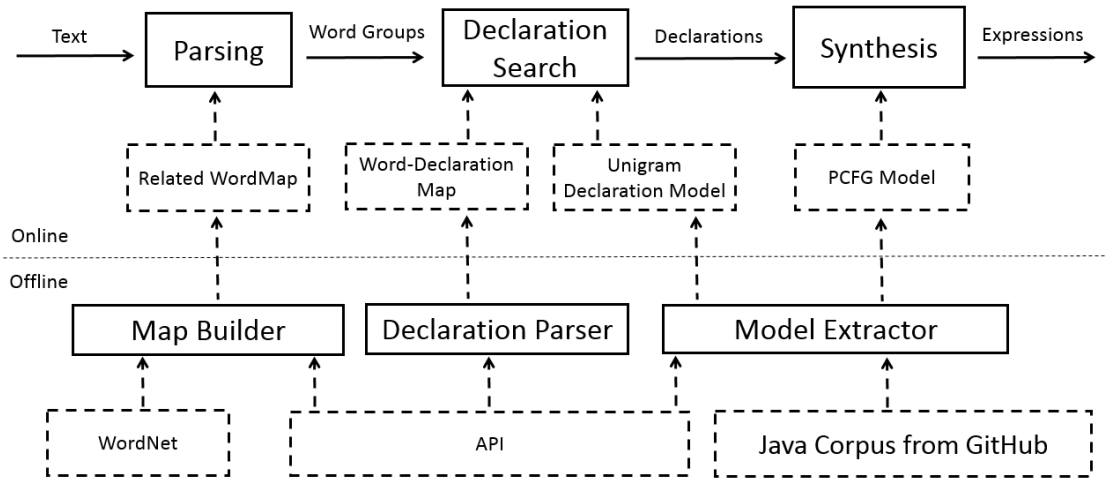

Figure 3.2: *anyCode* system overview. The offline components run only once and for all. The online components run as part of the Eclipse plugin.

```
getExpressions(text, context, N−bestExprs):
  // Text Parsing
  (WordGroups, Literals, Locals) ← parse(text, context)
  // Declaration Search
  DeclGroups ← declSearch(WordGroups, API, Unigram)
  // Synthesis
  ExPCFG ← extend(PCFG, Literals, Locals)
  Exprs ← synth(DeclGroups, ExPCFG , N−steps)
  return keepBest(Exprs, N−bestExprs)
```

Figure 3.3: The high level description of online portion of *anyCode*.

extracts a list of words from declarations and matches them against the words in the groups. Based on the number of words that match, declaration Unigram [51, Chapter 4] score, and other parameters the system estimates the matching score. We explain this in more details in Section 3.9. Finally, for each word group the system selects the top N−best declarations with the highest scores and puts them in a declaration group. In summary, the method declSearch transforms each word group into a declaration group. Lastly, we would like to mention that API contains a set of declarations we collect from different APIs and packages. This is done in advance, before a user invokes *anyCode* for the first time.

In the last step the system uses declaration groups and a probabilistic context free grammar (PCFG) model [51, Chapter 14] to synthesize expression. ExPCFG consists of the initial PCFG model, pre-collected from a large Java source code corpus and the extension, a set of production rules for literals and local variables. The method extend extends PCFG with the extension and returns ExPCFG. The method synth tries to unfold declaration arguments following ExPCFG model, in N−steps. Given a declaration, ExPCFG suggests declarations that

should fill in the argument places. The method synth also assigns scores to the expressions, based on the ExPCFG and declarations scores. Finally, the system uses the method keepBest in order to filter the top N−bestExprs expressions with the highest scores.

3.4 Evaluation

This section discusses a set of benchmarks we use to evaluate *anyCode* and presents the experimental results.

3.4.1 Benchmarks

We wrote 60 benchmarks shown in Tables 3.1 and 3.2. Each benchmark consists of a textual description and local variables as input, and a desired expression as output. We say that the system passes the benchmark if for the given input, the desired expression appears among the synthesized expressions. We use the benchmarks to estimate the effectiveness and the importance of different aspects of our system.

We ran all experiments on a machine with a quad-core processor with 2.7Ghz clock speed and 16MB of cache. We imposed a 8GB limit for allowed memory usage. Software configuration consisted of Windows 7 (64-bit) and Java(TM) Virtual Machine 1.7.0.55. The declaration search and the expression synthesis algorithm make use of multiple CPU cores.

We parametrized *anyCode* with N−bestExprs=10 and N−steps=5. By using a N−step=5 limit, our goal was to evaluate the usability of *anyCode* in an interactive environment (which IDEs usually are).

Results are shown in Tables 3.1 and 3.2. The Input column represents the textual descriptions, and the Output column represents the expected expressions. The column Rank represent the ranks of the expected expressions after we run *anyCode*. With >10 we mark the case when the expected expression is not among the top ten synthesized expressions. The Rank column is split in three sub-columns. Each column relates to a different *anyCode* configuration. The first, NoPU, denotes *anyCode* that does not use unigram and PCFG models. In this setting all declarations have the same unigram score and all PCFG productions have the same probability. The second, NoP, denotes *anyCode* that uses unigram, but does not use PCFG model. In this setting the tool uses unigram model to select declarations, but all PCFG productions still have the same probability. The last, All, denotes *anyCode* that uses both unigram and PCFG to guide the synthesis algorithm and to rank the expressions. The results show that the system without the both models generates only 8 (13%) expected expressions among the top ten solutions. The system with unigram model generates 21 (35%) and the system with both models generates 56 (93%) expressions. In the latter case, we observe that the expected expressions are not synthesizes in 4 examples for two reasons. The first reason is that the statistical data on declaration compositions, required to build those expressions, cannot be extracted and found

	Input	Output	Rank			Time
			NoPU	NoP	All	[ms]
1	copy file fname to bname	FileUtils.copyFile(new File(fname), new File(bname))	>10	>10	1	47
2	does x begin with y	x.startsWith(y)	>10	>10	1	62
3	load class “MyClass.class”	Thread.currentThread().getContextClassLoader() .loadClass(“MyClass.class”)	>10	>10	2	31
4	make file	new File(<arg>).createNewFile()	>10	>10	1	78
5	write “hello” to file “text.txt”	FileUtils.writeStringToFile(new File(“text.txt”), “hello”)	>10	>10	1	47
6	readFile(“text.txt”,“UTF-8”)	FileUtils.readFileToString(new File(“text.txt”), “UTF-8”)	>10	>10	5	31
7	parse “2015”	Integer.parseInt(“2015”)	>10	>10	1	16
8	substring “EPFL2015” 4	“EPFL2015”.substring(4)	>10	>10	1	31
9	new buffered stream “text.txt”	new BufferedReader(new InputStreamReader(new BufferedInputStream(new FileInputStream(“text.txt”))))	>10	1	1	47
10	get the current year	new Date().getYear()	>10	>10	6	62
11	current time	System.currentTimeMillis()	1	1	1	16
12	open connection “http://www.oracle.com/”	new URL(“http://www.oracle.com/”).openConnection()	>10	>10	1	31
13	create socket “http://www.oracle.com/” 8080	new Socket(“http://www.oracle.com/”, 8080)	>10	>10	6	47
14	put a pair (“Mike”,“+007-345-89-23”) into a map	new HashMap().put(“Mike”, “+007-345-89-23”)	>10	9	1	109
15	set thread max priority	Thread.currentThread().setPriority(Thread.MAX_PRIORITY)	>10	>10	1	109
16	set property “gate.home” to value “http://gate.ac.uk/”	new Properties().setProperty(“gate.home”, “http://gate.ac.uk/”)	>10	>10	3	94
17	does the file “text.txt” exist	new File(“text.txt”).exists()	>10	4	1	47
18	min 1 3	Math.min(1, 3)	>10	7	1	31
19	get thread id	Thread.currentThread().getId()	>10	1	1	31
20	join threads	Thread.currentThread().join()	>10	1	2	16
21	delete file “text.txt”	new File(“text.txt”).delete()	>10	1	1	31
22	print exception ex stack trace	ex.printStackTrace()	>10	>10	7	47
23	is file “text.txt” directory	new File(“text.txt”).isDirectory()	>10	>10	2	46
24	get thread stack trace	Thread.currentThread().getStackTrace()	>10	1	1	47
25	read line by line file “text.txt”	FileUtils.readLines(new File(“text.txt”))	>10	>10	2	94
26	set time zone to “GMT”	Calendar.getInstance().setTimeZone(TimeZone.getTimeZone(“GMT”))	>10	>10	1	47
27	pi	Math.PI	2	1	1	15
28	split “EPFL-2015” with “-”	“EPFL-2015”.split(“-”)	>10	>10	1	16
29	memory	Runtime.getRuntime().freeMemory()	2	2	1	15
30	free memory	Runtime.getRuntime().freeMemory()	3	4	1	16
31	total memory	Runtime.getRuntime().totalMemory()	2	2	1	31
32	exec “javac.exe MyClass.java”	Runtime.getRuntime().exec(“javac.exe MyClass.java”)	>10	1	1	16

Table 3.1: The table that shows the results of the comparison of the different *anyCode* configurations with and without unigram and PCFG models (part 1).

	Input	Output	Rank			Time
			NoPU	NoP	All	[ms]
33	new data stream “text.txt”	new DataInputStream(new FileInputStream(“text.txt”))	>10	>10	4	47
34	rename file fname1 to fname2	new File(fname1).renameTo(new File(fname2))	>10	>10	1	31
35	move file fname1 to fname2	FileUtils.moveFile(new File(fname1), new File(fname2))	>10	>10	1	62
36	concat “EPFL” “2015”	“EPFL”.concat(“2015”)	>10	3	1	16
37	read utf from the file “text.txt”	new DataInputStream(new FileInputStream(“text.txt”)).readUTF()	>10	>10	10	47
38	java home	SystemUtils.getJavaHome()	2	1	1	15
39	upper(text)	text.toUpperCase()	>10	2	1	31
40	compare x y	x.compareTo(y)	>10	>10	1	32
41	BufferedInput “text.txt”	new BufferedInputStream(new FileInputStream(“text.txt”))	>10	>10	1	15
42	set thread min priority	Thread.currentThread().setPriority(Thread.MIN_PRIORITY)	>10	1	1	78
43	create panel and set layout to border	new Panel().setLayout(new BorderLayout())	>10	1	1	172
44	sort array	Arrays.sort(array)	>10	>10	1	15
45	add label “names” to panel	new Panel().add(new Label(“names”))	>10	>10	1	78
46	write 2015 to data ouput stream “t”	new DataOutputStream(new FileOutputStream(“t”)).write(2015)	>10	>10	>10	70
47	get date when file “t” was last time modified	new Date(new File(“t”).lastModified()).getTime()	>10	>10	3	240
48	check file “t1” “t2” permission	AccessController.checkPermission(new FilePermission(“t1”, “t2”))	>10	>10	1	50
49	read lines with numbers from file “t”	new LineNumberReader(new InputStreamReader(new FileInputStream(“t”))).readLine()	>10	>10	5	80
50	StreamTokenizer(“t”)	new StreamTokenizer(new BufferedReader(new FileReader(“t”)))	>10	>10	6	30
51	read from console	new BufferedReader(new InputStreamReader(System.in)).readLine()	>10	>10	7	20
52	is file “t” data available	new DataInputStream(new FileInputStream(“t”)).available()	>10	>10	1	50
53	SequenceInputStream(“t1”, “t2”)	new SequenceInputStream(new FileInputStream(“t1”), new FileInputStream("t2"))	>10	>10	>10	50
54	get double value x	Double.valueOf(x).doubleValue()	>10	>10	>10	40
55	write object x to file “t”	new ObjectOutputStream(new ByteArrayOutputStream()) .writeObject(“t”)	>10	>10	1	60
56	1 xor 5	new BitSet(1).xor(new BitSet(5))	>10	>10	>10	30
57	create bit set and set its 5th element to true	new BitSet(5)	3	7	1	180
58	accept request on port 80	new ServerSocket(80).accept()	>10	>10	6	50
59	ResourceStream(“t”)	ClassLoader.getSystemResourceAsStream(“t”)	>10	>10	1	33
60	gaussian	new Random(System.currentTimeMillis()).nextGaussian()	3	3	3	10

Table 3.2: The table that shows the results of the comparison of the different *anyCode* configurations with and without unigram and PCFG models (part 2).

in the corpus. The second reason is that some selected declarations, unrelated to the expected expressions, are more popular than the expected declarations. Such declarations contribute to the synthesis of the expressions that are ranked higher that the expected one, and for this reason the expected expression does not appear among the top ten solutions.

Finally, the column Time shows the times needed to synthesize the top ten expressions for the *anyCode* with both models turned on. All times are between 10 and 240 milliseconds, with the average time of 50.6 milliseconds.

In summary, the results illustrate that our system greatly benefits from the unigram and the PCFG models. They also show that *anyCode* can efficiently synthesize the expressions in a small period of time (in less that 250 milliseconds).

3.4.2 Threats to Validity

The primary threat is to external validity: our set of examples, while fairly large by the standards of previous literature, may not be representative of general results. This limitation comes from the two facts: (1) there is no standardized set of benchmarks for the problem that we examine, and (2) we used the first 45 examples in Tables 3.1 and 3.2 to configure our system. Later, we used all 60 examples to evaluate our system. The primary purpose of the examples is to show that our tool is able to produce a set of real-worlds examples when configured. The parameters that we configure include declaration selection and reward-penalty parameters. The declaration selection parameters are important for our word-declaration matching. A parallel corpus with text as input and declarations (expressions) as output would be ideal for configuring the parameters, however no such corpus exists. Our attempt to create the corpus from the code and its descriptive comments led to irrelevant examples and the low quality corpus. The reward-penalty parameters are used to effect the size of the expressions, the aspect over which PCFG and unigram models have no control. In the rest of the chapter we take a deeper look at the techniques we employed in *anyCode*.

3.5 Parsing

We perform parsing both on an input text and API declarations. We use parsing to prepare the text and the declarations for the search and the matching.

In the sequel, a k-word denotes a chain of k English words connected without a whitespace between them, as often used in Java identifiers. The words are separated at places where a small letter meets a capital letter. Usually, declaration names are k-words (e.g. “readFile” is a 2-word that contains words “read” and “file”). A 1-word is a single English word. A token is either a k-word, a literal or a local variable name. A literal is a number, a string, or a boolean value. The input text consists of tokens, whereas declarations contain only k-words.

3.5.1 Input Text Parsing

The main idea behind input parsing is to group words in the text such that the groups can be matched to desired declarations. To describe the parsing we use a running example that shows different phases of the parse method in Table 3.3.

	Parsing		Example				
1	Text Input		copy	file	fname	to	"C:/user/text2.txt"
2	Decomposition		copy	file	Loc(fname)	to	L("C:/user/text2.txt")
3	Type Substitution		copy	file	String	to	String
4	POS Tagging		copy/Verb	file/Noun	String/Noun	to/To	String/Noun
5	Grouping	Words	1	2	3	4	5
		Primary	copy/Verb	file/Noun	String/Noun	to/To	String/Noun
		Secondary	file/Noun, to/To	String/Noun		String/Noun	
		Related	duplicate/Verb		chain/Noun		chain/Noun

Table 3.3: Phases of parsing an example input sentence.

Let us assume a user inserts 'copy file fname to ''C:/user/text2.txt''', as shown in the first row in Table 3.3. Each row represent the input to the next phase, but also the output of the previous phase.

In the input we identify the following tokens: three single words, one local variable and a string literal. We mark local variables and literals as shown in the second row. In the next phase we substitute literals and local variables with their types, and produce the output in the third row. The intuition is that literals and local variables will appear as declaration arguments, accordingly, we use their types to match potential declaration argument types. To perform our next parsing step we use the Stanford CoreNLP [29, 67, 92]. The CoreNLP is a pipeline of a natural language processing tools that takes an English raw text as input and returns tagged text with deep semantic connections among the words. First, we use the tools to lemmatize and tag the words. A lemmatizer analyzes a word context to obtain the word's canonical form, called lemma (e.g., "good" is the lemma of "better"). The Part-of-Speech (POS) tagger assigns a part of speech tags (e.g., noun, verb, adjective and etc.) to each word, based on the context. The tagging is important for two reasons. The first reason is that tagging is a necessary preprocessing step towards deeper text analysis, which we also perform. The second reason is that different tags impose different word places in a declaration. For instance, we observe that verbs appear in method-names, whereas non-verbs can appear almost anywhere. Therefore, POS-tags can help in assigning different roles and priorities to words. We also tag the words inside each k-word (if it contains more than one word).

To properly group the words, such that each group corresponds to a single declaration, we need more information than tags can provide. Thus, we use the parser to identify different semantical relations among the words. The result is a semantic graph, as show in Figure 3.4. The graph includes relations like "nsubj" (a nominal subject) or "dobj" (direct object) that identify a predicate, a subject and an object in a sentence. The relations are important because they separate the words that are more likely to appear in declaration names from the words that may appear as arguments. Because we expect that a user will type more often declaration

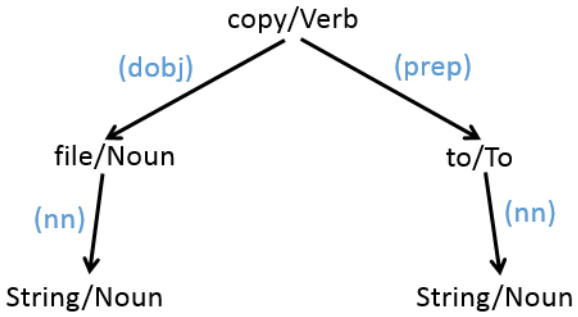

Figure 3.4: Natural language semantic graph for the input from Table 3.3.

name words to refer to the declaration, we call them *primary* words. The other words, which are more likely to appear as arguments, we call *secondary* words.

In the last phase, we form word groups, such that each group has primary, secondary and related words (see row five in the table). In our example we form five different groups marked with numbers 1-5. The group contains all the words below the corresponding number. The primary words are obtained directly from a k-word from the phase 4. The secondary words are connected to the primary words via the semantic graph. Namely, the secondary words are the neighbors (children) of the primary words in the graph. The related words are API words that are related to the primary words. Those words include synonyms, hypernyms and hyponyms. We build our own map of related words using set of all API words and WordNet [36], see Section 3.6. An important constraint that we try to fulfill is that each group contains at least one non-verb word, because we observe that declarations usually contain at least one non-verb word.

3.5.2 Declaration Parsing

We also use parsing in the pre-processing stage to create the representation of API declarations. We next define what we mean by a declaration, then sketch an algorithm that extracts words from declarations, forming a model suitable for matching.

Declaration Representation

A declaration can be a class method, a constructor, or a field. A declaration has a name and type, as well as an optional owner. Declaration name is a k-word. Declaration type can be: a Java primitive type, an API class type or a function type with multiple argument types, built from class and simple types. A declaration can be static or instance. A static declaration contains the owner class name. An instance declaration possesses also a receiver type, which we treat as any argument type.

Declaration Word Model

To match a declaration with a word group, we extract k-words from a declaration. Then we decompose complex k-words into single ones. Next, we apply lemmatizer and POS tagger to the words. In our final step we put the words into the primary or the secondary group based on the position in the declaration. The words that appear in the declaration name become the primary words and the word that appear elsewhere become the secondary words. Our goal is to assign higher priority to the primary words and lower to the secondary words. Section 3.9 shows the framework and the mechanisms to assign such priorities.

The declaration parsing allows us to transform API declaration set into a word-declarations hash map. This allows us to use an input word to quickly find all declarations that contain it. This optimization is crucial when searching for declarations.

3.6 Related WordMap: Modifying WordNet

To support inputs that does not strictly follow the actual words of API declarations, we use WordNet [36], a large lexical database of English words. WordNet groups words into synsets, which are sets of synonyms. Each synset represents a different meaning of a word. WordNet is a graph with synsets as the vertices and the relations as edges. The relations between synsets include antonyms, hypernyms and hyponyms.

API declarations contain only a subset of English words. We refer to this subset the *API words*. We build a map from all English words in WordNet to a ranked list of related API word meanings. A key of the map is an English word and the value is a ranked list of related API word meanings. The resulting related words are organized into meanings (synsets) to whom we assign scores (a value between zero and one). This score subsequently becomes the related weight $weight_r$ (see Section 3.9.1) of a particular related word.

We build meanings using WordNet relations. For each English word we first find synsets (synonyms). Then, we find the synsets' closest hypernyms and hyponyms. We filter out all non-API words. Finally, we assign scores using textual descriptions assign to each synset in WordNet. We use Stanford POS-tagger to tag each word in a description. Then, we calculate a percentage of API words in the description. We assign a description score to each synonym synset. However, to a hypernym (hyponym) synset we assign a score that is a product of its description score and the score of the synonym synset that input word uses to reach the hypernym (hyponym). This way we give an advantage to synonyms over *their closest* hypernyms and hyponyms.

3.7 Declaration Search

We expect that the developer will often insert a short text (two to ten words), omitting many details of a desired expression. The advantages of such an approach for developer are faster coding, with more time to focus on other development aspects. Such use, however, brings further challenges for generating expressions. A typical short and ambiguous input does not make it clear which declarations the expression should use, nor how to compose them. To solve this difficulty, our first step is to use the words as a starting point in identifying the desired declarations.

Recall from Figure 3.3 that the parsing phase returns word groups, making use of relations in the parse tree. Recall also that these word groups are enriched with related words based on WordNet. We use such word groups to find candidate declarations. The declarative description of this algorithm is simple: we match a word group with all declarations, calculate the matching score for each declaration, and find $N-\mathsf{bestDecl}$ declarations with the top score. We apply the same procedure to each group. To make the solution practical, we split the algorithm into two steps. *First*, we use a word group and the word-declaration API map to select a set of declarations that match with *at least one* word in the word group. In practice, the selected set is far smaller that the entire API collection. As a consequence, we will have far fewer calls to the expensive scoring procedure in the second step. The *second* step calculates the score for each selected declaration using the matching procedure from Section 3.9.1 and the unigram model. From the unigram model we obtain the declaration probability. We apply the two steps to each group in a list of word groups.

3.8 Synthesis

In this section we describe the algorithm that synthesizes the expressions using PCFG model. Because the algorithm relays on PCFG we first explain the model and later the expression synthesis.

3.8.1 Probabilistic Context Free Grammar Model

The idea is to use the information on declaration compositions to synthesize expressions. In general, a declaration has multiple arguments, meaning that it simultaneously composes with multiple declarations. We decide to treat the simultaneous compositions as one inseparable *multi-composition*. We collect from the corpus API *multi-composition*s and form the PCFG model.

Let us formally define our PCFG model. Let SType be a set of all simple types, that include Java primitive and all public API types (including classes and interfaces). Next, let Names be a set of declaration names, simple type names and the local variable names from the context. Finally, let $\mathsf{Literals}$ be the set of all literals from the textual input. Then, the production rules of our

PCFG can be described by the following grammar:

production ::= decl−production | localvar−production | literal−production

decl−production ::= ?Decl(name) $\xrightarrow{p}$ [rec.] name [(hole, ..., hole)]
hole :: = ?Decl(name) | ?LocalVar(type) | ?Literal(type)
rec :: = owner−name | hole

localvar−production ::= ?LocalVar(type) $\xrightarrow{p}$ localvar−name
literal−production ::= ?Literal(type) $\xrightarrow{p}$ literal

type ∈ STypes owner−name, name, localvar−name ∈ Names literal ∈ Literals

There are three kinds of production rules: a declaration, a local variable and a literal production rule. A declaration production rule encodes how a declaration with a name name is simultaneously composed with holes. Intuitively, each declaration production rule encodes one *multi-composition*. Before we describe declaration production rules in more details, let us introduce holes. There are three kinds of holes: a declaration (?Decl(name)), a local variable (?LocalVar(type)) and a literal (?Literal(name)) hole. Each hole is a nonterminal, which at the same time contains additional information. For a declaration it stores a name, for a local variable and a literal it stores a type.

Now, we can describe declaration production rules. First, note that name appears at the both sides of a declaration production rule decl−production. On the left-hand side it appears as a part of the hole and nonterminal ?Decl(name). On the right-hand side the same name appears as a terminal. It is preceded by rec that is either a hole or an owner class name. The declaration name is followed by the holes. The holes keep the information on declaration arguments and thus encode compositions.

We treat literals and local variables differently from declarations. During PCFG extraction, we abstract away literal and local variable names and leave only type information. Moreover, we do not build production rules for literal and local variable holes from the corpus, but from the user context. This means that our PCFG model collected from a corpus is incomplete. We complete it once we have the textual input and the context of the partial program where a user invoked *anyCode*. From the context and the textual input we build the missing production rules, localvar−production and literal−production, and expand PCFG model. These rules allow us to synthesize expressions with the local variables from the context and literals from the textual input.

Finally, for the remaining nonterminals without production rules we add the rules:

localvar−production ::= ?LocalVar(type) $\xrightarrow{1}$ ⟨arg⟩
literal−production ::= ?Literal(type) $\xrightarrow{1}$ default−literal
default−literal ::= "?" | 0 | **false**
type ∈ STypes

The symbols ⟨arg⟩, "?", 0 and $false$ are special termination symbols that represent missing arguments.

The reason why we call the non-terminals, holes, is that, during synthesis, they appear as holes in partial expressions. We use holes to access production rules that unfold them. We also keep a probability p for each rule, which is based on the rule frequency in the corpus. Also, note that a single hole can have a several different productions, e.g., denoting a several different *multi-compositions* for a single declaration.

3.8.2 Partial Expression Synthesis

A *partial expression* denotes an intermediate result arising in the construction of expressions of interest. A *complete expression* would be made up of variables, literals, and declaration applications. A partial expression may additionally contain different kinds of *holes*, as well as *connectors*. Holes denote places where a partial expression may expand using the rules of PCFG. A connector denotes a place in a partial expression that can expand by substituting another synthesized partial expression.

Our algorithm proceeds in two following phases:
First phase turns every declaration group into a partial expression group. For each declaration in a declaration group:

1. We wrap a declaration name in a hole, creating the initial partial expression.
2. We use production rules of PCFG to unfold holes in partial expressions. This creates new partial expression(s). Gradually, partial expressions grow, and we continue unfolding them until we reach some maximum number of steps. When we can, we insert a connector at the place where partial expressions may merge. This way, we leave the merging until the next phase. At each step, the arguments are unfolded using the PCFG score with the highest probability.
3. We calculate a partial expression score, based on the PCFG model and the declaration scores.
4. The expressions without holes form a partial expression group. (The expressions may have connectors.)

Second phase tries to connect as many as possible expressions that belong to the different partial expression groups. For each partial expression in a partial expression group:

1. We find connectors in the partial expression and substitute them with the appropriate expressions from another group. Again, this creates more partial expressions. They gradually grow and we keep substituting connectors until some maximum number of steps is reached.

2. The score of a new partial expression is calculated using PCFG model, declaration scores, the text input coverage and expression repetition parameters as explained in Section 3.10.

3. We keep and present to a user some maximal number of expressions.

The algorithm synthesizes expressions using eager selection that does not guarantee always the optimal solution, in terms of the PCFG probability and the expression score. Therefore, to make the algorithm more effective, we use the practical approach that synthesizes more expressions than specified by a user, then it sorts them by the score, and finally selects and outputs the required smaller number of the expressions.

3.9 Declaration Score

WordGroup-Declaration matching score and declaration unigram score influence total declaration score. More precisely the declaration score is equal to the product of the two scores.

3.9.1 WordGroup-Declaration Matching Score

We next present a word group and a declaration matching algorithm. Recall that the word group contains primary, secondary and related words. We further split them based on their input origin. Namely, each word has a word in input to which it belongs. We call this word an origin; a group can have a several origins. Our goal is to maximize the matching score between group origins and declaration words. However, we do not match an origin with a declaration word. Instead, we split the group into disjoint subgroups based on origins, and match subgroups with declaration words. A word matches with a subgroup if a subgroup contains the lexically identical word with the same tag. Let us create the bipartite graph such that one set, SGS, of vertices is a set of subgroups, and the other, W, is the set of declaration words. Also, we create edges between subgroups and declaration words that match. Each edge has a weight, a value between zero and one. Now, calculating the maximum matching score turns into finding a maximum weight matching in a weighted bipartite graph. This is the well-known assignment problem [19]. To solve it, we choose the Hungarian method [56], with the following optimizations. We observed that many vertices remain without edges, so we remove all such vertices. We also observe that the bipartite graph is usually disconnected. We identify its connected components, and decompose it into smaller bipartite graphs. We calculate the score of each component and sum them up. The final score is the word group-declaration matching score. Another optimization is that, if we need to calculate matching 1 to n, we simply find the maximum among the weights. This is an important optimization because this sort of matching and its special case (1 to 1 matching) often occur. We define a total match weight between a group word w_g and a declaration word w_d as follows:

$$weight_{match}(w_g, w_d) = weight_i(w_g) * weight_i(w_d) * weight_k(w_g, w_d) * weight_r(w_g)$$

Word Importance Weight. Primary words are more important than secondary and related words. To encode this, we introduce word importance weight, $weight_i$, which is a real value between one and zero and assigns a higher value to a primary than to the secondary and related words.

Kind Weight. We reward more a matching between primary input and primary declaration words (a primary-primary match) than a primary-secondary and a secondary-secondary matching. (Related words are treated same as primary words in this context.) We use a quantitative function, called kind weight, $weight_k$, that returns a real value between one and zero. It assigns a higher value to a primary-primary than to a secondary-primary and a secondary-secondary matching.

Related Word Weight. We use WordNet and collection of API words to calculate $weight_r$ (see also Section 3.6). We penalize related words that are *far* from the primary words. To measure a primary-related word *distance* we introduce a quantitative function, called related word weight, $weight_r$, that returns a real value between one and zero. Synonyms are closer to a word than hyponyms and hypernyms.

To apply the matching algorithm, we take the maximum weight between a subgroup SG and a declaration word w, to be the edge $(\mathsf{SG}, \mathsf{w})$ weight:

$$weight_{match}(\mathsf{SG}, \mathsf{w}) = \mathsf{max}(\{weight_{match}(w_g, \mathsf{w}) \mid w_g \in \mathsf{SG}\})$$

3.9.2 Declaration Unigram Score

The unigram model [51, Chapter 4] assigns a probability to each declaration based on call frequency in a corpus. The higher the declaration frequency, the higher the probability. We smooth the model by assigning the minimal frequency value (collected in the corpus) to a declaration that does not appear in the corpus. The declaration unigram score is equal to the declaration probability.

3.10 Partial Expression Score

To rank the (partial) expressions, $expr$, and to guide the synthesis algorithm we use the score $score(expr)$ that is computed by the following formula:

$$score(expr) = \log(score(expr)_{pcfg}) + \log(score(expr)_{decl}) + score(expr)_{cov} - score(expr)_{rep}$$

The PCFG score, $score(expr)_{pcfg}$, is equal to the product of all composition probabilities used in the (partial) expression. The declaration score, $score(expr)_{decl}$, is equal to the product of all declaration scores whose declarations appear in $expr$. The coverage score, $score(expr)_{cov}$, estimates and favors the higher coverage of the input text words. We say that the word is covered if it selects a declaration in $expr$. Finally, the repetition score, $score(expr)_{rep}$, is proportional to the number of extra partial expressions used in $score(expr)_{rep}$. We say that a

partial expression is extra if another partial expression from the same partial expression group is also used in $score(expr)_{rep}$. Preferably, we would like to use a single partial expression per each partial expression group.

3.11 Constructing PCFG and Unigram Models

We build both unigram and probabilistic context-free grammar models by analyzing the corpus of projects from GitHub.

Java Code Corpus. We use the GitHub Java corpus [11] that contains over 14'500 Java projects. The corpus includes only projects from GitHub that were forked at least once, to select more popular repositories. We decide to analyze each Java source file individually to reduce analysis time and avoid the need to execute essentially arbitrary build processes of various projects. Whereas this reduces the quality of the data we can extract from corpus, it is compensated by the fact that we can analyze many more projects: we were able to analyze 14'500 Java projects containing over 1.8 million files.

Model Extraction. We use Eclipse JDT parser [2] to parse each file. To improve the model we build our own symbol table and type-checker. The symbol table keeps track of all imported API declarations (of our interest). We analyze an expression using the symbol table and the type-checker. The symbol table identifies API declarations in an expression and the type-checker checks if the expression type-checks against them. Using those methods we extract a declaration multi-composition along with its occurrence frequency. We also extract the declaration occurrence frequency. Then, we use them to calculate the composition's and the declaration occurrence probabilities. Finally, we use the composition and its probability to build PCFG model, and the declaration and its probability to build Unigram model. The models are formed once we extract the information for all compositions and declarations in the corpus. In general, an expression contains user-defined declarations. We reduce their number to improve the quality of the analysis. In particular, where suitable, we inline local variables. The rest we mark with a special symbols and encode them in the PCFG model.

3.12 Related Work

We mention related work that combine NLP and program synthesis techniques, as well as program synthesis tools with similar goals as our work.

SmartSynth [60] generates smartphone automation scripts from natural language descriptions. It uses NLP techniques to infer components and their partial dataflow from NL description. Then, it uses type based synthesis to constructs the scripts. Macho [25] transforms natural language descriptions into a simple programs using a natural language parser, a database of corpus and input-output examples. It maps English into database queries, then selects them, combines them and test them using examples. The queries are based on variables names. Little

and Miller [62] built a system that translates a small number of keywords, provided by the user, into a valid expression. It extracts words from a declaration in the context, and tries to match them using *explanatory* vectors. The system tries to cover as many as possible words from the input, using declaration words. It also penalizes the unmatched words. NaturalJava [79] allows a user to create and manipulate Java programs using an NL input. It uses a restricted form of NL, based on Java's programming concepts, and translates it to Java statements. It requires the user to think and explicitly describe commands at the syntactical level of Java. Also, Metafor [63] transforms a story (in NL form) into a program template. It tries to obtain program structure by interpreting nouns as program objects, verbs as functions and adjectives as properties.

Unlike all mentioned tools, *anyCode* uses code corpus, PCFG and unigram model to synthesize and rank the expressions. We also *automatically* infer the set of words that map to declaration (components). Finally, those tools usually relay on the mapping model, where verbs are mapped to actions (methods), and nouns to objects (arguments). As discussed in Section 3.9.1, we introduce a more sophisticated model and its framework that maps an input text to declarations, resolves complex declaration names and takes into account related words (e.g. synonyms). We also show how to encoded the mapping into the assignment problem and solve it using Hungarian method.

SLANG [80] takes a program with holes and produces the most likely completions, sequences of method calls. It uses an N-gram language model to predict and synthesize a likely method invocation sequence, as well as method arguments. SNIFF [22] uses natural language to search for code examples. It collects a corpus, code examples, and uses API documentation to annotate the examples, and method calls, with keywords. As mentioned before in Chapter 2, InSynth asks user to specify the desired type and produces a set of ranked expressions, instances of the desired type. InSynth ranks the solutions based on the declaration unigram model. In this chapter, we change the input interface to the textual one, giving a user more freedom in specifying his wishes. We additionally use more sophisticated PCFG model, extracted from a far bigger corpus than the one used in InSynth. CodeHint [41] is a dynamic synthesis tool that uses a runtime information to generate and filter candidate expressions. A user provides tests and a specification, and tool generate candidates and checks them against the tests and specification. To guide a generation, the tool uses only a declaration unigram model. XSnippet [82] takes a user query to extract Java code from the sample repository. It offers a range of queries from generalized to specialized. XSnippet ranks solutions based on their length, frequency, and context-sensitive as well as context-independent heuristics. The user needs to initiate additional queries to fill in the method arguments. Strathcona [47] automatically extracts a query based on the structure of the developed code. It does not allow a user to explicitly describe their needs. PARSEWeb [88] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. The advanced code completions tools [8, 18], proposes declarations and code templates. Both systems use API declaration call statistics from the existing code examples to present a solutions with appropriate statistical confidence value.

3.13 Conclusions

We presented *anyCode*, to the best of our knowledge the first tool for code synthesis that combines unique flexibility in both its input and its output. On the one hand, *anyCode* performs parsing of the free-form text input that may contain a mixture of English and code fragments. On the other hand, *anyCode* automatically constructs valid Java expressions for a given program point and is able to generate combinations of methods not encountered previously in the corpus. Ensuring this flexibility required a new combination of techniques from natural language processing, code synthesis, and statistical inference. Our experience with the tool, as reported on 60 diverse examples, suggests that there is a number of scenarios where such functionality can be useful for the developer.

4 Test Generation through Programming in UDITA

We present an approach for describing tests using non-deterministic *test generation programs*. To write such programs, we introduce UDITA, a Java-based language with non-deterministic choice operators and an interface for generating linked structures. We also describe new algorithms that generate concrete tests by efficiently exploring the space of all executions of non-deterministic UDITA programs.

Unlike synthesis tools we presented in previous chapters, UDITA requires more intervention from a user. The reason is that synthesizing automatically complex test inputs, like programs, is often more challenging than synthesizing code fragments. Therefore, in some cases users will need to describe as precisely as possible such test inputs in order to help a system to generate them efficiently. This is where UDITA technique is useful. While less automated, it can generate many programs quickly and is thus useful for test generation.

We implemented our approach and incorporated it into the official, publicly available repository of Java PathFinder (JPF), a popular tool for verifying Java programs. We evaluate our technique by generating tests for data structures, refactoring engines, and JPF itself. Our experiments show that test generation using UDITA is faster and leads to test descriptions that are easier to write than in previous frameworks. Moreover, the novel execution mechanism of UDITA is essential for making test generation feasible. Using UDITA, we have discovered a number of bugs in Eclipse, NetBeans, Sun javac, and JPF.

4.1 Motivation

Testing is the most widely used method for detecting software bugs in industry, and the importance of testing is growing as the consequences of software bugs become more severe. Testing tools such as JUnit are popular as they automate text *execution*. However, widely adopted tools offer little support for test *generation*. Manual test generation is time-consuming and results in test suites that have poor quality and are difficult to reuse. This is especially the case for code that requires structurally complex test inputs, for example code that operates on

programs (e.g., compilers, interpreters, model checkers, or refactoring engines) or on complex data structures (e.g., container libraries).

Consider, for example, testing a feature in a Java compiler. In this case, test inputs are representations of programs. Our intuition could be that we can expose bugs in the feature under test using as inputs, say, Java programs with complex *inheritance graphs* (consisting of classes and interfaces). The expected properties of the Java inheritance graphs are: there are no directed cycles, all explicit parents of interfaces are interfaces, and each class has at most one parent class. It is tedious and error-prone to manually generate tests that cover many "corner cases" for inheritance graphs: a test with only one class, a test with a subclass and a superclass, a test with two classes not related by inheritance, a test with an interface, a test with multiple inheritance through interfaces, a test with multiple paths to a common ancestor, with abstract classes, etc.

Recent techniques aim to reduce the burden of manual testing using systematic test generation based on specifications [16, 52] or on symbolic execution [24, 54] and its hybrids with concrete executions [20, 26, 32, 44, 53, 75, 83, 90]. Modern (hybrid) symbolic execution techniques can handle advanced constructs of object-oriented programs, but practical application of these techniques were largely limited to testing units of code much smaller than hundred thousand lines, or generating input values much simpler than representations of Java programs. The inherent requirement for not only building *path conditions*, albeit with partial constraints, but also determining their feasibility poses a key challenge for scaling to structurally complex inputs and entire systems. Automatically handling programs of the *complexity of a compiler* remains challenging for current systematic approaches. Our approach is to allow testers to utilize their domain knowledge to scale these systematic approaches.

We propose a new technique to generate a large number of complex test inputs by allowing the tester to write a *test generation program* in UDITA, a Java-based language with non-deterministic choices, including choices used to generate linked data structures. Each execution of a test generation program generates one test input. Our execution engine systematically explores all executions to generate inputs for *bounded-exhaustive testing* [69, 87] that validates the code under test for *all test inputs* within a given bound (e.g., all trees with up to N nodes). UDITA thus enables testers to avoid manual generation of individual tests. However, our approach does not attempt to fully automatically identify tests [20, 44], because such approaches do not provide much control to the tester to encode their intuition. Instead, we provide testers with an expressive language in which they have sufficient control to define the space of desired tests.

This chapter makes several contributions:

- New language for describing tests: We present UDITA, a language that enhances Java with two important extensions. The first extension are *non-deterministic choice* commands and the *assume* command that (partially) restricts these choices. These con-

structs are familiar to users of model checkers such as Java PathFinder (JPF) [94]. Thanks to the built-in non-determinism, writing a test generation program (from which *many* test inputs can be generated) is often as simple as writing Java code that generates *one* particular test input. The second extension is the *object pool* abstraction that allows the tester to control generation of linked structures with any desired sharing patterns, including trees but also DAGs, cyclic graphs, and domain-specific data structures. Due to its expressive power, UDITA enables testers to write test generation programs using any desirable mixture of two styles—*filtering* (also previously called declarative) [16, 35, 42, 52, 68, 69] and *generating* (also previously called imperative) [28, 49]—whereas previous systems required the use of only one style.

- New test generation algorithms: We present efficient techniques for test generation by systematic execution of non-deterministic programs. Our techniques build on systematic exploration performed by explicit-state model checkers to obtain the effect of bounded-exhaustive testing [69, 87]. The efficiency of our techniques is based on a general principle of *delayed choice* [73], i.e., lazy non-deterministic evaluation [38]. The basic delayed choice technique postpones the choices for each variable until it is first accessed. The more advanced *copy propagation* technique further postpones the choices even if the values are being copied. Like lazy evaluation, our techniques guarantee that each non-deterministic choice is executed at most once.

  Our techniques support primitive fields but are particularly well-suited for linked structures (Section 4.4.2). The techniques use a new *object pool* abstraction. We postpone the choice of object identity until object's first non-copy use, reducing the amount of search. Furthermore, we avoid isomorphic structures [48, 68] which gives another source of exponential performance improvement. Finally, to determine the feasibility of symbolic fresh-object constraints in the current path, we use a new polynomial-time algorithm (figures 4.10 and 4.11), which is in contrast to NP-hard constraints in traditional symbolic execution [24, 54], and it follows a design choice to shift the burden from a constraint solver to the optimized execution engine [46].

- Implementation: We describe an implementation of UDITA and our optimizations on top of JPF [94], a popular model checker for Java, which makes it easy to provide UDITA as a library. Our code is publicly available [9] and included as an extension (called `delayed`) in the (old) JPF repository:

  http://javapathfinder.sourceforge.net

- Evaluation: We evaluate our implementation on a number of complex test abstractions. We show that the new test abstractions can be shorter and lead to faster test generation. The experiments revealed bugs in Eclipse, NetBeans, and JPF.

  We have performed several sets of experiments to evaluate UDITA, mostly for black-box testing. The *first set* of experiments, on six data structures, shows that our optimizations improve the time to generate test inputs up to a given bound.

The *second set* of experiments is on testing refactoring engines, which are software development tools that take as input program source code and refactor (transform) it to change its design without changing its behavior [74]. Modern IDEs such as Eclipse or NetBeans include refactoring engines for Java. A key challenge in testing refactoring engines is generating input programs. Figures 4.5 and 4.6 show some example programs with multiple inheritance that revealed bugs in Eclipse. To generate such programs, we need to both "generate inheritance graphs" and "add methods" in the classes and interfaces in the graphs. Our experience with UDITA's combined filtering/generating style shows that, compared to our prior approach, ASTGen [28, 49], UDITA is more expressive, resulting in shorter (and easier to write) test generation programs, with sometimes faster generation (even on a slower JPF virtual machine). Through these experiments, we revealed four bugs in Eclipse and NetBeans (all four have been confirmed by developers and assigned to be fixed), and even two bugs in the Sun Java compiler.

The *third set* of experiments, on testing parts of the UDITA implementation, revealed several new bugs in JPF, and one bug in our JPF extension that we subsequently corrected. These results suggest that UDITA is effective in helping detect real bugs in large code bases.

The *fourth set* of experiments, for white-box testing, compared UDITA with Pex [90], a state-of-the-art testing tool based on symbolic execution. Our results found that object pools are a powerful abstraction for guiding exploration, orthogonal to the path-bounding approaches used by tools such as Pex. In particular, even a naive implementation of object pools helped Pex enumerate structures and find bugs faster. Our experimental results are publicly available [9].

4.2 Example

To illustrate UDITA, we consider generation of inheritance graphs for Java programs. Such generation helps in testing real-world applications including compilers, interpreters, model checkers, and refactoring engines (Section 4.5). The example illustrates how UDITA can describe data structures with non-trivial invariants. Figure 4.1 shows a simple representation of inheritance graphs in Java. A graph has several nodes. Each node is either a class or an interface, and has zero or more supertypes that are classes or interfaces. (We do not explicitly model the `java.lang.Object` class.)

Specification of inheritance graphs. Each inheritance graph needs to satisfy the following two properties: 1) **DAG** (directed acyclic graph): The nodes in the graph should have no directed cycle along the references in `supertypes`. 2) **JavaInheritance**: All supertypes of an interface are interfaces, and each class has at most one supertype class.

UDITA allows the tester to express these properties using full-fledged Java code extended with non-deterministic choices. Testers describe properties in UDITA using any desired mix of *filtering* and *generating* style. In a purely filtering style, embodied in techniques such as

```
class IG {
  Node[] nodes;
  int size;
  static class Node {
    Node[] supertypes;
    boolean isClass;
  }
}
```

Figure 4.1: A representation of inheritance graphs

```
boolean isDAG(IG ig) {
  Set<Node> visited = new HashSet<Node>();
  Set<Node> path = new HashSet<Node>();
  if (ig.nodes == null || ig.size != ig.nodes.length) return false;
  for (Node n : ig.nodes)
    if (!visited.contains(n))
      if (!isAcyclic(n, path, visited)) return false;
  return true;
}

boolean isAcyclic(Node node, Set<Node> path, Set<Node> visited) {
  if (path.contains(node)) return false;
  path.add(node);
  visited.add(node);
  for (int i = 0; i < supertypes.length; i++) {
    Node s = supertypes[i];
    // two supertypes cannot be the same
    for (int j = 0; j < i; j++)
      if (s == supertypes[j]) return false;
    // check property on every supertype of this node
    if (!isAcyclic(s, path, visited)) return false;
  }
  path.remove(node);
  return true;
}

boolean isJavaInheritance(IG ig) {
  for (Node n : ig.nodes) {
    boolean doesExtend = false;
    for (Node s : n.supertypes)
      if (s.isClass) {
        // interface must not extend any class
        if (!n.isClass) return false;
        if (!doesExtend) {
          doesExtend = true;
          // class must not extend more than one class
        } else { return false; }
      }
  }
}
```

Figure 4.2: Filtering approach for inheritance graphs

TestEra [52] and Korat [16, 35, 42, 68, 69], the tester writes the *predicates—what* the test inputs should satisfy; then the *tool searches* for valid tests. In contrast, in a purely generating style, embodied in techniques such as ASTGen [28, 49], the tester directly writes *generators—how* to generate valid inputs; then *the tool executes* these generators to generate the inputs. We first present these two pure approaches, then discuss how UDITA allows freely combining them, and finally how UDITA efficiently generates inputs.

```
IG initialize(int N) {
  IG ig = new IG();
  ig.size = N;
  ObjectPool⟨Node⟩ pool = new ObjectPool⟨Node⟩(N);
  ig.nodes = new Node[N];
  for (int i = 0; i < N; i++) ig.nodes[i] = pool.getNew();
  for (Node n : nodes) {
    // next 3 lines unnecessary when using generateDAGBackbone
    int num = getInt(0, N − 1);
    n.supertypes = new Node[num];
    for (int j = 0; j < num; j++) n.supertypes[j] = pool.getAny();
    // next line unnecessary when using generateJavaInheritance
    n.isClass = getBoolean();
  }
  return ig;
}

static void mainFilt(int N) {
  IG ig = initialize(N);
  assume(isDAG(ig));
  assume(isJavaInheritance(ig));
  println(ig);
}

static void mainGen(int N) {
  IG ig = initialize(N);
  generateDAGBackbone(ig);
  generateJavaInheritance(ig);
  println(ig);
}
```

Figure 4.3: Examples of bounded-exhaustive generation

Filtering approach. To generate complex test inputs such as inheritance graphs, one approach [16] is to use arbitrary (Java) code to write *predicates* that encode properties of test inputs. We call this style *filtering* as it only specifies *what* the inputs look like, although the specification itself is written in an imperative language [68]. Figure 4.2 shows Java predicates that return `true` when the above inheritance graph properties hold. Among the advantages of such test abstractions is that the developers need not learn an entirely new specification language and can choose to hand-optimize the checks using property-specific algorithms (such as the recursive algorithm for **DAG** in Figure 4.2), while the compiler can also use standard techniques to optimize the code.

To generate *all* test inputs from predicates, the tester needs to specify bounds on possible values for input elements, which in our example are the nodes, array sizes, and `isClass` fields. For this purpose, UDITA uses *non-deterministic choices*. JPF already has choices for primitive values. For example, the assignment `k=getInt(1, N)` introduces N branches in a non-deterministic execution, where in branch i (for $1 \leq i \leq N$) the variable `k` has value i. JPF can systematically explore all (combinations) of non-deterministic choices. UDITA

additionally provides non-deterministic choices for pointers/objects through the notion of *object pools* (described in detail in Section 4.4.2). Figure 4.3 shows the non-deterministic initialization of an inheritance graph data structure. The method `initialize` proceeds in several steps: (1) sets the graph size (the number of nodes), (2) creates a *pool* of `Node` objects of this size, and (3) iterates over all objects in the pool to non-deterministically initialize their `supertypes` to point to other objects in the pool. The `getNew` and `getAny` methods pick a fresh object and an arbitrary object from the pool, respectively. Running `mainFilt` on JPF/UDITA generates all inheritance graphs of size N.

```
void generateDAGBackbone(IG ig) {
  for (int i = 0; i < ig.nodes.length; i++) {
    int num = getInt(0, i); // pick number of supertypes
    ig.nodes[i].supertypes = new Node[num];
    for (int j = 0, k = -1; j < num; j++) {
      k = getInt(k + 1, i - (num - j));
      // supertypes of "i" can be only those "k" generated before
      ig.nodes[i].supertypes[j] = ig.nodes[k];
    }
  }
}

void generateJavaInheritance(IG ig) {
// not shown imperatively because it is complex:
// topologically sorts "ig" to find what nodes can be classes or interfaces
}
```

Figure 4.4: Generating approach for inheritance graphs

Generating approach. Instead of generating possible graphs in Figure 4.3 and then filtering those that are not inheritance graphs, Figure 4.4 shows a simpler and a faster alternative that directly generates DAGs of size N with the `generateDAGBackbone` method. We say that Figure 4.4 presents a *generator* for DAGs, which is in contrast to the *predicate* `isDAG` in Figure 4.2. The generator establishes *by construction* that there are no directed cycles (because `supertypes` of a node `i` can only be nodes `k` that were generated before `i`).

Writing generators instead of predicates can dramatically speed up generation. However, using generators alone is fairly involved. Although it is relatively easy to write a generator for all *arbitrary* DAGs, it is non-trivial to eliminate isomorphic graphs (Section 4.4.2) or to properly label nodes as classes and interfaces (`generateJavaInheritance`). Properties of other data structures can be even harder to express as generators. For example, an entire research paper was devoted to efficient generation of red-black trees [13]. In comparison, filtering is often easier, anecdotally confirmed by the fact that even undergraduate students are able to write appropriate checks [69]. This trade-off justifies the need for optimized execution for predicate-based exploration but also asks for an approach to combine predicates and generators.

Unifying predicates and generators. UDITA makes combination of predicates and genera-

```
import java.util.List;
class A implements B, D {
  public List m(){
    List l=null;
    A a=null;
    l.add(a.m());
    return l;
  }
}

interface D { public List m(); }
interface B extends C { public List m(); }
interface C { public List m(); }
```

```
import java.util.List;
class A implements B, D {
  public List<List> m() {
    List<List<List>> l=null; //bug
    A a=null;
    l.add(a.m());
    return l;
  }
}

interface D { public List<List> m(); }
interface B extends C { public List<List> m(); }
interface C { public List<List> m(); }
```

Figure 4.5: InferGenericType bug in Eclipse: when the refactoring is applied on the input program (left), Eclipse incorrectly infers the type of `A.m.l` as `List<List<List>>`, which does not match the return type of `A.m`

tors possible because they are both expressed in a unified framework: systematic execution of non-deterministic choices. Consider the properties in our running example. For the **DAG** property, comparing Figure 4.4 and Figure 4.2, one could argue it is easier to write a generator than a predicate. However, for the **JavaInheritance** property, it is much easier to write a predicate than a generator. UDITA allows the tester to combine, for example, a generator for **DAG** with a predicate for **JavaInheritance**: one would write a new `main` that uses `generateDAGBackbone` and `assume(isJavaInheritance)`.

Test generation. After the tester writes some predicates and/or generators, it is necessary to execute them to generate the tests. JPF already provides an execution engine for `getInt` and `getBoolean` non-deterministic choices. Naive implementations of the object pool's `getNew` and `getAny` choices (whose use is shown in Figure 4.3) can be simply done with `getInt` (as discussed in Section 4.4.2). However, these naive implementations, which we call *eager* as they immediately return a value, result in a combinatorial explosion, e.g., `mainFilt` from Figure 4.3 for $N = 4$ does not terminate in an hour!

We provide more efficient implementations, which we call *delayed* as they postpone choices of primitive values (`getInt` and `getBoolean`) and additionally optimize exploration for object pools (`getAny` and `getNew`). For example, `mainFilt` from Figure 4.3 for $N = 4$ terminates in just 5.5 seconds with our delayed choice. Generating approach can be even faster than filtering search. Section 4.5.1 shows our experimental results for data structures. We evaluate mostly the combined filtering/generating style, since test programs are much easier to write than for purely generating style, and generation for purely filtering style is several orders of magnitude slower on basic JPF without delayed choice.

Section 4.5.1 shows our results for testing refactoring engines, where we built on the inheritance graph generator to produce Java programs as test inputs. Figures 4.5 and 4.6 show two example input programs, generated by UDITA, which found bugs in Eclipse, specifically in the InferGenericType and UseSupertypeWherePossible refactorings.

```
class A implements B {
  public A m() {
    A a = null;
    return a;
  }
}

interface B extends C { public B m(); }
interface C { public C m(); }
```

```
class A implements B {
  public C m() { // bug
    C a = null;
    return a;
  }
}

interface B extends C { public B m(); }
interface C { public C m(); }
```

Figure 4.6: UseSupertypeWherePossible bug in Eclipse: when the refactoring is applied on `A`, the return type of `A.m` is incorrectly changed to `C` instead of displaying a warning or suggesting changing the return type to `B`

```
class ObjectPool⟨T⟩ {
  public ObjectPool⟨T⟩(int size, boolean includeNull) { ... }
  public T getAny() { ... }
  public T getNew() { ... }
}
```

Figure 4.7: Basic operations for object pools

```
interface IGenerator⟨T⟩ { T generate(); }

class IntGenerator implements IGenerator⟨int⟩ {
  int lo, hi;
  IntGenerator(int lo, int hi) { this.lo = lo; this.hi = hi; }
  int generate() { return getInt(lo, hi); }
}

class IGGenerator implements IGenerator⟨IG⟩ {
  IG ig;
  IGGenerator(int N) { ig = initialize(N); }
  IG generate() {
    assume(isDAG(ig) && isJavaInheritance(ig));
    return ig;
  }
}

class PairGenerator⟨L, R⟩ implements IGenerator⟨Pair⟨L, R⟩⟩ {
  IGenerator⟨L⟩ lg;
  IGenerator⟨R⟩ rg;
  PairGenerator(IGenerator⟨L⟩ lg, IGenerator⟨R rg) { ... }
  Pair⟨L, R⟩ generate() {
    return new Pair⟨L, R⟩(lg.generate(), rg.generate());
  }
}
```

Figure 4.8: UDITA interface for generators and some example generators

4.3 UDITA Language

UDITA language makes it easy to develop generic, reusable, and composable generators. The key aspects of the UDITA are: (1) constructs for generating primitive values and objects; (2) the

ability to encapsulate UDITA generators into reusable components using interfaces; and (3) the ability to compose these components.

Basic Generators. The generators for UDITA borrow from JPF non-deterministic choices for primitive values. For example, `getInt(int lo, int hi)` returns an integer between `lo` and `hi`, inclusively; and `getBoolean()` returns a boolean value. UDITA also provides a new notion, *object pools*, for non-deterministic choices of objects. Figure 4.7 shows the interface for object pools. The constructor can create finite (if `size > 0`) and infinite (if `size < 0`) pools, which may or may not include the value `null`. The method `getAny` non-deterministically returns any value from the pool (including optionally `null`), whereas `getNew` returns an object that was not returned by previous calls (and never `null`). Section 4.4.2 describes the implementation of these operations.

Generator Interface. UDITA provides `IGenerator` interface for encapsulating generators, as shown in Figure 4.8. The only method, `generate`, produces *one* object of the generic type `T`. During the execution on JPF, this method will be *systematically* explored for all non-deterministic choices, and will generate *many* objects of the type `T`. The figure also shows an example `IntGenerator` for primitive values (ignoring any boxing of primitive values needed in Java) and an example `IGGenerator` that encapsulates filtering style predicates (`isDAG` and `isJavaInheritance`).

The design of UDITA generators is influenced by ASTGen [28] (which provides Java generators for abstract syntax trees for testing refactoring engines) and QuickCheck [23] (which provides a Haskell framework for generators). UDITA provides a much simpler interface than ASTGen: instead of one method, the basic `IGenerator` for ASTGen has *five methods* [28, Sec. 3.2]. The cause of that complexity is that ASTGen runs on a deterministic language; to obtain bounded-exhaustive generation, the implementor of the interface must manually manipulate the generator state (to reset it, advance it, store/restore it). In contrast, UDITA supports non-determinism, with program execution enumerating all non-deterministic choices. Compared to QuickCheck [23], which supports only random generation, UDITA focuses on bounded-exhaustive generation, obtaining random generation for free as one of the possible exploration strategies of non-deterministic choices (where additional strategies include depth-first and breadth-first).

Composing generators. An important feature of frameworks such as ASTGen, QuickCheck, or UDITA is to allow reuse and composition of basic generators into more complex generators [23, 28]. UDITA again offers a substantially simpler solution than ASTGen. Figure 4.8 shows an example generator that produces pairs of values based on generators for left and right pair elements. Note that the `generate` method of `PairGenerator` has only one line of code. In contrast, the corresponding ASTGen generator has *ten lines of code* [28, Sec. 3.3]. The reason is, again, that ASTGen needs to explicitly iterate over possible values to produce their combinations for bounded-exhaustive generation. QuickCheck provides composition through higher-order functional combinators [23] but is designed for the purely functional language

Haskell and has no support for generating non-isomorphic graph structures. Neither ASTGen nor QuickCheck provide unified filtering/generating style like UDITA.

4.4 Test Generation in UDITA

We next describe our test generation algorithms, which rely on the notion of delayed (lazy) execution of non-deterministic choices.

4.4.1 Test Generation for Primitive Values

Eager choice execution. We could, in principle, use a straightforward implementation of `getInt` that *immediately* chooses a concrete value and returns it. When the execution backtracks, the implementation picks a different value. This approach allows us to easily obtain a baseline implementation on top of JPF. Unfortunately, the combinatorial explosion in typical test generation programs (e.g., the `initialize` method in Figure 4.3) causes this baseline implementation to explicitly consider a large number of unnecessary possibilities. We therefore use a more efficient and more complex approach that still preserves the simple non-deterministic semantics on which testers can rely.

Delayed choice execution. UDITA provides efficient test generation by extending JPF with lazy evaluation of non-deterministic choices [38, 73]. The key idea of delayed execution strategy is to delay the non-deterministic choices of values to the point where the values are used for the first time. Consequently, the order in which the values are used for the first time creates a dynamic ordering of the variables in the search space.

Algorithm for getInt. Our algorithm for delayed execution of `getInt` can be expressed as a program transformation that postpones branching in the computation tree generated by the program. The transformation extends the domain of variables so that it stores a pointer to a mutable cell c where c contains either 1) a concrete value as before, or 2) an expression of the form $\mathsf{Susp}(a,b)$, denoting the set of values $\{x \mid a \leq x \leq b\}$ from which a concrete value may be chosen in the future. A reference to $\mathsf{Susp}(a,b)$ corresponds to representations of delayed expressions in implementations of non-strict functional languages [38]. The transformation changes the meaning of x=$\mathsf{getInt}(a,b)$ to be lazy, storing only a symbolic representation (a,b) of possible values. We use statement $\mathsf{force}(x)$ to denote making an actual non-deterministic choice of the stored symbolic value of x. The algorithm inserts $\mathsf{force}(x)$ before the first non-copy use of the variable x, treating all variable uses other than copying as strict operations. Although in general both delayed and eager choice could explore exponentially many paths, in experiments we found exponential speedup when using delayed choice instead of eager choice (figures 4.12 and 4.17). Delayed choice provides speedup because it avoids exploring the values of variables not used in an execution that evaluates `assume(false)`.

```
class ObjectPool⟨T⟩ {
  ArrayList⟨T⟩ allocated;
  int maxSize;
  ObjectPool⟨T⟩(int size) {
    allocated = new ArrayList⟨T⟩();
    maxSize = size;
  }
  T getAny() {
    int i = getInt(0, allocated.size());
    if (i < allocated.size()) return allocated.get(i);
    else return getNew();
  }
  T getNew() {
    assume(allocated.size() < maxSize);
    T res = new T();
    allocated.add(res);
    return res;
  }
}
```

Figure 4.9: Eager implementation of object pools

4.4.2 Test Generation for Linked Structures

Eager implementation. Figure 4.9 presents a Java-like pseudo code for an eager implementation of object pools. We focus here on implementation of object pools of finite size that return non-`null` objects only. Our implementation also handles the (straightforward) extensions with unbounded object pools and possibly-`null` objects.

Isomorphism avoidance. An important issue in generating object graphs is to avoid structures that are isomorphic due to the abstract nature of Java references [16, 48]. For instance, DAGs that have the same structure but differ in the identity of nodes are isomorphic. In a purely generating approach, the control of isomorphism is up to the tester and not UDITA. (Indeed, the code in Figure 4.4 generates isomorphic DAGs.) In a filtering approach that uses the `getAny` method from object pools, UDITA automatically avoids isomorphic structures, like Korat [16]. The implementation in Figure 4.9 avoids isomorphism by returning only the first fresh object (rather than several different fresh objects).

Delayed execution implementation. The eager implementation in Figure 4.9 serves as a reference for our delayed choice implementation. The delayed choice implementation results in exploring the equivalent set of states as the reference implementation but does so much more efficiently. The high-level idea of delayed execution is the same as for `getInt`, but the implementation for object pools is more complex because `getNew` is a command that changes the state (the `allocated` set). As a result, simply creating a suspension around the methods from Figure 4.9 would *not* preserve the semantics because the side effects on the `allocated` set would occur in a different order.

To preserve the set of reachable states of the eager implementation, our implementation intro-

duces symbolic values at each call to `getNew` or `getAny` and also accumulates the constraints imposed by the requirement that `getNew` returns objects *distinct* from previously returned objects. When the program uses symbolic objects (doing a `force` of the value), UDITA assigns a concrete object to the symbolic object, ensuring that the accumulated constraints on distinct objects are satisfied. UDITA also ensures that it will be possible to instantiate the remaining symbolic objects while satisfying all the constraints. In the terminology of symbolic execution [54], UDITA maintains an efficient representation of the path condition, which expresses that certain symbolic objects are distinct, and ensures that the path condition is satisfiable. To see the non-triviality of our path conditions, consider this example with an object pool of size 3:

```
p = new ObjectPool<Node>(3);
n1 = p.getNew();
a1 = p.getAny();
a2 = p.getAny();
a3 = p.getAny();
n2 = p.getNew();
n3 = p.getNew();
use(a1);
use(a2);
use(a3);
```

The delayed execution will pick the concrete values of `a1`, `a2`, `a3` only at their use points. When it picks the values, it must have enough information to deduce that all values `a1`, `a2`, `a3` must be equal; otherwise, it will be impossible, in the pool of size 3, to assign values `n2`, `n3` such that `n2` $\notin$ {`n1`, `a1`, `a2`, `a3`} and `n3` $\notin$ {`n1`, `a1`, `a2`, `a3`, `n2`}.

Figures 4.10 and 4.11 show the pseudo-code of the desired delayed execution algorithm for object pools, implemented in UDITA. Type `List`$\langle C \rangle$ denotes an indexable linked list (such as Java `ArrayList`) storing objects of type C. Type `Sym`$\langle T \rangle$ denotes a symbolic variable, whose `chosen` field denotes concrete field (and is `null` if the concrete object is not chosen yet). The methods `getAny` and `getNew` from Figure 4.10 introduce a new symbolic variable and store it into the appropriate position in the two-dimensional `levels` data structure; `getAny` stores the symbolic variable at the current level, whereas `getNew` starts a new level. This structure encodes, for `j < i` and for all applicable `k`, that

```
levels.get(i).get(0).chosen != levels.get(j).get(k).chosen.
```

The `force` method from Figure 4.11 picks a concrete value for a given symbolic variable by respecting the recorded constraints. After selecting in the `candidate` variable the set of objects to which the symbolic variable could be made equal to, it either 1) selects one of these objects or 2) introduces a new concrete object. Finally, it recomputes the minimal size of the model under the current constraints, ensuring that the current choice of variables is satisfiable in the pool of the given size. Note that, although the problem has the flavor of the NP-complete graph coloring problem, the structure of our constraints (building levels in layers) allowed us to design the efficient test in the `findMinModelSize` method.

```
class Sym⟨T⟩ { // symbolic variable
  T chosen;
  int level;
  boolean isGetNew;
  Sym⟨T⟩(int level, boolean isGetNew) { ... }
}

class ObjectPool⟨T⟩ {
  List⟨T⟩ allChosen;
  List⟨List⟨Sym⟨T⟩⟩⟩ levels;
  int maxSize, lastLevel, minModelSize;
  ObjectPool(int size) {
    allChosen = new List⟨T⟩();
    levels = new List⟨List⟨Sym⟨T⟩⟩⟩();
    maxSize = size;
    lastLevel = −1;
    minModelSize = 0;
  }
  Sym⟨T⟩ getAny() {
    if (lastLevel < 0) return getNew();
    sym = new Sym⟨T⟩(lastLevel, false);
    levels.get(lastLevel).add(sym);
  }
  Sym⟨T⟩ getNew() {
    lastLevel++;
    newLevel = new List⟨Sym⟨T⟩⟩();
    levels.add(newLevel);
    sym = new Sym⟨T⟩(lastLevel, true);
    newLevel.add(sym);
    minModelSize++;
    assume(minModelSize <= maxSize);
  }
}
```

Figure 4.10: Delayed execution for object pools: data structures, getAny, getNew

Correctness proof. The correctness of our algorithm can be shown by viewing it as an efficient implementation of a symbolic execution with disequality constraints. The only subtle part is showing that the `findMinModelSize` method from Figure 4.11 correctly computes the size of the *smallest* model of the equality and disequality constraints imposed by current symbolic variables and any concrete values assigned to them. The correctness can be shown by considering the iteration I in which `minModelSize` reaches its maximum. The concrete nodes at levels up to I together with any `getNew` nodes at higher levels must all be distinct, so each model is at least of size `minModelSize`. Conversely, by a greedy assignment that favors previously chosen concrete objects, we can construct a model of size `minModelSize`.

The `levels` data structure encoding a path condition of the form $\wedge_i(x_i \neq x'_i)$. The non-null `chosen` fields encode the condition $x_i = o_k$ for concrete objects o_k. Each object pools also has an implicit condition $|\{x_1,\ldots,x_n\}| \leq$ `maxSize` where $x_1,\ldots,x_n$ are all symbolic variables. Our implementation ensures this implicit condition by computing, in To show the correctness of this method, note that `levelModelSize` in iteration `i` computes the size of the

```
void force(Sym⟨T⟩ x) {
  if (x.chosen == null) {
    List⟨T⟩ candidates;
    if (x.isGetNew) {
      candidates = new List⟨T⟩();
      for (int i = x.level; i ≤ lastLevel; i++) {
        List⟨T⟩ currentLevel = levels.get(i);
        for (int j = 1; j < currentLevel.size(); j++)
          Sym⟨T⟩ s = currentLevel.get(j);
          if (s.chosen != null && !candidates.contains(s.chosen))
            candidates.add(s.chosen);
      }
    } else { // x created by getAny
      candidates = new List⟨T⟩(allChosen);
      for (int i = x.level+1; i ≤ lastLevel; i++) {
        Sym⟨T⟩ s = levels.get(i).get(0); // getNew
        if (s.chosen != null) candidates.remove(s.chosen);
      }
    }
    int choice = getInt(0, candidates.size());
    if (choice < candidates.size())
      x.chosen = candidates.get(choice);
    else {
      x.chosen = new T();
      allChosen.add(x.chosen);
    }
    findMinModelSize();
    assume(minModelSize <= maxSize);
  }
}

void findMinModelSize() {
  List⟨T⟩ chi = new List⟨T⟩();
  minModelSize = lastLevel;
  for (int i = 0; i ≤ lastLevel; i++) {
    foreach (Sym⟨T⟩ s in levels.get(i))
      if (s.chosen != null && !chi.contains(s.chosen))
        chi.add(s.chosen);
    int levelModelSize = chi.size() + lastLevel − i;
    minModelSize = max(minModelSize, levelModelSize);
  }
}
```

Figure 4.11: Picking a concrete object for symbolic variable of object pool in delayed execution

model consisting of 1) the `chi.size()` already allocated objects in levels up to `i`, and 2) the `lastLevel-i` objects that need to be chosen as values of **getNew** variables at levels strictly above `i`. All of these objects must be distinct, so the smallest model must have at least the sum of `minModelSize` elements, for each value of variable `i`.

Having shown that the computed value is the lower bound on the minimal model size, we next show the converse, by describing a construction of a model of size `minModelSize`. It suffices to specify the choice of concrete objects for **getNew** variables (stored in `levels.get(i).get(0)`)

that are not yet chosen: the remaining **getAny** variables can always be chosen equal to the **getNew** variables at the same level. Let us remove from the constraints all concrete objects already chosen by a **getNew** variable, and remove all **getAny** variables to which they are assigned. We use a greedy algorithm to choose the remaining **getNew** variables from level 0 to `lastLevel`. We choose either 1) a concrete **getAny** object at a higher or equal level, or, if there are not sufficiently many of those, 2) an additional fresh object. Let I be the largest level at which `levelModelSize` reaches maximum greater than `lastLevel`, and let C be the value of `chi.size()` at this step. Then for all levels below I the assignment process had sufficiently many objects already assigned to **getAny** variables and did not need to use any fresh objects. The number of concrete objects assigned to variables up to level I is therefore C. When the assignment process continues at levels above I, then all **getAny** objects are used up (if there were some left, the value I would not be the point of maximal `levelModelSize`). Consequently, the number of additional distinct objects at levels above I is exactly `lastLevel` $-$ `I`. The total number of objects is therefore C + `lastLevel` $-$ `I`, which is exactly the value of `levelModelSize` when it reaches maximum. This shows that the constructed model has the size `minModelSize` computed by `findMinModelSize`.

4.4.3 Benefits of Object Pools

Specification advantage. Previous work on symbolic execution (e.g., CUTE [83]) uses equality and disequality constraints on *individual* object references (`==` and `!=`). Our work introduces the new object pool abstraction, which allows testers to conveniently express "freshness" disequality constraints of one reference against *all* references from a given user-defined set.

Algorithmic advantage. Note that an attempt to encode object pool constraints using equalities and disequalities over individual symbolic variables typically results in constraints whose satisfiability is *NP-hard*. In particular, consider a straightforward encoding of the constraint $|\{x_1,\ldots,x_n\}| \leq$ `maxSize` on the maximal size of object pool. The encoding would introduce `maxSize` fresh constants $a_1,\ldots,a_{\texttt{maxSize}}$ denoting distinct references and require

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{\texttt{maxSize}} x_i = a_j$$

Such encoding thus introduces non-trivial disjunctions into the problem. In contrast, we developed a *polynomial-time* algorithm to test the satisfiability of object pool constraints.

4.5 Evaluation

We implemented UDITA by modifying Java PathFinder (JPF) version 4. The key changes were our delayed choice algorithms and object pools. We implemented them using JPF's attribute mechanism [75] to store non-deterministic values that have not been read yet. We correspondingly modified the implementation of `getInt` to generate such delayed values. We

			JPF Baseline		Delayed Choice	
program	N	structs	time [s]	explored	time [s]	expl.
DAG	3	34	11.14	4802	2.19	321
	4	2352	o.o.m.	-	12.42	21196
	5	769894	o.o.m.	-	1673.91	4997210
HeapArray	6	13139	29.00	160132	12.50	27664
	7	117562	407.45	2739136	49.20	227494
	8	1005075	7892.88	54481005	417.70	2325069
NQueens	6	4	13.81	46656	1.82	746
	7	40	170.82	823543	3.60	3073
	8	92	3416.38	16777216	6.50	13756
RBTree	6	20	5.91	8448	5.79	3588
	7	35	21.76	54912	8.20	16983
	8	64	107.49	366080	22.27	80470
SearchTree	4	490	5.00	3584	2.26	1484
	5	5292	27.49	131250	8.29	21210
	6	60984	1810.93	6158592	60.67	305052
SortedList	6	924	11.70	55987	5.10	3967
	7	3432	126.14	960800	6.92	18026
	8	12870	2495.49	19173961	17.87	80089

Figure 4.12: Enumeration of structures satisfying their invariants ("o.o.m." means "out of memory")

also implemented object pools as described in Section 4.4.2. Our code is publicly available [9].

We performed several experiments to evaluate UDITA. UDITA is most applicable for black-box testing. The first set of experiments, on six data structures, compares delayed choice with base JPF for bounded-exhaustive test generation. The second set of experiments, on testing refactoring engines, compares UDITA with ASTGen [28]. The third set of experiments uses UDITA to test parts of the UDITA implementation itself. UDITA can be also used for white-box testing. The fourth set of experiments compares UDITA with symbolic execution in Pex [90]. We ran the experiments on an AMD Turion 1.80GHz laptop with Sun JVM 1.6.0_12, Eclipse 3.3.2, NetBeans 6.5, and Pex 0.19.41110.1.

4.5.1 Black-Box Testing

Generating Data Structures

We present an evaluation of delayed choice using a variety of data structure implementations: `DAG` represents directed acyclic graphs related to the example introduced in Section 4.2; `HeapArray` is an array-based heap data structure; `RBTree` is red-black tree; `SearchTree` is binary search tree; and `SortedList` is a doubly-linked list containing sorted elements. Additionally, `NQueens` is the traditional problem from constraint solving [12]. For each structure, we wrote its representation invariant using our combined filtering/generating style. Our experimental setup compares base JPF against JPF extended with our delayed choice execution, using the same test generation program. We turn off JPF state hashing in our experiments, because duplicate states rarely arise in executions of our examples [42].

Enumerating structures. Figure 4.12 shows the efficiency of our approach for structure enumeration. For each program and several bounds N, we tabulate the total number of successful

paths in the exploration tree (i.e., the number of structures generated), the exploration time, and the total number of explored paths. JPF generates the same number of structures with and without delayed choice, but delayed choice explores fewer paths than the base JPF, providing significant speed-ups, from 2x up to 500x as size increases.

Summary. The results show that delayed choice significantly improves the time to enumerate test inputs up to a given bound.

Testing Refactoring Engines

We applied UDITA to generate Java input programs for testing refactoring engines as briefly described in Section 4.2 and as previously done with ASTGen [28, Sec. 5]. Since the inputs are generated automatically, the outputs are validated using programmatic oracles such as checking for refactoring engine crashes, obtaining non-compilable output programs, or getting different outputs for Eclipse and NetBeans (known as differential testing [70]). We perform two kinds of experiments: (1) rewriting some existing ASTGen generators in UDITA to compare the ease of writing generators and the efficiency of generation, and (2) writing new generators that would be very difficult to express in ASTGen.

Rewriting ASTGen generators. We rewrote five (randomly chosen, advanced) ASTGen generators in UDITA. Figure 4.13 shows the results. The generators in UDITA have fewer lines of code ("LOC", which includes the top-level generator and the library it uses) and are, in our experience, often easier to write. UDITA conceptually subsumes ASTGen, so we could not find a case where UDITA code would be more complex than ASTGen code. UDITA generators are about as efficient as ASTGen generators—sometimes a bit faster, and sometimes a bit slower—which was quite surprising to us at first: ASTGen runs on top of a regular JVM, while UDITA runs on top of JPF, and JPF can be two orders of magnitude slower than JVM. We did expect UDITA generators to be easier to write but not to be faster, at least not without special optimizations [42]. Our investigation shows that UDITA can be faster for two reasons: (1) it has a faster backtracking due to JPF's storing and restoring of states rather than the re-execution of code in ASTGen, and (2) combined filtering/generating style for iteration/generation allows more efficient positioning of backtracking points (UDITA need not build an entire input before realizing that backtracking is required).

Writing new generators. We wrote three new generators in UDITA that would be extremely difficult to write in ASTGen. All these generators use inheritance graphs which, as discussed in Section 4.2, are much easier to express by combining filtering and generating styles. UDITA is more expressive than ASTGen since UDITA allows natural mixing of these two styles. These generators allowed us to test some refactorings we did not test with ASTGen (UseSupertypeWherePossible, which replaces one class/interface with its superclass/superinterface where possible, and InferGenericType, which finds the most appropriate generic type parameters for raw types [91]) and to more thoroughly test a refactoring we did test (RenameMethod).

		ASTGen		UDITA	
generator	inputs	time [s]	LOC	time [s]	LOC
2ClsMethParent	2160	492.87	1316	117.92	835
3ClsMethChild	1152	265.19	1342	89.17	848
2ClsMethChild	576	135.34	1320	44.01	822
2Cls2FldChild	540	1.13	713	36.96	389
2Cls2FldRef	240	2.62	714	27.96	430

Figure 4.13: Comparing ASTGen and UDITA

			Eclipse		NetBeans	
refactoring	time [s]	inputs	fail	bug	fail	bug
RenameMethod	105.15	207	0	0	75	1
UseSupertypeWP	85.80	402	59	1	7	1
InferGenericType	258.55	414	171	1	n/a	n/a

Figure 4.14: Refactorings tested and bugs found

Figure 4.14 shows the results. We revealed four bugs in Eclipse and NetBeans, two of which are shown in figures 4.5 and 4.6. As can be seen from the table, the number of failing tests is much larger than the number of (unique) bugs; we used our oracle-based test clustering technique [49] to inspect the failures.

Differential testing of compilers. While testing the refactoring engines, we effectively used the same input programs to also perform differential testing of the Sun javac (version 1.6.0_10) and Eclipse (version 3.3.2) Java compilers. This revealed two differences, which are likely bugs in the Sun javac compiler as it incorrectly rejects valid programs accepted by the Eclipse compiler. We had reported these bugs to Sun, but they were not confirmed as of this writing.

Summary. The combined filtering/generating style in UDITA is better than purely generating style in ASTGen: UDITA is more expressive, results in shorter (and easier to write) test generation programs, and, in some cases, even provides faster generation (despite running on JPF, which is much slower than JVM). We found several new bugs with UDITA; details of all the bugs are online [9].

Testing JPF and UDITA

We also applied UDITA to generate Java input programs for testing parts of UDITA itself. Specifically, we used differential testing [70] to check (1) whether (base) JPF correctly implements a JVM, and (2) whether our delayed choice implementation behaves as non-delayed choice.

Testing JPF. JPF is implemented as a specialized JVM that provides support for state exploration of programs with non-deterministic choices [94]. For programs without non-deterministic choices, JPF should behave as a regular JVM. We knew from our experience with JPF that it does not always behave as JVM, especially for some standard libraries (e.g., related to reflection or native methods) or for the latest Java language features (e.g., annotations or enums). We wrote generators to produce small Java programs that exercise these libraries/features. We also wrote a generic driver that would compile each generated program, run it on JPF

generator	time [s]	inputs	failures	bugs
AnnotatedMethod	31.28	1280	0	0 (2)
ReflectionGet	23.71	160	80	1
DeclaredMethods	7.91	64	0	0
DeclaredMethodReturn	41.07	288	32	1
ReflectionSet	26.97	160	32	1
NotDefaultAnnotatedField	48.53	1760	0	0
Enum	1.67	78	0	0
ConstructorClass	12.01	387	27	1 (4)
DeclaredFieldTest	14.38	180	12	1
ClassCastMethod	27.96	102	75	1

Figure 4.15: Generators for testing JPF; bugs in parentheses were found in an older JPF version (revision 954)

and JVM, and compare the outputs from the two. Figure 4.15 shows the results. Through this process, we found eleven unique bugs in an older version of JPF (five of which were corrected in a more recent revision, 1829, from the JPF repository). Detailed results are online [9].

Testing delayed choice implementation. Although we proved that our delayed choice algorithm is correct, we still need to test its implementation, especially the challenging part of object pools (Section 4.4.2). We wrote a generator that produces Java programs with various sequences of `getAny` and `getNew` calls on an object pool (and then reads the returned values in various orders). We also wrote a script to compile each program and run it on JPF both with and without delayed choice. This process found a bug in our implementation (related to the computation of `levels` from Section 4.4.2) which occurred only for some sequences that mix between `getNew` calls a number of `getAny` calls exactly equal to the pool size. We subsequently corrected the bug and used the generator to increase our confidence in the corrected implementation.

Summary. The use of UDITA helped us identify a number of bugs in parts of the UDITA implementation.

4.5.2 White-Box Testing

UDITA is primarily designed for *black-box testing* [95]: UDITA executes test generation programs to create test inputs, and then those *inputs are run on the code under test as usual, without UDITA*. However, UDITA can be also applied for a limited form of *white-box testing* [95] by *executing the code under test itself on UDITA*. Note that UDITA does not use the information about the code under test, e.g., to increase syntactic coverage. Instead, UDITA executes the code just to speed up the full coverage of the specified, bounded region of the input space. Consider, for instance, using the following code, in Figure 4.16, to test the `remove` method from a red-black tree [9]. Generating *any* tree that fails the assertion reveals a bug. Figure 4.17 shows the effectiveness of our approach for revealing bugs. Eight bugs of omission were manually inserted into an implementation of `RBTree` by students not familiar with our work. For each bug, the first row is for Pex (Section 4.5.2). The second row is for purely filtering

```
static void main(int N) {
  RBTree t = new RBTree();
  t.initialize   (N);                    // Pick a graph
  assume(t.isRBT());                     // satisfying   invariant ,
  int v = getInt(0, N);                  // and pick a value.
  t.remove(v);                           // Run code under test,
  assert(t.isRBT());                     // and check invariant .
}
```

Figure 4.16: Testing the `remove` method

		UDITA Eager		UDITA Delayed		Pex
bug#	style N	time [s]	expl.	time	expl.	time
1	filter 1-*					22.05
	filter 1–4	timeout	-	1.89	799	14.49
	f/g 1–4	1.61	168	1.59	132	8.13
2	filter 1-*					timeout
	filter 1–6	timeout	-	12.37	16620	137.66
	f/g 1–6	10.26	7166	8.20	3163	89.34
3	filter 1-*					21.53
	filter 1–2	9.20	10710	0.70	27	9.83
	f/g 1–2	0.61	9	0.62	9	5.14
4	filter 1-*					8.93
	filter 1–3	timeout	-	0.84	136	7.10
	f/g 1–3	0.75	30	0.80	27	5.03
5	filter 1-*					24.65
	filter 1–3	timeout	-	1.12	151	12.45
	f/g 1–3	1.08	31	1.09	28	5.59
6	filter 1-*					4.55
	filter 1–1	0.50	1	0.47	1	4.69
	f/g 1–1	0.36	1	0.39	1	4.19
7	filter 1-*					2.72
	filter 1–1	0.53	1	0.53	1	4.99
	f/g 1–1	0.47	1	0.49	1	4.27
8	filter 1-*					12.50
	filter 1–4	timeout	-	1.58	676	22.95
	f/g 1–4	1.22	145	1.36	120	7.87

Figure 4.17: Time taken and structures explored to find the first bug in remove/put methods of red-black tree. "timeout" denotes time over 1 hour. "filter" denotes using purely filtering; "f/g" denotes combined filtering/generating style. UDITA requires bounds; "1–s" for N denotes the generation of all trees of sizes from 1 to *s*, where *s* is the smallest size that reveals the bug. Pex can also work without bounds (denoted "filter 1-*").

style (as in figures 4.2 and 4.3, with `initialize` using `getInt/getAny/getNew`), in which base JPF is extremely slow. The third row is for combined filtering/generating style, and delayed choice again outperforms base JPF for larger sizes.

Comparison with Symbolic Execution

Symbolic execution is a very active area of research, with a number of recent testing tools including Crest, CUTE, DART, DySy, EGT, EXE, KLEE, Pex, SAGE, Splat, JPF's Symbc. However, many of these tools are not publicly available and/or do not support symbolic references (either not at all or not with isomorphism avoidance). Pex [90] is a publicly available, state-of-the

art tool from Microsoft Research that supports symbolic references and avoids isomorphism. Pex is used for testing C#/.NET code. To solve path conditions, Pex uses Z3 [30], one of the very best constraint solvers (see http://www.smtexec.org).

We compared UDITA with Pex. To enable this comparison, we translated buggy red-black tree code from Java into C#. We also translated the (filtering) predicates and (generating) generators for red-black tree to C#. We used Pex, as UDITA, to test one method in isolation, `remove` or `put`. (An alternative is to test several methods at once through method sequences [27, 89, 96]). The predicate is required to specify pre- and post-condition for the method under test, in both Pex and UDITA. Note that Pex, unlike UDITA, does *not* require specifying bounds on the input size, but we wrote a simple, *eager* implementation of object pools in Pex/C# to be able to limit the search space for Pex.

Figure 4.17 shows the results. Pex times are averaged over five runs, because the results can differ as objects get allocated at different locations in different runs. Pex is able to quickly find all the bugs except that none of the five runs found bug2 in filtering mode with no object pools. Pex, like other tools based on symbolic execution, aims at *exploring paths* of the code under test (with the goal of increasing coverage to find bugs), unlike UDITA that is designed for *generating all test inputs of a given size* (bounded-exhaustive testing). We hypothesized that Pex misses bug2 because it requires a path with several repeated branches (resulting from loop unrolling) for a tree of size 6, while Pex aims at increasing branch coverage, thus giving less priority to repeated branches. Pex developers investigated bug2 and found that it is indeed missed because Pex's default exploration strategy does not give priority to paths that could find this bug.

However, when we ran Pex with object pools (even a simple, *eager* implementation), Pex was able to find bug2 in about 137 seconds. Despite these results, we do not expect UDITA by itself (concrete execution with object pools) to be better than Pex for white-box testing. Our view is that object pools are a powerful abstraction for guiding exploration, orthogonal to the path-bounding approaches used by tools such as Pex. We therefore expect tools like Pex to integrate object pools into their symbolic engines in the future, effectively implementing delayed choice for object pools. In addition to the current JPF implementation, UDITA can be implemented on top of other platforms, with similar costs and benefits: if the tester spends more time guiding the exploration, the tool may find some bugs faster.

4.6 Related Work

There is a large body of work on automated test generation. This chapter focuses on test generation programs, combining filtering [16, 35, 42, 52, 68, 69] and generating [28, 49] styles in a general-purpose programming language. Related work on topics such as specification-, constraint-, and grammar-based testing [59] is reviewed in more detail in a previous paper [28] and a PhD thesis [68]. The key technique that enables efficient test generation for UDITA is delayed execution, so we review here related work on that topic.

Noll and Schlich [73] proposed delayed non-deterministic execution for model checking assembly code. While their and our approaches share the name, the algorithms differ: UDITA precisely *shares* non-deterministic values that are copied, using lazy evaluation, whereas their approach [73] *copies* non-deterministic values, effectively using call-by-name semantics and over-approximating state space, possibly exploring executions that are infeasible in regular execution. Further differences stem from different abstraction levels, with UDITA modeling each non-deterministic integer as one symbolic value as opposed to a set of bits, and UDITA handling graph isomorphism for allocated objects.

Techniques similar to delayed choice execution are common in constraint solving—for constraints written in both imperative and declarative languages. For example, Korat [16] implicitly uses delayed choice by monitoring field accesses and using them in field initializations for the new candidates it explores. Generalized symbolic execution [53] uses "lazy initialization" to make non-deterministic field assignments on first access. Deng et al.'s [32] "lazier initialization" builds on generalized symbolic execution and makes non-deterministic field assignments on first use. Visser et al. [95] use preconditions written in Java for checking satisfiability but require the users to provide "conservative preconditions" which are hard to provide manually or generate automatically. A key difference between previous work and work in this chapter is that we provide a generic framework that supports delayed choice execution for arbitrary Java code extended with non-deterministic choices for primitive values and objects. We also apply UDITA on testing much larger code bases, finding bugs in Eclipse, NetBeans, Sun javac, and JPF.

The ECLiPSe constraint solver [12] provides a constraint logic programming (CLP) interface for writing declarative constraints. ECLiPSe provides *suspensions* that delay testing of predicates until more information is available. Researchers have proposed translating imperative programs into CLP engines [39] but faced limitations of current CLP implementations. We believe that non-deterministic extensions of popular programming languages such as Java can lead both to advances of software model checking and to scalable implementations of constraint solvers.

Approaches to automated test generation includes those based on exploration of method sequences for generation of object-oriented unit tests [27, 89, 96]. Such exploration cannot be used to generate complex test inputs when there are no appropriate methods, e.g., for building inheritance graphs. UDITA can directly generate complex test inputs, and generators in UDITA can even use method sequences.

Unlike symbolic execution [24, 54], UDITA relies mostly on concrete execution to generate test inputs, and uses a polynomial-time algorithm (Section 4.4.2) to ensure the feasibility of the currently explored path. This is in contrast to traditional symbolic execution where path conditions belong to NP-hard logics (often containing propositional logic, uninterpreted functions, and bitvector arithmetic). Several recent approaches show promising results by combining symbolic with concrete execution [20, 26, 44, 45, 75, 83, 90] or with grammar-based

input generation [43]. In contrast to combination of concrete executions with abstraction [14, 76], UDITA focuses on test generation by efficiently covering a set of concrete executions, without approximation. UDITA is most applicable for black-box testing as shown by finding bugs in Eclipse, NetBeans, Sun javac, and JPF. However, UDITA requires/allows the tester to manually guide the exploration. Tools based on symbolic execution are more automated and better than UDITA for white-box testing. Our experience with Pex suggests that other tools can benefit by incorporating object pools from UDITA.

4.7 Conclusions

We have found UDITA to be an expressive and convenient framework for specifying complex test inputs. Because it extends Java, it has the expressive power and the appeal of a full-fledged programming language. Because it contains non-deterministic constructs, it is appropriate for describing tests in a wide range of styles, from predicates that indicate properties, to generators that create only desired structures. To describe linked structures, we have found the new object pool abstraction to be particularly helpful. We have found UDITA easier to use than previous frameworks.

UDITA quickly revealed bugs in data structure implementations, and was effective in systematically generating structures up to a given size. The effectiveness of UDITA was in large part due to our lazy evaluation technique for non-deterministic choices, and the algorithms for delayed execution of object pool operations without solving NP-hard constraints. We have applied UDITA to real-world software and uncovered previously unknown bugs in Eclipse, NetBeans, Sun javac, and Java PathFinder.

5 Conclusions

In this dissertation we have presented three systems (InSynth, PolySynth and anyCode) that synthesize program fragments, and the system (UDITA) that consists of a Java-like language that helps a user to describe a set of test inputs and the algorithm that efficiently generates test inputs. All systems incorporate a set of techniques that help them efficiently and effectively traverse large search space:

- For InSynth these techniques include: a succinct type representation that avoids unnecessary explorations for the same argument types; a backwards search that explores and uses only declarations reachable from a desired type; and a weights mechanism that guides the search and help rank the expressions.
- For PolySynth these techniques include: a resolution algorithm that systematically traverses the search space; the same weight mechanism used in InSynth; and filtering based on test case execution that filters out irrelevant expressions.
- For anyCode these techniques include: natural language processing techniques that reduce the textual input ambiguity; a declaration selection model that identifies a set of most likely declarations to appear in the final expressions; and unigram and probabilistic context free grammar statistical models that, like the weights mechanism in InSynth and PolySynth, steer the synthesis and help rank the generated expressions.
- For UDITA algorithm these techniques include: a delayed choice technique that reduces exploration (generation) time by delaying non-deterministic variable choices (assignments) as much as possible; and a technique that reduces a number of generated isomorphic structures.

We have also presented evaluation results that show effectiveness of our tools and techniques. The results show that InSynth can be used in an interactive setting to help developers synthesize desired expressions. The evaluation is conducted using 50 benchmarks based on the real-word examples that demonstrate proper usage of API. The results show that InSynth

returns the expected expression in 96% of the examples, in a short period of time. The evaluation results for anyCode, based on 60 examples, show that anyCode can generate 93% of expected expressions in a short period of time and that it greatly benefits from statistical models. Finally, we use evaluation to demonstrate that UDITA can efficiently and effectively generate test programs for Java compilers, refactoring engines (in Eclipse and NetBeans) and an early implementation of our UDITA algorithm. Using the generated programs we managed to discover numerous bugs in these tools, including some previously unknown.

We conclude this dissertation by highlighting future research directions of interest.

Complete synthesis algorithm for the system F. In the Chapter 2 we proposed two synthesis algorithms: the InSynth algorithm considers ground and function types and the PolySynth algorithm considers polymorphic types, and has limited support for function types. It would be interesting to combine the two algorithms and build a complete algorithm that supports the type system F [78, Section 23.8]. This is also important because modern program languages like Java, Scala and C# rely on type systems that support polymorphic and functional types.

Advanced code repair. It would be also interesting to build a more sophisticated algorithm that repairs broken expressions than the algorithm proposed in Chapter 3. Such an algorithm should use programming language symbols from a broken expression to build additional set of constraints. The constraints will encode declaration positions in the broken expression. We believe that combining such an algorithm with statistical models may lead to more useful solutions.

Program synthesis beyond expressions. We would also like to explore synthesis of code fragments that include programming language constructs beyond declarations, like local variable assignments, conditional statements, and loop statements. Primarily, we would like to use approaches, similar to ones proposed by A. Mishne et al. [72], that suggest temporal order among declarations, and combine them with our expression synthesis algorithm. We believe that such an approach may lead to efficient synthesis of code fragments whose complexity and size go up to complexity and size of method bodies.

Recommender system for program synthesis. To make the program synthesis systems more effective we would like to implement an engine that will learn directly from a user and her code. This includes collecting and reasoning about recently used snippets and other properties we can automatically extract from the user's code.

Compiler for the UDITA language. To speed up UDITA test input generation, it would be interesting to build a compiler for UDITA language. At the moment, UDITA test generation programs run on top of Java PathFinder (JPF), an interpreter that runs on top of a Java virtual machine (JVM). This slows down the execution of the UDITA code. To overcome this problem, we can build a compiler and translate UDITA code to a code, of an existing programming language, which can be executed fast. The challenge is to translate correctly non-deterministic choices, which appear in test generation programs, and which rely on backtracking mecha-

nism.

UDITA combined with symbolic execution. In Section 4.5.2 we built an eager UDITA support for Pex. The constrains that are built over variables with non-deterministic values are eagerly assigned to the variables. It would be interesting to further track the constrains along the execution path and simplify them when possible. We believe that this might lead to early search space pruning, and thus it can lead to faster test input generation.

The main goal of the systems, we have presented, is to automate processes developers and testers often perform during software development. They aim to improve users productivity during implementation, testing and software maintenance.

A Appendix

A.1 InSynth Algorithm Completeness Proof

In theorems and lemmas that follow we will assume that the listed statements below hold. Let Γ_o be lambda environment, Γ and Γ' succinct environments, $x_1, \ldots, x_n$ (fresh) variables with lambda types $\tau_1, \ldots, \tau_n$, S a set of succinct types, τ and υ lambda types and t a succinct type, then:

- $\Gamma = \sigma(\Gamma_o)$
- $S = \sigma(\{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = \{\sigma(\tau_1), \ldots, \sigma(\tau_n)\}$
- $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \upsilon$
- $t = \sigma(\tau) = \sigma(\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \upsilon) = S \rightarrow \upsilon$
- $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$
- $\Gamma' = \sigma(\Gamma'_o) = \sigma(\Gamma_o) \cup \sigma(\{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = \Gamma \cup S$.

Theorem 2.3.3 Let Γ_o be an original environment, e an lambda expression, $\tau \in \tau_\lambda(B)$ and functions RCN and $\mathscr{D}$ defined as above, then:

$$\Gamma_o \vdash_\lambda e : \tau \Leftrightarrow e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$$

Proof Let us first show the $\Rightarrow$ direction. We do this by induction on $\mathscr{D}(e)$.

Base case [$\mathscr{D}(e) = 1$]: then $e \equiv \lambda x_1 \ldots x_n.a$ and $\tau \equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \upsilon$. We have to prove $\lambda x_1 \ldots x_n.a \in \mathsf{RCN}(\Gamma_o, \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon, 1)$. Let us follow the RCN algorithm with corresponding arguments. First, we take the **else** branch because $d = 1$.

Our first goal is to show that $((\Gamma \cup S)@\emptyset : \nu) \in \mathsf{CL}(\Gamma, S \rightarrow \nu)$. By the CL definition this is equivalent to proving that $(\emptyset \rightarrow \nu) \in (\Gamma \cup S)$ and $\forall t' \in \emptyset . \Gamma \cup S \vdash_c t'$. The second statement is trivially **true**, and the first we reduce to $\nu \in (\Gamma \cup S)$ by the convention. From $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n . a : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \nu$ it follows that $\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda a : \nu$ by the Abs (LNF) rule. Using the App (LNF) rule we conclude that $a : \nu \in (\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = (\Gamma_o \cup S) = \Gamma'_o$. Now we apply σ and get $\sigma(a : \nu) \in \sigma(\Gamma_o \cup S)$, i.e., $\nu \in (\Gamma \cup S)$. This shows that $((\Gamma \cup S)@\emptyset : \nu) \in \mathsf{CL}(\Gamma, S \rightarrow \nu)$.

Our next goal is to show that $(a : \nu) \in \mathsf{Select}(\Gamma'_o, a)$. By the Select definition this is equal to proving that $(a : \nu) \in \Gamma'_o$ and $\nu = \sigma(\nu)$ hold. We have previously proved the first statement, and the second statement follows from the σ definition.

From the previous we can conclude that the body of the second **foreach** statement in RCN will be executed. We select $a : \nu$ to be the type declaration $f : t_o$. Therefore, $m = 0$ and $\lambda x_1 \ldots x_n . a$ is put into TERMS. Note that the fresh variables $\lambda x_1 \ldots x_n$ we can choose exactly to match lambda variables of our e expression. This proves that $\lambda x_1 \ldots x_n . a \in \mathsf{RCN}(\Gamma_o, \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \nu, 1)$.

Inductive hypothesis [$\mathscr{D}(e) \leq k$]**:** Let Γ_o be an original environment, e an lambda expression, $\tau \in \tau_\lambda(B)$ and functions RCN and $\mathscr{D}$ defined as above, then:

$$\Gamma_o \vdash_\lambda e : \tau \Rightarrow e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$$

Inductive step [$\mathscr{D}(e) = k + 1$]**:** then $e \equiv \lambda x_1 \ldots x_n . h\ e_1 \ldots e_m$ and $\tau \equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \nu$. It also holds that e_i are terms in LNF and $\mathscr{D}(e_i) \leq k$. We assume that each e_i has a type ρ_i. We have to prove that $\lambda x_1 \ldots x_n . h e_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \nu, k + 1)$. We again follow the **else** branch in RCN.

Our first goal is to show that $((\Gamma \cup S)@\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} : \nu) \in \mathsf{CL}(\Gamma, S \rightarrow \nu)$. By the CL definition this is equivalent to proving that $(\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \nu) \in (\Gamma \cup S)$ and $\forall t' \in \{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} . (\Gamma \cup S) \vdash_c t'$.

Let us prove the statements. From $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n . h\ e_1 \ldots e_m : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \nu$ it follows that $\Gamma'_o \vdash_\lambda h\ e_1 \ldots e_m : \nu$, by the Abs (LNF) rule. Using the App (LNF) rule we conclude that $(h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu) \in \Gamma'_o$. After we apply σ, we have $\sigma(h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu) \in \sigma(\Gamma'_o)$, i.e., $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \nu \in \Gamma \cup S$. From the previous App (LNF) rule application it follows that $\Gamma'_o \vdash_\lambda e_i : \rho_i$. Applying Theorem A.1.1 to this we get $\Gamma \cup S \vdash_c \rho_i$, for $i = [1..m]$. Thus, by the App (succinct) rule we conclude that $(\Gamma \cup S, @\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} : \nu) \in \mathsf{CL}(\Gamma, S \rightarrow \nu)$ holds.

Our second goal is to prove $(h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu) \in \mathsf{Select}(\Gamma'_o, \{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \nu)$. Using Select definition we can see that this is equivalent to proving that $(h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu) \in \Gamma'_o$ and $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \nu = \sigma(\rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu)$ hold. We have previously prove the first statement, and the second statement follows from the σ definition.

Therefore, we conclude that the body of the second **foreach** loop will be executed, whereas we select $h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \nu$ to be $f : t_o$. Finally, we have to prove that $e_i \in T_i = \mathsf{RCN}(\Gamma'_o, \rho_i, d - 1)$.

This follows from the inductive hypothesis and Lemma A.1.4 because $d-1=k$ and $k \geq \mathscr{D}(e_i)$. Therefore, one iteration in third **foreach** loop will contain $e_1,\ldots,e_m$ sub-terms in the correct order. (The argument order is preserved due to the order of argument types in t_o). Finally, we use them to construct $\lambda x_1 \ldots x_n.h\ e_1 \ldots e_m$ and put it into TERMS. This proves that $\lambda x_1 \ldots x_n.h\ e_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon, k+1)$. Note that if n is zero then $e \equiv h\ e_1 \ldots e_m$.

Now, let us show correctness of the $\Leftarrow$ direction. We do this by induction on $\mathscr{D}(e)$.

Base case [$\mathscr{D}(e)=1$]**:** then $e \equiv \lambda x_1 \ldots x_n.a$ and $\tau \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$. We have to prove that $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.a : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$. If there exists $\lambda x_1 \ldots x_n.a \in \mathsf{RCN}(\Gamma_o, \tau, 1)$ then there is an execution of the **else** branch in RCN that produces $\lambda x_1 \ldots x_n.a$. (Because $\mathscr{D}(e)=d=1$ we conclude that the first execution will be the one that will produce e.) Also m must be zero because only the second then branch produce $\lambda x_1 \ldots x_n.a$. Also, $t_0 = \upsilon$. It follows that there exist a pattern p and Γ'_o such that $a : \upsilon \in \mathsf{Select}(\Gamma'_o, p)$. Note that by the Select definition every element that belongs to $\mathsf{Select}(\Gamma'_o, p)$ also belongs to Γ'_o, i.e., $a : \upsilon \in \Gamma'_o$. We know that there exists Γ_o such that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, and it will be exactly an environment that is passed as an argument to $\mathsf{RCN}(\Gamma_o, \tau, 1)$. Moreover, because $x_1, \ldots, x_n$ are fresh they are all different from a ant therefore $a : \upsilon \in \Gamma o$. Also, the types τ_j, $j=[1..n]$ are the corresponding argument types that appear in the right order in τ. Thus we can first apply the App (LNF) and then the App (LNF) rule to $\mathsf{RCN}(\Gamma_o, \tau, 1)$ and get $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.a : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$, i.e., $\Gamma_o \vdash_\lambda e : \tau$.

Inductive hypothesis [$\mathscr{D}(e) \leq k$]**:** Let Γ_o be an original environment, e an lambda expression, $\tau \in \tau_\lambda(B)$ and functions RCN and $\mathscr{D}$ defined as above, then:

$$\Gamma_o \vdash_\lambda e : \tau \Leftarrow e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$$

Base case [$\mathscr{D}(e)=k+1$]**:** then $e \equiv \lambda x_1 \ldots x_n.h\ e_1 \ldots e_m$ and $\tau \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$. We have to prove that $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.h\ e_1, \ldots e_m : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \upsilon$. If $\lambda x_1 \ldots x_n.h\ e_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau, k+1)$ then there exist an execution of the **else** branch in RCN that produces $\lambda x_1 \ldots x_n.h\ e_1 \ldots e_m$. (This must be the first execution of RCN because it is the only one that can produce terms with depth $\mathscr{D}(e)$. All others produce smaller terms.) Also $m \neq 0$ which means that last **foreach** is executed, and that $t_o = \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \upsilon$. It follows that there exist a pattern p and Γ'_o such that $h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \upsilon \in \mathsf{Select}(\Gamma'_o, p)$. Note that by the Select definition every element that belongs to $\mathsf{Select}(\Gamma'_o, p)$ also belongs to Γ'_o, i.e., $h : \rho_1 \rightarrow \cdots \rightarrow \rho_m \rightarrow \upsilon \in \Gamma'_o$. We know that there exists Γ_o such that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, and it will be exactly an environment that is passed as an argument to $\mathsf{RCN}(\Gamma_o, \tau, k+1)$. Types τ_j, $j=[1..n]$ are the corresponding argument types that appear in the right order in τ. From the inductive hypothesis and Lemma A.1.4 it follows that $\Gamma'_o \vdash_\lambda e_j : \rho_j$, where $j=[1..m]$. Thus we can first apply the Abs (LNF) and then the App (LNF) rule and get $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.h\ e_1 \ldots e_m : \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \upsilon$, i.e., $\Gamma_o \vdash_\lambda e : \tau$.

Theorem A.1.1 *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form (LNF) then $\sigma(\Gamma_o) \vdash_c \sigma(\tau)$.*

Proof By induction on $\mathscr{D}(e)$.

Base case [$\mathscr{D}(e) = 1$]**:** then $e \equiv \lambda x_1 \ldots x_n.a$ and $\tau \equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \upsilon$. From the fact that $\Gamma_o \vdash_\lambda e : \tau$ by the Abs (LNF) rule we know that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda a : \upsilon$. Form this it follows that $a : \upsilon$ is an element of Γ'_o. Knowing that $a : \upsilon \in \Gamma'_o$ by the definition of σ we also know that $\upsilon \in \Gamma'$. By applying the App (succinct) rule it follows that $\Gamma' \vdash_c \Gamma'@\emptyset : \upsilon$. Because $\Gamma' = \Gamma \cup S$, by applying the Abs (succinct) rule it follows that $\Gamma \vdash_c S \rightarrow \upsilon$. By simple substitutions we have $\sigma(\Gamma_o) \vdash_c \sigma(\tau)$. This way we proved that for all judgements of the depth one, the theorem claim holds.

Inductive hypothesis [$\mathscr{D}(e) = k$]**:** if $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then $\sigma(\Gamma_o) \vdash_c \sigma(\tau)$.

Inductive step [$\mathscr{D}(e) = k + 1$]**:** Then $e \equiv \lambda x_1 \ldots x_n.f\, e_1 \ldots e_m$ and $\tau \equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \upsilon$. Just like before, by the Abs (LNF) rule we know that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda f\, e_1 \ldots e_m : \upsilon$. Each e_i must have some type ρ_i, $i = [1..m]$. Then f has the type $\rho_1 \rightarrow \ldots \rightarrow \rho_m \rightarrow \upsilon$. By the Abs (LNF) rule we have $f : \rho_1 \rightarrow \ldots \rightarrow \rho_m \rightarrow \upsilon \in \Gamma'_o$ and $\Gamma'_o \vdash_\lambda e_i : \rho_i$. When we apply σ we get $\sigma(\rho_1 \rightarrow \ldots \rightarrow \rho_m \rightarrow \upsilon) \in \Gamma'$ i.e $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \upsilon \in \Gamma'$. Also, from the hypothesis it follows that $\Gamma' \vdash_c \sigma(\rho_i)$ because the depth of e_i is less or equal to k. Now we apply the App (succinct) rule to $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \rightarrow \upsilon \in \Gamma'$ and all $\Gamma' \vdash_c \sigma(\rho_i)$ and conclude $\Gamma' \vdash_c \Gamma'@\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} : \upsilon$. By simple substitutions we have $(\Gamma \cup S) \vdash_c (\Gamma \cup S)@S : \upsilon$ Finally, we apply the Abs (succinct) rule to $(\Gamma \cup S) \vdash_c (\Gamma \cup S)@S : \upsilon$ and conclude $\Gamma \vdash_c S \rightarrow \upsilon$. Again, by simple substitutions we have $\sigma(\Gamma_o) \vdash_c \sigma(\tau)$. This way we proved that for all judgements of the depth $k+1$, the theorem claim holds.

Lemma A.1.2 *Let Γ_o be a lambda environment and τ be a lambda type. Also let $d \geq 1$ be a number, then:*

$$RCN(\Gamma_o, \tau, d) = RCN(\Gamma_o, \tau, d-1) \uplus Terms_d$$

where $Terms_d$ is the set that contains only expressions with depth d.

Proof We prove this by induction on d.

Base case [$d = 1$]**:** then we have to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1) = \mathsf{RCN}(\Gamma_o, \tau, 0) \uplus Terms_1$. Because $\mathsf{RCN}(\Gamma_o, \tau, 0)$ is empty set by the RCN definition we have to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1) = Terms_1$. In other words, we need to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1)$ produces only terms with depth one. This is **true** because all $T_i = \mathsf{RCN}(\Gamma_o, \rho_i, 0) = \emptyset$. Therefore if there exist at least one expression it must be created in the `then` branch of **if** statement where $m = 0$. Thus $\mathsf{RCN}(\Gamma_o, \tau, 1)$ may only contain expressions of the form $\lambda x_1 \ldots x_n.a$ whose depth is 1.

Inductive hypothesis [$d = k$]**:** The following holds $\mathsf{RCN}(\Gamma_o, \tau, k) = \mathsf{RCN}(\Gamma_o, \tau, k-1) \uplus Terms_k$.

Inductive step [$d = k+1$]**:** then we have to prove that $\mathsf{RCN}(\Gamma_o, \tau, k+1) = \mathsf{RCN}(\Gamma_o, \tau, k) \uplus Terms_{k+1}$. We use the induction hypothesis to substitute $\mathsf{RCN}(\Gamma'_o, \rho_i, k)$ with $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1) \uplus Terms_k^{(i)}$. Thus, T_i sets become $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1) \uplus Terms_k^{(i)}$. Then from $\mathsf{RCN}(\Gamma_o, \tau, k+1)$ function it follows that we can choose each e_i either from $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$ or from $Terms_k^{(i)}$ (the sets are disjoint). Therefore, the Cartesian product $(T_1 \times \cdots \times T_m)$ becomes the union of 2^m Cartesian products of the form $(T_1^{(j)} \times \cdots \times T_{m'}^{(j)})$, $j = [1..2^m]$. $T_i^{(j)}$ is either $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$ or $Terms_k^{(i)}$. Therefore we can split $\mathsf{RCN}(\Gamma_o, \tau, k+1)$ into 2^m+1 subsets. We produced 2^m subsets using function with the same code as the RCN, except that $T_i^{(j)}$ replaces T_i. Let us denote new functions with $\mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. The last set contains all expressions generated when $m = 0$, and we denote it by RCN_0. Then $\mathsf{RCN}(\Gamma_o, \tau, k+1) := \mathsf{RCN}_0 \uplus \biguplus_{j=[1..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. If $\mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(1)}$ is the function where $T_i^{(j)} = \mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$, for $i = [1..n]$, then it holds $\mathsf{RCN}_0 \uplus \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(1)} = \mathsf{RCN}(\Gamma_o, \tau, k)$. In other functions at least one $T_i^{(j)} = Terms_k^{(i)}$. Such a function synthesizes only terms with depth $k+1$, because at least one sub-term e_i has depth k. This is the maximal depth of the sub-terms as well. By the definition of $\mathscr{D}$ we have that $\mathscr{D}(\lambda x_1 \ldots x_n.f\, e_1 \ldots e_n) = 1 + max(\mathscr{D}(e_1), \ldots, \mathscr{D}(e_n)) = 1 + k$. Therefore, it holds that all terms in those functions have depth $k+1$. We denote the set of all such terms by $Terms_{k+1} = \biguplus_{j=[2..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. Finally, we can conclude that $\mathsf{RCN}(\Gamma_o, \tau, k+1) = \mathsf{RCN}_0 \uplus \biguplus_{j=[1..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)} = \mathsf{RCN}(\Gamma_o, \tau, k) \uplus Terms_{k+1}$.

Lemma A.1.3 *Let Γ_o be a lambda environment and τ be a lambda type. Also let d and m be numbers, s.t. $d \geq 1$ and $d \geq m > 0$. Next, let $Terms_{d-m+1,d}$ be a set that contains terms with depth from $d-m+1$ to d then:*

$$RCN(\Gamma_o, \tau, d) = RCN(\Gamma_o, \tau, d-m) \uplus Terms_{d-m+1,d}$$

Proof By applying Lemma A.1.2 m times to $\mathsf{RCN}(\Gamma_o, \tau, d)$.

Lemma A.1.4 *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form and $d \geq \mathscr{D}(e)$ then the following holds:*

$$e \in RCN(\Gamma_o, \tau, \mathscr{D}(e)) \Leftrightarrow e \in RCN(\Gamma_o, \tau, d)$$

Proof If $e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$ then $e \in \mathsf{RCN}(\Gamma_o, \tau, d)$ is true because Lemma A.1.3 states that $\mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$ is a subset of $\mathsf{RCN}(\Gamma_o, \tau, d)$. On the other hand if $e \in \mathsf{RCN}(\Gamma_o, \tau, d)$ then either $e \in \mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$ or $e \in Terms_{\mathscr{D}(e)+1,d}$ by the same Lemma. However, by the definition $Terms_{\mathscr{D}(e)+1,d}$ contains only terms with depth greater then $\mathscr{D}(e)$. Thus, e cannot be in this set and must be in $\mathsf{RCN}(\Gamma_o, \tau, \mathscr{D}(e))$.

Bibliography

[1] SourceForge Source Code Repository. http://sourceforge.net/, 1999.

[2] EclipseJDT. http://www.eclipse.org/jdt/, 2004.

[3] The Djinn Theorem Prover. http://www.augustsson.net/Darcs/Djinn/, 2004.

[4] BitBucket Repository Hosting Service. https://bitbucket.org/, 2008.

[5] GitHub Repository Hosting Service. https://github.com/, 2008.

[6] Hayoo! API Search. http://holumbus.fh-wedel.de/hayoo/hayoo.html, 2008.

[7] Hoogle API Search. http://www.haskell.org/hoogle/, 2008.

[8] Eclipse Code Recommenders. http://www.eclipse.org/recommenders/, 2009.

[9] UDITA website. http://mir.cs.illinois.edu/udita, 2010.

[10] Intellij IDEA website. http://www.jetbrains.com/idea/, 2011.

[11] Miltiadis Allamanis and Sutton Charles. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[12] K. Apt and M. G. Wallace. *Constraint Logic Programming using Eclipse*. CUP, 2006.

[13] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State Generation and Automated Class Testing. *STVR*, 2000.

[14] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from Tests. In *ISSTA*, 2008.

[15] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *TPHOLs*, 2009.

[16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. pages 123–133, July 2002.

[17] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Inf. Comput.*, 93:172–221, July 1991.

[18] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from Examples to Improve Code Completion Systems. In *ESEC/SIGSOFT FSE*, pages 213–222, 2009.

[19] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[20] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS*, 2006.

[21] Aemon Cannon. Ensime. https://github.com/aemoncannon/ensime/, 2010.

[22] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *FASE*, pages 385–400, 2009.

[23] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*, 2000.

[24] Lori Clarke and Debra Richardson. Symbolic Evaluation Methods for Program Analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. 1981.

[25] Anthony Cozzie and Samuel T. King. Macho: Writing Programs with Natural Language and Examples. Technical report, University of Illinois at Urbana-Champaign, 2012.

[26] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In *ICSE*, 2008.

[27] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing. In *ASE*, 2006.

[28] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *ESEC/FSE 2007*, Dubrovnik, Croatia, September 2007. (To appear.).

[29] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating Typed Dependency Parses from Phrase Structure Parses. In *LREC*, pages 449–454, 2006.

[30] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[31] David Delahaye. Information Retrieval in a Coq Proof Library Using Type Isomorphisms. In *TYPES*, pages 131–147, 1999.

[32] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *ASE*, 2006.

[33] G. Dowek. Higher-Order Unification and Matching. *Handbook of automated reasoning*, II:1009–1062, 2001.

[34] Gilles Dowek and Ying Jiang. Enumerating Proofs of Positive Formulae. *Comput. J.*, 52(7):799–807, October 2009.

[35] Bassem Elkarablieh, Darko Marinov, and Sarfraz Khurshid. Efficient Solving of Structural Constraints. In *ISSTA*, 2008.

[36] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[37] Mauro Ferrari, Camillo Fiorentini, and Guido Fiorino. fCube: An Efficient Prover for Intuitionistic Propositional Logic. In *LPAR (Yogyakarta)*, 2010.

[38] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely Functional Lazy Nondeterministic Programming. In *ICFP*, 2009.

[39] Cormac Flanagan. Automatic Software Model Checking via Constraint Logic. *Sci. Comput. Program.*, 50(1-3):253–270, 2004.

[40] The Eclipse Foundation. http://www.eclipse.org/, 2007.

[41] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*, pages 653–663, 2014.

[42] Milos Gligoric, Tihomir Gvero, Steven Lauterburg, Darko Marinov, and Sarfraz Khurshid. Optimizing Generation of Object Graphs in Java PathFinder. In *ICST*, 2009.

[43] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-Based Whitebox Fuzzing. In *PLDI*, 2008.

[44] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[45] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In *TestCom/FATES*, 2009.

[46] Tihomir Gvero, Milos Gligoric, Steven Lauterburg, Marcelo d'Amorim, Darko Marinov, and Sarfraz Khurshid. State Extensions for Java PathFinder. In *ICSE*, 2008.

[47] Reid Holmes and Gail C. Murphy. Using Structural Context to Recommend Source Code Examples. In *ICSE*, pages 117–125, 2005.

[48] Radu Iosif. Symmetry Reduction Criteria for Software Model Checking. In *SPIN*, 2002.

[49] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the Costs of Bounded-Exhaustive Testing. In *FASE*, 2009.

[50] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-Guided Component-Based Program Synthesis. In *ICSE (1)*, 2010.

[51] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2 edition, 2008.

[52] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-Based Testing of Java Programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.

[53] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, pages 553–568, 2003.

[54] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[55] Etienne Kneuss, Viktor Kuncak, Ivan Kuraj, and Philippe Suter. Synthesis Modulo Recursive Functions. In *OOPSLA*, 2013.

[56] Harold W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[57] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete Functional Synthesis. In *PLDI*, 2010.

[58] Ivan Kuraj. Interactive Code Generation. Master's thesis, EPFL, February 2013.

[59] Ralf Lämmel and Wolfram Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *TestCom*, pages 19–38, 2006.

[60] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language. In *MobiSys*, pages 193–206, 2013.

[61] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. Temporal Code Completion and Navigation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1181–1184, Piscataway, NJ, USA, 2013. IEEE Press.

[62] Greg Little and Robert C. Miller. Keyword Programming in Java. In *ASE*, pages 84–93, 2007.

[63] Hugo Liu and Henry Lieberman. Metafor: Visualizing Stories as Code. In *IUI*, pages 305–307, 2005.

[64] Zhaohui Luo. Coercions in a Polymorphic Type System. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

[65] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*, 2005.

[66] Zohar Manna and Richard Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[67] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL*, pages 55–60, 2014.

[68] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.

[69] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An Evaluation of Exhaustive Testing for Data Structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[70] William M. McKeeman. Differential Testing for Software. *J-DTJ*, 10(1), 1998.

[71] Sean McLaughlin and Frank Pfenning. Efficient Intuitionistic Theorem Proving with the Polarized Inverse Method. In *CADE*, 2009.

[72] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-Based Semantic Code Search over Partial Programs. In *OOPSLA 2012 part of SPLASH, Tucson, AZ, USA, October 21-25, 2012*, pages 997–1016, 2012.

[73] Thomas Noll and Bastian Schlich. Delayed Nondeterminism in Model Checking Embedded Systems Assembly Code. In *HVC*, 2007.

[74] William F. Opdyke and Ralph E. Johnson. Refactoring: an Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *SOOPPA*, 1990.

[75] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *ISSTA*, 2008.

[76] C.S. Pasareanu, R. Pelánek, and W. Visser. Predicate Abstraction with Under-Approximation Refinement. *J-LMCS*, 3(1), 2007.

[77] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-Directed Completion of Partial Expressions. In *PLDI*, pages 275–286, 2012.

[78] Benjamin Pierce. *Types and Programming Languages*. 2001.

[79] David Price, Ellen Riloff, Joseph L. Zachary, and Brandon Harvey. NaturalJava: A Natural Language Interface for Programming in Java. In *IUI*, pages 207–211, 2000.

[80] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *PLDI*, page 44, 2014.

[81] John C. Reynolds. Using Category Theory to Design Implicit Conversions and Generic Operators. In *Semantics-Directed Compiler Generation*, pages 211–258, 1980.

[82] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for Sample Code. In *OOPSLA*, 2006.

[83] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[84] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching Stencils. In *PLDI*, 2007.

[85] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by Sketching for Bit-Streaming Programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[86] Richard Statman. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theoretical Computer Science*, 9(1):67 – 72, 1979.

[87] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software Assurance by Bounded Exhaustive Testing. In *ISSTA*, 2004.

[88] Suresh Thummalapenta and Tao Xie. PARSEWeb: a Programmer Assistant for Reusing Open Source Code on the Web. In *ASE*, 2007.

[89] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *ESEC/FSE*, 2009.

[90] Nikolai Tillmann and Jonathan de Halleux. Pex—White Box Test Generation for .NET. In *TAP*, 2008.

[91] Frank Tip. Refactoring using Type Constraints. In *SAS*, 2007.

[92] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *HLT-NAACL*, 2003.

[93] Pawel Urzyczyn. Inhabitation in Typed Lambda-Calculi (a Syntactic Approach). In *TLCA*, 1997.

[94] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[95] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. In *ISSTA*, 2004.

[96] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test Input Generation for Java Containers using State Matching. In *ISSTA*, 2006.

[97] J. B. Wells and Boris Yakobowski. Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In *LOPSTR*, pages 262–277, 2004.

Tihomir Gvero

Professional Address
EPFL IC IIF LARA BC358, Station 14
1015 Lausanne, Switzerland
(+41) 21 69 31221
tihomir.gvero@epfl.ch

Private Address
Chemin des Rosiers 1
1004 Lausanne, Switzerland
(+41) 76 56 79522
tihomir.gvero@gmail.com

Education

PhD in Computer, Communication and Information Sciences — 2009 - 2015
Swiss Federal Institute of Technology Lausanne (EPFL)
Switzerland, GPA 5.50/6.00

M.Sc. in Computer Science and Engineering, School of Electrical Engineering — 2007 - 2009
University of Belgrade, Serbia, GPA 10.00/10.00

B.Sc. in Computer Science and Engineering, School of Electrical Engineering — 2003 - 2007
University of Belgrade, Serbia, GPA 9.44/10.00

Conference Publications

1 **T. Gvero**, V. Kuncak, I. Kuraj and R. Piskac — 2013
"Complete Completion using Types and Weights"
Conference on Programming Language Design and Implementation (PLDI 2013)
pages 27-38, Seattle, Washington, USA, June 2013

2 **T. Gvero**, V. Kuncak and R. Piskac, "Interactive Synthesis of Code Snippets" — 2011
Computer Aided Verification (CAV Tool Demo 2011), pages 418-423
Snowbird, UT, USA, July 2011.

3 B. Daniel, D. Dig, **T. Gvero**, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec — 2011
S.H. Tan and D. Marinov, "ReAssert: A Tool for Repairing Broken Unit Tests"
International Conference on Software Engineering, (ICSE Demo 2011)
pages 1010-1012, Waikiki, Honoulu, Hawaii, USA, May 2011.

4 B. Daniel, **T. Gvero**, and D. Marinov, "On Test Repair using Symbolic Execution" — 2010
International Symposium on Software Testing and Analysis (ISSTA 2010)
pages 207-218, Trento, Italy, July 2010.

5 M. Gligoric, **T. Gvero**, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov — 2010
"Test generation through programming in UDITA"
32nd International Conference on Software Engineering (ICSE 2010)
pages 225-234, Cape Town, South Africa, May 2010.
(This paper won an ACM SIGSOFT Distinguished Paper Award.)

6 M. Gligoric , **T. Gvero**, S. Lauteburg, D. Marinov and S. Khurshid — 2009
"Optimizing generation of object graphs in Java PathFinder"
2nd IEEE International Conference on Software Testing
Verification and Validation (ICST 2009), Denver, CO, April 2009.

7 **T. Gvero**, M. Gligoric, S. Lauterburg, M. d'Amorim, D. Marinov and S. Khurshid — 2008
"State extensions for Java PathFinder"
Formal research demo at the 30th International Conference
on Software Engineering (ICSE Demo 2008), Leipzig, Germany, May 2008.

Technical Reports

8 **T. Gvero** and V. Kuncak, "On Synthesizing Code from Free-Form Queries" 2014
IC Technical Report LARA-REPORT-2014-09. 2014.

9 **T. Gvero**, V. Kuncak and R. Piskac 2011
"Code Completion using Quantitative Type Inhabitation"
IC Technical Report LARA-REPORT-2011-11. 2011.

10 M. Gligoric , **T. Gvero**, S. Khurshid, V. Kuncak and D. Marinov 2008
"On delayed choice execution for falsification"
IC Technical Report LARA-REPORT-2008-08. 2008.

Research Experience

PhD fellowship student, Laboratory for Automated Reasoning and Analysis (LARA), EPFL. Working with Prof. Viktor Kuncak on: 2009 - Present
- Interactive synthesis of Scala and Java code snippets [1, 2, 8, 9]
- Test input generation in UDITA language [5]

Summer intern, Microsoft Research Redmond. Worked with Nikolai Tillmann on a search strategy and a technique that improves test input generation in Pex, an automatic white-box test input generation tool. Summer 2010

Summer intern, EPFL, LARA. Worked with Prof. Viktor Kuncak on finding errors in programs and specifications [10] Summer 2008

Summer intern, University of Illinois at Urbana-Champaign (UIUC), Information Trust Institute (ITI) Undergraduate Research Internship Program, Worked with Prof. Darko Marinov on software testing and model checking [6, 7] Summer 2007

Tools & Extensions

anyCode is a tool that uses natural language input to synthesize code snippets. It employs a set of natural language processing tools, developed at Stanford, together with the costume-built related word map, based on WordNet, a large lexical database of English, to map natural language input to code snippets. To backed up the synthesis and make it more effective and efficient it uses unigram and probabilistic context free grammar models. It can also repair broken (ill-typed) expressions. 2014

InSynth is a tool for interactive synthesis of code snippets. It applies theorem proving technology to synthesize code fragments that use given library functions. The first version of InSynth synthesizes code applying polymorphic type constraints as well as code behavior. The second version is suitable for synthesizing code that contains first class functions. Both systems use the weights mechanism, based on unigrma model, to make synthesis effective and efficient. 2011 - 2013

Generation based on object invariants is an extension for Pex that: automatically infers an object (test input) invariant, sends the invariant to a constraint solver to determine valid object states, and synthesizes code that initializes the object using only the object's public API. It is used to aid test input generation, when object's state is hard to construct due to access control constraints. 2010

Delayed choice execution is an extension for Java PathFinder (JPF) that postpones the choices for each variable until it is first accessed. The extension also includes copy propagation technique that further postpones the choices even if the values are being copied. The techniques support primitive fields and linked structures. 2009

Undo optimization for backtracking is a JPF extension that speeds up state-space exploration, focusing on backtracking. The key idea is to incrementally store and restore states. Our results show an over of magnitude speedup for a number of programs. 2008

Untracked field is an extension for JPF for accumulating values over all paths 2007

	JPF patches that include 8 bug-fixes in JPF (array element type compatibility, arraycopy method type compatibility, annotation element types, casting primitive type arrays, output stream method, enumeration method, reflection method, cloning of primitive type arrays)	2007

Class Projects	**CPUSimulator** is a Swing-based visual simulator of a simple processor: Register Transfer Logic view, per-clock, per-instruction and per-program simulation advance, real-time register and memory values editing, support for custom program loading (developed using Java, 5000 lines of code)	2006
	Multi-threading support for C++ is a Java-like threading model for C++, with the following features: switching, explicit synchronous preemption, asynchronous preemption (caused by an interrupt), time sharing, round-robin scheduling; and primitives: semaphores, events, mutexes, monitors.	2005

Talks & Presentations	**PLDI Conference**, Seattle, WA, USA. gave a talk based on paper [1]	June 2013
	Argo Seminar, Belgrade, Serbia, gave a talk based on paper [2]	Feb 2011
	Scala Days, Lausanne, Switzerland, presented a poster on UDITA	Apr 2010
	Argo Seminar, Belgrade, Serbia, gave a talk based on technical report [10]	Apr 2009
	ICSE Research Demo Track, Leipzig, Germany, gave a talk based on paper [7]	May 2008
	ICSE, Leipzig, Germany, presented a poster based on paper [7]	May 2008

Awards & Honors	**ACM SIGSOFT Distinguished Paper Award** for [5]	2010
	Selected for Summer Internship at the Information Trust Institute, UIUC	2007
	Serbian Ministry of Education Student Scholarship	2004 - 2008

Professional Activities

Teaching assistant:

- Computer Science 1 (Fall 2013, Fall 2012)
- Information Technology Project (Fall 2011)
- Theoretical Computer Science (Spring 2011)

Paper reviewer: PLDI (2014), VSTTE (2012), POPL (2011), SAS (2011), ESOP (2011) ASE (2011, 2009, 2007), ICST Student Track (2008)
Student volunteer: ICSE 2008
Student member: ACM, ACM SIGSOFT

Technical Skills

Testing and Development Tools: Pex, JPF, Eclipse, MS Visual Studio .NET Rational Rose, MySQL, StarUML
Programming Languages: Java, C#, C++, C, Scala, SQL, 80x86 Assembler, VHDL, Latex