ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Interactive Code Generation

by

Ivan Kuraj

BSc., Software Engineering
School of Electrical Engineering, University of Belgrade (2010)

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

January 2012

Thesis supervisors: Professor Viktor Kuncak and Tihomir Gvero

Interactive Code Generation

by

Ivan Kuraj

Submitted to the School of Computer and Communication Sciences
on January 18, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis presents two approaches to code generation (synthesis) along with a discussion of other related and influential works, their ideas and relations to these approaches. The common goal of these approaches is to efficiently and effectively assist developers in software development by synthesizing code for them and save their efforts. The two presented approaches differ in the nature of the synthesis problem they try to solve. They are targeted to be useful in different scenarios, apply different set of techniques and can even be complementary.

The first approach to code synthesis relies on typing information of a program to define and drive the synthesis process. The main requirement imposes that synthesized solutions have the desired type. This can aid developers in many scenarios of modern software development which typically involves composing functionality from existing libraries which expose a rich API. Our observation is that the developer usually does not know the right combination for composing API calls but knows the type of the desired expression. With the basis being the well-known type inhabitation problem we introduce a succinct representation for type judgements that significantly speed up the search for type inhabitants. Our method finds multiple solutions and ranks them before offering them to the developer. We implemented this approach as a plugin for the Eclipse IDE for Scala. From the evaluation we concluded that this approach goes beyond available related techniques and can be very useful in practical software development.

In the second approach, synthesis of code is driven by explicit specification of code in terms of (formal) specification. The goal is to allow the developer to specify a program, by giving formal description of its behavior, rather than writing the code - the actual implementation is synthesized automatically. The practical value of such synthesis is immediately clear since this problem is generally hard. The approach solves this problem by combining existing tools for code generation, verification and testing within the synthesis process, and applies techniques for speeding it up. Interesting modifications to the synthesis driven by types were made to allow synthesizing expressions lazily, on demand, by searching for solutions in an incremental fashion. Results of the evaluation on several examples show that the implementation can be effective and useful in practice, while the approach still offers a lot of room for improvements.

Acknowledgments

I would like to thank my supervisor, Viktor Kuncak, for the great course and continuous motivation that got me interested into the topics of software analysis and verification. The opportunity he gave me in joining his ongoing work on InSynth turned out to be an important turning point that lead me into the interesting areas of software synthesis research. Frequent discussions with him about my progress and his willingness to communicate at almost any point during my thesis had an enormous positive impact and kept me motivated to learn and try out more things. I think that his continuous energetic efforts in motivating his group to exchange insights and ideas is the right way to spur a pleasant and effective atmosphere for any prosperous research community.

I would like to thank my advisor, Tihomir Gvero, for sparking the interest for his line of work and introducing me to the InSynth project. I respect that he accepted me as an equal team member and trusted me with important parts of InSynth design and implementation. Regardless of the results I was reporting to him, whether they were excellent or complete failures, he always kept his composure and reacted in a positive and pleasant way. All the interaction we had made me feel not only pleased to be working with him but also very friendly outside the scope of our collaboration.

In addition, I would like to thank Philippe Suter for his superb work in the topics that largely influenced my interests and the thesis. Having him within our group (and in the office just across mine) was extremely helpful since through our discussions at various occasions I received many useful suggestions and advices. I cannot forget to mention Regis Blanc who was always supportive and gave me a lot of advices during my thesis. I also thank Ruzica Piskac for making all those, from our point of view sometimes tedious and frustrating "against the clock" project meetings, enjoyable and fun. I must acknowledge a very enjoyable and motivating group of people from the Scala IDE team, especially Iulian Dragos, who have helped, taught and supported me and made me very lucky for being accepted for a summer project in their team. I cannot forget to thank my office colleagues for providing a great working environment, and an unexpectedly pleasant ways for spending free time between the work. I also thank the support staff at EPFL, Sylviane Dal Mas, Silva Patricia and Antonella Martin-Veltro who turned administrative obligations into amenities. I owe very special thanks to Antonella who was always kind but most importantly patient and empathic when hearing out my issues and trying to search for ways to help me.

I would like to thank many great people that I had a chance to meet, learn from, cooperate and have fun with during my master thesis. Special acknowledgment deserves the whole LARA group for which I feel especially honored and proud to be considered a part of. Finally, I thank my family and friends for believing in me and my ambitions, and not forgetting about me when I got back home. This thesis could serve as a justification for all the time I could have spent with them but instead had to sacrifice, and to some of them, for those awkward mornings in which they found me staring at the monitor instead of sleeping in the bed.

Contents

Chapter 1

Introduction

In this thesis we present techniques that construct a program that writes programs, i.e. techniques for program synthesis. Program synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints [1]. Synthesis requires a mechanism for expressing user intent that starts and drives the synthesis, thus synthesizers usually take input from the user. Synthesizers can accept a variety and mixed form of constraints including input-output examples, demonstrations, logical relations between inputs and outputs, even partial or inefficient programs. To make the synthesis meaningful and valuable the developer should be able to describe (i.e. specify) the program in a way that is simpler than writing the program itself. The problem of defining such a synthesis process is hard and even too abstract and infeasible in general. Without relaxing the problem and narrowing its domain, the problem can hardly be tackled.

The area of program synthesis has been studied for decades but it remained an active area of research. Many works consider this problem in a variety of specialized contexts and settings. One way to characterize and categorize approaches to program synthesis is presented in [1] and it identifies three key dimensions:

1. **User Intent** This represents the mechanisms used to describe the intent of the synthesis process, i.e. the input that drives the synthesis. Some of the choices include logical (formal) specification, input/output examples, natural language, incomplete and even inefficient programs.

2. **Search space** The key for efficiency of the synthesis is to define (and restrict) its search space. Synthesis process should be careful about the ratio between the expressiveness (thus applicability) and space of programs to consider in the synthesis process. The search space of programs can be qualified by two attributes reasoned about: operators and control structure. Besides programs

some approaches aim at grammars and logics.

3. **Search technique** Although this category directly depends on the considered search space, it deserves a separate discussion. Main classes of search strategies include brute-force search, logical reasoning and machine learning.

Works that describe techniques that are related or influenced techniques used in approaches to synthesis presented in this thesis include deductive synthesis and transformation to theorem proving [77, 48, 47], learning from input/output examples and pattern recognition [70, 40, 24], program sketching [67], frameworks for step-wise synthesis [22, 36], approaches based on generating models from decision procedures [42, 43], the type inhabitation problem [76] and various other strategies useful in specific scenarios that occur in practice, ranging from giving assistance to the developer, to automatic generation of tests [46, 25, 50, 57, 67, 55, 9].

The goal of our work is to tackle problems of program synthesis that relate to difficulties often encountered in the practice of modern software development and introduce techniques that could lead to tools that assist the developer. The developer specifies intent of synthesis with type constraints, for the first approach (explicitly, by giving the desired type, or implicitly, if the desired type is inferred) and with formal (logical) specification of the program to be synthesized and input/output examples, for the second approach. For the first approach, search space is narrowed down to searching for small code snippets that represent chains of function (API) calls that are to be inserted at holes of partial programs. Code snippets are searched with an exhaustive enumeration strategy that is driven by weights and based on the brute-force search[1]. The second approach cannot be strictly categorized according to the mentioned key dimensions because it internally uses other existing tools for synthesis. Although this means that all characteristics (and limitations) of the employed synthesizer are reflected to the overall approach, this does not prevent enhancing its expressiveness and effectiveness[2]. The search strategy can be described as being based on the "generate and test" approach (in some contexts referred to as brute-force search, while in our case it represents a variant of back-tracking) where code is synthesized and then tested for correctness. Both approaches to synthesis have goals to operate in various practical scenarios and settings so that they can prove useful to developers during software development.

[1]note that the search space is narrowed down by introducing a novel representation of types and terms

[2]this approach can utilize the one driven by types and enhance its expressiveness as it will be shown in Chapter 4

In the following sections we will introduce both approaches to synthesis in more detail.

1.1 Synthesis driven by types and weights

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 [73] and IntelliJ [35] 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.

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[3]. We therefore do not require the developer to indicate a starting value (such as a receiver) 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 recent work of Perelman et al [57] or the pioneering work on the Prospector tool [46].

Considering this more general scenario 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. In other words, find e such that $\Gamma \vdash e : T$. In our deployment, we compute Γ from the position of the cursor in the editor buffer. We look up T by examining the declared type appearing left of the cursor in the editor (or where type inference applies, as described in Section 2.4.4). The goal of the tool is to find an expression e, and insert it at the current program point, so that the overall program type checks.

The type inhabitation in the simply typed λ-calculus corresponds to provability

[3]providing desired type may not be necessary if type inference is supported

in propositional intuitionistic logic; it is decidable and PSPACE-complete [68, 76]. Guided by the experience from previous works [60, 31, 29], we developed a version of the algorithm that is both complete in the lambda calculus sense, so it is able to synthesize not only function applications, but also lambda abstractions and efficient when used in practice. We present our result in a *succinct types* calculus, which we tailored for efficiently solving type inhabitation queries. The calculus computes equivalence classes of types that reduce the search space in goal-directed (and weight-directed) search, without losing completeness. Moreover, our algorithm generates a representation of all solutions using the appropriate graph structure, from which any number of solutions can be extracted (including the cases when there are infinite number of solutions), thus making our synthesis approach complete in solving the type inhabitation problem. We also show how to use weights to guide the search. We present an implementation within the Eclipse IDE for Scala. 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.

Besides solving the problem of synthesizing valid expressions, our work addresses the problem of ranking found expressions so that the higher ranked expressions are more likely to be helpful to the developer. Our work combines proof search with a technique to find multiple solutions and to rank them. We introduce proof rules that manipulate weighted formulas, where smaller weight indicates a more desirable formula. Given an instance of the synthesize problem, we find proofs that determine multiple expressions of the desired type, and rank them according to their weight.

Our proof rules combine weights of premises to determine the weight of the conclusion, and ensure that very long proofs result in terms with a very large weight. Weight prioritization does not prevent the tool from finding proofs that an exhaustive application of proof rules would find, but play an important factor for the quality of generated results. 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.

We implemented our tool, InSynth within the Scala Eclipse plugin. 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 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 as a function of a number of visible declarations. We found that the expected snippets were found among the top dozen solutions in the great majority of cases in a short period of time. In over 90% of benchmarks the expected snippet appears among the first ten solutions. Moreover, in over 60% of benchmarks, the expected snippet appears first in the list. Except in the case of one benchmark, the total execution time of the synthesis process was less half a second. This suggests that InSynth can efficiently and effectively help the developer in software development. Furthermore, we evaluated a number of techniques deployed in our final tool, found that all of them are important for obtaining good results, and found that, even for checking existence of terms, on our benchmarks, InSynth outperforms recent propositional intuitionistic provers [49,21]. The results show that techniques presented in this paper are essential for the performance of the synthesis algorithm. Our and experience of users of InSynth testify the practical value of our tool in real-world development scenarios.

1.2 Lazy approach to reconstruction

The standard implementation of the reconstruction process in InSynth is eager, since it tries to reconstruct a specific number of code snippets at once, by employing an eager search strategy to find type inhabitants of the highest rank. This process requires a parameter that specifies how many code snippets should be synthesized. Knowing such a parameter is a serious constraint for the synthesis, in some practical scenarios. Moreover, if an instance of the type inhabitation problem has an infinite number of solutions, in terms of type inhabitants, the synthesis process gets inherently constrained by practical limitations in terms of number of code snippets it can synthesize.

In order to remedy this, we present an idea that allows systematic enumeration of reconstructed code snippets one by one, and makes the enumeration possible even in cases where an infinite number of solutions exists. We present techniques that realize the idea of lazy enumeration of reconstructed type inhabitants. The enumeration is lazy in the sense that the search for solutions is performed only when needed, i.e. when the next solution needs to be enumerated, and this involves exploring only the smallest amount of search space necessary for the next enumerated solution.

We present unordered lazy enumeration which enumerates reconstructed type inhabitants in an arbitrary order and guarantees that if an inhabitant is valid, it will be enumerated eventually, and its ordered flavor which poses additional constraints on the ordering of the enumeration. We show correctness properties that hold for these lazy approaches to reconstruction and present evaluation of the implementation within the InSynth reconstruction phase. Results show that this approach to reconstruction makes synthesis driven by types practically feasible, and even outperforms its eager counterpart, in many scenarios.

1.3 Synthesis driven by specification

In many scenarios, it is easier to describe what a computation does than it is to define it explicitly [10, 48]. That is, writing down the relation between the input and the output variables can be easy, even when constructing a program that would satisfy such a relation is difficult. Such relation, usually referred to as specification, effectively describes what should be done, instead of how it should be done. Thus, the main goal becomes to construct a program synthesizer that takes a relational description and tries to produce a program that is guaranteed to satisfy the given relationship. Since this relationship drives the synthesis process, the synthesized program is correct by construction, with respect to the relationship. Therefore, under the assumption that the specification is correct, no further efforts in terms of debugging or verification are needed from the developer.

The motivation behind our approach to program synthesis driven by specification comes from examining implementation of practical algorithms and tasks in functional programming. These examples were collected from various sources including textbooks on verification and practical implementations (most of them can be found in [16, 54, 71]). Some of these examples belong to the benchmark set used for evaluation of the implementation of this approach presented in Section 4.3.7. Most of these examples operate on arguments that represent algebraic data types (such as *List, Tree*) and have a similar implementation pattern. The implementation usually tests for an actual type of an algebraic data type argument (usually with a *match* or *if* expression) and proceeds with implementation of the behaviour that holds for each case. One key observation is that algorithms are recursive, that is, they implement control flows with a recursive call (usually as the single case of a complex branch expression). As an example, implementation of a concatenation of lists checks if an argument list is actually of type Nil - if yes, the other list can be returned (simple case), otherwise function branch contains a complex expression that invokes a recursive call. Through a large variety of examples, ranging from ones that implement short and simple to the ones that implement large and complex algorithms, the common pattern for encoding control flow with branches and recursive calls remains (complex algorithms usually have multiple cases to consider and even such patterns nested). This pattern effectively encodes a program with branches that represent correct implementations for certain inputs. The space of inputs is partitioned and each branch gives a correct implementation for inputs from one such partition, together with a condition that filters out inputs that are not in the partition. Such patterns can be encoded with multiple nested expressions of the form *if c then e*. The key observation is that the expressions for these implementations are simpler when reasoned about in separation and they can be synthesized effectively (e.g. with the synthesis approach driven by types presented in Chapter 2).

The technique of synthesis with condition abduction focuses on synthesizing code of a purely functional programing language. Its focus is on recursive rather than the iterative programs and the notion of looping is represented only with appropriate recursive calls. Introduction of recursive calls is closely related to the use of the principle of mathematical induction in the corresponding proof of the program behavior. Reasoning about the behavior of the synthesized program can usually be done with the help of structural induction [10, 13]. Although the technique does not reason about iterative programs and loops, the induction principles allow synthesis of corresponding recursive program forms.

Synthesis with condition abduction offers specifying the program to be synthesized with both formal specification and input/output examples. The goal is to synthesize expressions, that potentially implement complex algorithms, correct with respect to given specification from the developer. The aim of the approach is to achieve practical value in assisting developers in development of modern software. Thus, the user intent supports the combination of these two means of specifying behavior of a program. Specifications with example pairs have the advantages of naturalness (examples are easy to elaborate) and conciseness (examples can implicitly describe manipulations of parameters). Their disadvantages are limited expressive power and ambiguity (examples cannot completely specify a problem). Formal (logic) specifications have the advantages of expressiveness (axioms benefit from the full expressive power of logic) and non-ambiguity (axioms can completely specify a problem). Their disadvantages are artificiality (axioms can be difficult to elaborate, and to understand) and length (axioms require a complex formalization process) [22]. Note that formal specification subsumes the specification with example pairs but involves more complex mechanisms for specifying user intent for the synthesis. We aimed at combining these two means of specification because of their complementary strengths and weaknesses.

We present algorithms that are based on "generate and test" approaches and employ existing tools for code generation, verification and testing to achieve synthesis of programs correct by given specifications. In our approach, the developer is able to declare a function, attach its formal specification (in terms of precondition and postcondition) and provide input/output examples but omit its body in order to invoke the synthesis process. Our tool was implemented as Scala compiler plugin that internally uses InSynth, implementation of the synthesis approach driven by types given in Chapter 2 and Leon, a system capable of verifying functional programs [72]. It utilizes various techniques and heuristics to overcome practical limitations of the naive implementation of the core algorithm and optimize the synthesis process. We evaluate our tool on benchmarks that consist of several examples of practical algorithms, elaborate the synthesis procedure in detail for each example and demonstrate with results that synthesis with condition abduction can indeed be effective and useful in practice.

1.4 Contributions

The first contribution of this thesis is the proposal of a new code generation feature for IDEs that goes beyond currently available tools. It includes an efficient goal-directed search algorithm that solves the type inhabitation problem by operating on the newly introduced succinct representation of types and uses weights to guide the search. This contribution lead to an implementation of a Scala IDE plugin, InSynth, that implements all of the proposed techniques and is publicly available.

The second contribution is the proposal of an approach for generation of code that is correct with respect to both formal specification and specification given in terms of input/output examples. It presents a technique that employs existing tools for code synthesis, verification and testing to effectively synthesize arbitrarily complex expressions. In order to reuse the InSynth as the underlying synthesis tool, we presented a novel approach to term reconstruction that allows lazy enumeration of reconstructed trees according to their weights while still being complete. We present an implementation of a Scala compiler plugin that utilizes InSynth and Leon and is effective in synthesizing recursive algorithms that occur in practice (e.g. operations on lists and insertion sort) and provides insights into the potential of the approach for achieving high practical value.

An interesting contribution that emerged as a requirement for utilizing our approach to synthesis driven by types and weights in the approach to synthesizing correct programs, is the lazy enumeration of reconstructed code snippets. The approach for lazy reconstruction provides techniques for a systematic way of enumerating synthesized snippets in an incremental fashion, on demand. These techniques are based on several sound algorithms that allow constructing a stream of solutions that enumerates all possible solutions, while delaying the space exploration until it is necessary, and can be applied to a variety of traversal problems. We present an unordered and ordered enumeration, where the ordering can be determined by arbitrary weight function, and present an evaluation which shows the practical advantages of the lazy approach over the eager one[4].

1.5 Organization of this thesis

The rest of the thesis is organized as follows: Chapter 2 presents the approach to code generation driven by types and weights, Chapter 3 introduces lazy enumeration of reconstructed code snippets that extends and improves code generation driven by types and weights, while Chapter 4 presents the approach to code generation driven by specifications. Each of these chapters introduces one major contribution of our work, explains its goals through examples and motivation, presents necessary techniques, design and implementation, together with results of evaluation. Chapter 5 discusses works that motivated both of the approaches to synthesis and analyzes relations to our current work and ideas for its extensions.

[4]at this point, we refer to the standard reconstruction as eager, since its does not defer any computation

Chapter 2

Efficient code synthesis driven by types and weights

This chapter explains the idea and design decisions behind the code synthesis approach using types and weights, and its implementation realized in InSynth. We will start by presenting examples that demonstrate functionality of our approach, continue with theoretical foundations and introduction of the *succinct types* calculus, explain the implementation of InSynth which is integrated into Scala IDE for Eclipse [65] as a plugin and conclude the chapter with a discussion on correctness properties and evaluation of effectiveness.

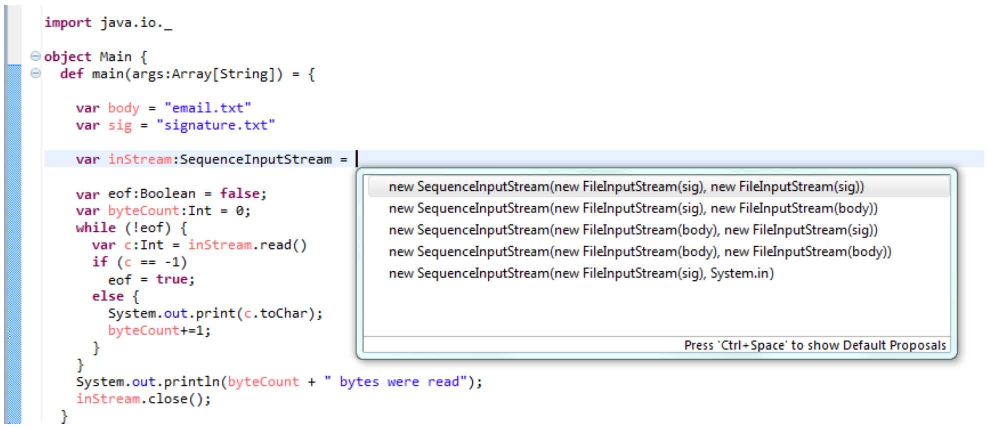
Although this approach to code generation is flexible and can be used for synthesizing code snippets in a variety of (functional) programming languages, in this section we will consider Scala [53] as the domain language and the only currently supported language for synthesis with InSynth. Flexibility of our modular design allows adding support for other languages easily which represents an interesting point left for the future work.

2.1 Motivating examples

We illustrate the functionality of InSynth through several examples. The first example serves as a simple illustration of typical usage scenario and goals of InSynth when used in an IDE. The second example is taken from the online repository of Java API examples `http://www.java2s.com/`. The third example is a real world example taken from code base of the Scala IDE for Eclipse[1]. The final example demonstrates how InSynth deals with subtyping. The original code of these examples imports only

[1]Scala IDE for Eclipse, `http://scala-ide.org/`

declarations from a few classes. To make the synthesis problem much harder for InSynth, we import all declarations from packages where those classes reside.



```
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)
        eof = true;
      else {
        System.out.print(c.toChar);
        byteCount+=1;
      }
    }
    System.out.println(byteCount + " bytes were read");
    inStream.close();
  }
```

```
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

Defining a variable. While programming in Scala, a developer very often needs to write an expression in order to initialize a newly declared variable or a field, or to generate a method body with a single statement. Our observation is that the developer usually knows the type of the expression, but is not sure *what* declarations to use and/or *how* to combine them. InSynth uses a desired type information and declarations in the context to synthesize an expression that type-checks at the point of invocation.

Consider the problem of opening a file with a given name at a given position given in the following fragment of code.

```
def openFileAt(name:String, pos:Int):File = {...}
var filename: String = "my_file.txt"
var position: Int = 100
...
var file:File = ▌
```

Assume that the developer set variables `name` and `position` to the name of the file and the desired file position for opening, respectively. Also, assume that a method `openFile` takes a name and opens a file at a given position. In order to open a file the developer defines a variable `file` of type `File`. The developer can initiate a query at a place where the `file`'s initializer should be written (denoted with ▌ in the figure). If so, InSynth extracts a desired type, i.e. `File`, with all visible declarations in the context. In the example, among the others, those include `openFile`, `filename` and `position`. InSynth then runs the synthesis algorithm and finds several solutions among which is also the code snippet `openFileAt(fileName, position)`. This expression, a code *snippet*, is of type `File`. Note, that argument types of `openFileAt` and types of `fileName` and `position` must match in order to insert that snippet into the code and make the whole variable definition successfully type-check. If there are more solutions, they are ordered and presented to the developer in a drop-down list.

Sequence of Streams. Here, 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 he invokes InSynth at the program point indicated by the cursor, in a fraction of a second it displays the ranked list of five expressions. Seeing the list, the developer can decide that e.g. the second expression in the list matches his 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 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 particular example, InSynth loads over 3000 initial declarations from the context, and finds the expected solution in less than 250 milliseconds. [2]

The effectiveness in the above example is due to several aspects of InSynth. InSynth ranks the resulting expressions according to the weights and selects the ones with the lowest weight. The weights of expressions and types guide not only the final ranking but also make the search itself more goal-directed and effective. InSynth learns weights from a corpus of declarations, assigning lower weight (and thus favoring) declarations appearing more frequently.

TreeFilter We demonstrate the generation of expressions with higher-order functions on real code from the Scala IDE project (see the code bellow). The example shows how a developer should properly check if a Scala AST tree satisfies a given property. In the code, the tree is kept as an argument of the class TreeWrapper, whereas property p is an input of the method filter.

```
import scala.tools.eclipse.javaelements._
import scala.collection.mutable._
trait TypeTreeTraverser {
  val global: tools.nsc.Global
  import global._
  class TreeWrapper(tree: Tree) {
```

[2]evaluation of benchmarks, including this and other examples, is shown in Section 2.6

```
    def filter(p: Tree => Boolean): List[Tree] = {
      val ft:FilterTypeTreeTraverser = ▮
      ft.traverse(tree)
      ft.hits.toList
    }
  }
}
```

The property 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 exactly the one expected (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.

Drawing Layout. Consider the next example, often encountered in practice, of implementing a getter method that returns a layout of an object Panel stored in a class Drawing. The following code is used to demonstrate how to implement such a method.

```
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.

2.2 Type inhabitation problem for succinct types

To answer whether there is 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 synthesizing code snippets.

Let T be a set of types. A *type environment* Γ is a finite set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ containing pairs of the form $x_i : \tau_i$, where x_i is a variable of a type $\tau_i \in T$. The pair $x_i : \tau_i$ is called a type declaration. With $\Gamma \vdash e : \tau$ we denote that from the environment Γ we can derive the type declaration $e : \tau$ by applying rules of some calculus. The type inhabitation problem is defined as: for a given calculus, 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 the standard lambda calculus restricted to normal form terms. We then introduce a new succinct representation of types and terms. To distinguish the original and succinct version of the calculus we use $\vdash_\lambda$ and $\vdash_S$ to denote derivability in the simply typed λ-calculus and in the succinct types calculus, respectively.

2.2.1 Simply typed λ-calculus for deriving terms in long normal form

Let B be a set of basic types. Types are formed according to the following syntax:

$$\tau ::= \tau \rightarrow \tau \mid v, \quad \text{where } v \in B$$

We denote the set of all types as $\tau_\lambda(B)$. When B is clear from the context we only write τ_λ.

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

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

The calculus given in Figure 2-2 describes how to derive new type judgements. Note that this calculus is slightly more restrictive than the standard λ-calculus. The APP rule requires that only those functions present in the original environment Γ_o can be applied on terms.

We restrict the APP rule in order to derive only the terms that are in so-called *long normal form* [69]. Our main motivation is to find suitable code snippets efficiently (while avoiding finding unnecessary terms that are not in *long normal form*).

$$\textsc{App}\ \frac{f:\tau_1 \to \ldots \to \tau_m \to \tau \in \Gamma_o \qquad \Gamma_o \vdash_\lambda e_i : \tau_i}{\Gamma_o \vdash_\lambda f e_1 \ldots e_m : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda e : \tau}{\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n . e : \tau_1 \to \ldots \to \tau_n \to \tau}$$

Figure 2-2: Calculus rules for deriving lambda terms in long normal form

Therefore, we derive only terms in long normal form, as they simplify and speed up the reconstruction process for code snippets. Note, that this does not restrict expressiveness of our calculus. Each simply-typed term can be converted to its long normal form [69, 6]. We now formally define long normal form.

Definition 2.2.1 (Long Normal Form) *A judgement $\Gamma_o \vdash_\lambda e : \tau_e$ is in long normal form if the following holds:*

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

In long normal form the number of bound variables corresponds exactly to the number of arguments. As an illustration, $f : \tau_1 \to \tau_2$ is not in long normal form, but $\lambda x . f x : \tau_1 \to \tau_2$ is in long normal form (η-conversion can converts between those two whenever x does not appear free in f).

We define the length $\mathcal{L}$ of a term from a long normal form judgement as follows:
$\mathcal{L}(\lambda x_1 \ldots x_m . a) = 1$
$\mathcal{L}(\lambda x_1 \ldots x_m . f e_1, \ldots, e_n) = \max(\mathcal{L}(e_1), \ldots, \mathcal{L}(e_n)) + 1$

2.2.2 Succinct types

Consider the code declaring a value and a function:

```
val a:Int = 0
def f(i1: Int, i2: Int, i3: Int):String = {...}
```

In the standard λ-calculus this code translates to the type environment $\Gamma_o = \{a : \text{Int}, f : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{String}\}$. Checking whether there is an inhabitant of type String requires three calls of the App rule. The application of currying typically constraints the search space even further and makes conceptually shallow proofs deeper. In order to make the search more efficient we therefore introduce *succinct types*, which are types modulo isomorphism of products and currying, or, equivalently, commutativity, associativity, and idempotence of conjunction (according to Curry-Howard correspondence [58,17]). In this example, succinct types would enable us to find an inhabitant in only one step.

Definition 2.2.2 (Succinct Types) *Let B_S be a set containing basic types. Succinct types t_s are constructed according to the grammar:*

$$t_s ::= \{t_s, \ldots, t_s\} \rightarrow B_S$$

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

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

Every type $\tau \in \tau_\lambda(B)$ can be converted into a succinct type in $t_s(B)$. With $\sigma : \tau_\lambda(B) \rightarrow t_s(B)$ we denote the conversion function. Every basic type $v \in B$ becomes an element of the set of basic succinct types, i.e. $B_S = B$ and $\sigma(v) = \emptyset \rightarrow v$. Let A (arguments) and T(type) be two functions defined on $t_s(B_S)$ as follows:

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

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

$$\sigma(\tau_1 \rightarrow \tau_2) = \{\sigma(\tau_1)\} \cup A(\sigma(\tau_2)) \rightarrow T(\sigma(\tau_2))$$

A type of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow v$ (a type that often appears in practice) has the succinct representation $\{\sigma(\tau_1), \ldots, \sigma(\tau_n)\} \rightarrow v$.

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

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

It can easily be shown that for every two type environments $\sigma(\Gamma_o^1 \cup \Gamma_o^2) = \sigma(\Gamma_o^1) \cup \sigma(\Gamma_o^2)$. By induction we can prove that the same holds for any union of type environments.

Types in InSynth

InSynth synthesizes valid Scala code snippets and it is straightforward to relate purely functional subset of the Scala programming language to the simply typed λ-calculus. In the first step, a typing environment needs to be extracted from the given Scala program and encoded in the succinct type representation. We illustrate correspondence between Scala and λ-calculus types on the examples given in Table 2.1. Note that Scala subtype relation (e.g. **class** τ_1 **extends** τ_2) deserves a special attention - we encode subtyping relation in the simply typed λ-calculus (e.g. $\tau_1 <: \tau_2$) with coercion functions in the *succinct types* calculus (more details about how we deal with subtyping can be found in Section 2.3.3).

Scala declaration	*Simply typed λ-calculus declaration*
val i: Int	i : Int
def f(a:Int, b:Char, c:Int): String	f : $Int \to Char \to Int \to String$
def g(f:(Int => Char), l:Long): String	$g : (Int \to Char) \to Long \to String$
class A { **val** s: String }	A.s : $A \to s$
class A **extends** B	$A <: B$

Table 2.1: Examples of the type declaration translation from Scala to the simply typed λ-calculus

Note that the type translation procedure to succinct types is irreversible. We cannot do the translation in the other direction and construct the original Scala type from a succinct type. Once we translate a Scala type we keep both type representations for each program declaration so that the information needed for reconstructing correct code snippets is preserved during the synthesis process. This allows us to use the succinct representation when solving the type inhabitation problem and afterwards to reconstruct type inhabitants in the corresponding Scala representation.

Table 2.2 shows these examples of Scala declarations (and their types) transformed to appropriate terms encoded in the *succinct types* calculus.

Note that while Definition 2.2.2 does not define succinct types for polymorphic types, but only for the set of basic types B_S which includes all constant types (which correspond to Scala primitive types) and all instantiations of type constructors (which correspond to instantiated Scala generic types). This means that the synthesis process is effectively limited to reason about succinct representation of types including *Int,*

Scala declaration	*Succinct declaration*
val i: Int	i : $\emptyset \to Int$
def f(a:Int, b:Char, c:Int): String	f : $\{Int, Char\} \to String$
def g(f:(Int => Char), l:Long): String	g : $\{Long, \{Int\} \to Char\} \to String$
class A { **val** s: String }	A.s : $\{A\} \to s$
class A **extends** B	cf : $\{A\} \to B$

Table 2.2: Correspondence between Scala declarations and proof representation terms

String, List[String], Map[Int, List[String]] but not of the type *Map[X, Y]*, where *X, Y* are polymorphic type variables.

Succinct patterns

Succinct patterns have the following structure $@\{t_S, \ldots, t_S\} : t_S$, where t_S are succinct types. A pattern $@\{t_1, \ldots, t_n\} : t$ indicates that types $t_1, \ldots, t_n$ are inhabited and an inhabitant of type t can be computed from them. They abstractly represent an application term in λ-calculus. We identify $@\emptyset : t_S$ and t_S.

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 schemes how a term can be computed. Additionally, each succinct pattern is annotated with the type environment for which it was derived. These annotations are needed for the second phase, where we do a term reconstruction based on the original type declarations (Γ_o) and the set of succinct patterns.

Calculus. Figure 2-3 describes the calculus for succinct types. We derive judgements for succinct patterns. As the patterns are derived only in the APP rule, we annotate the derived pattern with the actual Γ and save them into the set of all derived patterns. The rule ABS is a rule that modifies Γ - it can either reduce Γ or enlarge it, depending on whether we are doing backward or forward reasoning.

$$\textsc{Abs}\ \frac{\Gamma \cup S \vdash_S \pi : t}{\Gamma \vdash_S S {\to} t} \qquad \textsc{App}\ \frac{\{t_1, \ldots, t_n\}{\to} t \in \Gamma \qquad \Gamma \vdash_S t_1 \quad \ldots \quad \Gamma \vdash_S t_n}{\Gamma \vdash_S \ @\{t_1, \ldots, t_n\} : t}$$

Figure 2-3: Calculus rules for deriving succinct patterns . (The subscript $_S$ in $\vdash_S$ is a fixed symbol for "succinct" types, unrelated to the set of assumptions S in $\Gamma \cup S$)

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 $\Gamma = \{\emptyset \to \text{Int}, \{\text{Int}\} \to \text{String}\}$. By applying the APP rule on $\emptyset \to \text{Int}$, we derive a succinct pattern $@\emptyset : \text{Int}$ and we add $(\Gamma, @\emptyset : \text{Int})$ to the set of derived patterns. Having a pattern for Int we apply the ABS rule. By setting $S = \emptyset$, we derive $\Gamma \vdash_S \emptyset{\rightarrow}\text{Int}$. Finally, by applying again the APP rule, we directly derive a pattern for the String type and $(\Gamma, @\{\text{Int}\} : \text{String})$ becomes an element of the set of derived patterns.

2.3 Algorithms

In this section we will present algorithms used in InSynth. We will introduce two main algorithms: the algorithm that searches for solutions of the type inhabitation problem in the *succinct types* calculus and the term reconstruction algorithm. Afterward we will show how InSynth reasons about subtyping information using coercion functions. Finally we will present how the type inhabitation problem is modified to reason about multiple solutions and rank them according to weights, that is, formalize the so-called quantitative type inhabitation problem.

InSynth reasons about a subset of Scala that corresponds to simply typed lambda calculus. At a high-level, the algorithm behind our implementation consists of the following steps:

1. parse the program and extract declarations in *succinct types* calculus
2. follow the *succinct types* calculus rules to derive succinct patterns that encode inhabitants of the required type
3. use these patterns to reconstruct code snippets and rank them
4. present snippets with highest ranks to the developer

2.3.1 Type inhabitation problem in succinct calculus

We are interested in generating any desired number of expressions of a given type without missing any expressions equivalent up to β reduction [58]. To describe our approach to solving the type inhabitation problem in the *succinct types* calculus (and prove certain properties of it), we introduce two functions: CL and RCN[3]. The CL function takes as arguments a succinct type environment Γ and a succinct type $S{\rightarrow}t$. It returns the set of all patterns $@S_1{:}\tau$ derived in $\Gamma \cup S$:

$$\begin{aligned}\mathsf{CL}(\Gamma, S{\rightarrow}t) = \{(\Gamma \cup S, @S_1 : t) \mid {} & (S_1{\rightarrow}t) \in (\Gamma \cup S) \\ & \text{and } \forall t' \in S_1. \Gamma \cup S \vdash_S t'\}\end{aligned}$$

[3]the implementation of these functions will be described in Section 2.3

The function RCN is used to reconstruct lambda terms, based on the set of patterns and the original type environment. An additional argument of the RCN function is a non-negative integer d, used to specify that we only synthesize terms with length smaller or equal to d[4]. The algorithmic description of the RCN function is given in Figure 2.1.

```
fun RCN($\Gamma_o$, $\tau_1 \to \cdots \to \tau_n \to v$, d) :=
  if (d = 0) return $\emptyset$
  else
    $S \to v := \sigma(\tau_1 \to \cdots \to \tau_n \to v)$
    $\Gamma := \sigma(\Gamma_o)$
    $\Gamma'_o := \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ //$x_1, \ldots, x_n$ are fresh
    TERMS := $\emptyset$
    foreach ($\Gamma \cup S$, $@\{t_1, \ldots, t_{m'}\} : v$) $\in$ CL($\Gamma$, $S \to v$)
      foreach ($f : t_o$) $\in$ Select($\Gamma'_o$, $\{t_1, \ldots, t_{m'}\} \to v$)
        $\rho_1 \to \cdots \to \rho_m \to v := t_o$
        foreach $i \leftarrow [1..m]$
          $T_i$ := RCN($\Gamma'_o$, $\rho_i$, d$-$1)
        if ($\forall i \in [1..m].\ T_i \neq \emptyset$)
          foreach $(e_1, \ldots, e_m) \leftarrow (T_1 \times \cdots \times T_m)$
          //if m=0 then the empty tuple executes this loop once
          TERMS := TERMS $\cup$ $\{\lambda x_1 \ldots x_n . f e_1 \ldots e_m\}$
    return TERMS
fun Select($\Gamma_o$, $t$) := {v:$\tau$ | v:$\tau$ $\in \Gamma_o$ and $\sigma(\tau) = t$}
```

Listing 2.1: A function that constructs lambda terms in long normal form up to given length d.

Having CL and RCN functions, allows us to formalize our approach to solving type inhabitation problem and prove its soundness and completeness properties (presented in Section 2.5.1).

2.3.2 Synthesis of all terms in *long normal form*

We next present an algorithm based on the *succinct types* calculus that we use for finding type inhabitants. This algorithm is further used for the implementation of an interactive tool for synthesizing expression suggestions from which the developer can select a suitable expression. In order 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.

[4]note that the implementation uses weights instead of this depth parameter

In Figure 2-4 we illustrate the main algorithm that creates at most N terms with a type τ_o in Γ_o. All synthesized terms are in long normal form. The algorithm first uses σ to transform Γ_o and τ_o into succinct environment and type. Then it invokes the algorithm that calculates Derived on this environment and the type, Figure 2.2. The set Derived contains pairs (Γ, p), where p is a pattern derived in Γ. We also give a time limit (timeout) to the Core algorithm. Finally, the Rcnst-n algorithm takes Derived and constructs at most N lambda terms in long normal form.

```
TIP−ALL($\Gamma_o$, $\tau_o$, N, timeout) =
  Derived = Core($\sigma(\Gamma_o)$, $\sigma(\tau_o)$, timeout)
  Rcnst−n(Derived, $\Gamma_o$, $t_o$, N)
```

Figure 2-4: The algorithm that generates all terms with a given type τ_o and the environment Γ_o

The algorithms and implementation that implement Core and Rcnst-n are prestented in Section 2.4.2 and 2.4.3, respectively.

2.3.3 Subtyping using coercion functions

We use a simple method of coercion functions [45, 62, 11] 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 $v_1 <: v_2$ by introducing into the environment a fresh coercion expression $c_{12} : \{v_1\} \to v_2$. If there is an expression $e : \tau$, and e was generated using the coercion functions, then while translating e into a simply typed lambda terms, the coercion is removed (in the reconstruction phase). 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 did not change and the type inhabitation remains a PSPACE-complete problem [27, 26]. If subtyping constraints are present, then the coercion functions are used in construction of succinct patterns. However, in the reconstruction phase the coercion functions are omitted when deriving lambda terms (as explained in Section 2.4.3).

2.3.4 Quantitative type inhabitation problem

When answering the question of 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. Similarly 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 (or multiple terms with lowest weights).

Let w be a weight function that assigns to each variable a non-negative number. As the weight plays the crucial role in directing the search for inhabitants, it is important to assign meaningful weights. Section 2.3.4 describes how InSynth computes the weights. In general, the weight of a symbols is 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 2.3 we assign the least weight to local symbols declared in the same method. We assign the weight of one level higher to symbols defined in a class where a query is initiated. We assign an even higher weight to symbols in the same package.

2. the frequency with which the symbol appears in the training data corpus, as described in Section 2.6.2 below. For an imported symbol x, we determine its weight using the formula in Table 2.3. Here $f(x)$ is the number of occurrences of x in the corpus, computed by examining syntax trees in a corpus of code.

We also assign small weight to an inheritance conversion function that witnesses the subtyping relation. 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.

Based on these values we define a weight function w that assigns a weight to every symbol f. The weight of a term $\lambda x_1 \dots x_m . f e_1 \dots e_n$ is the sum of weights of all symbols that occur in the expression:

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

To guide the algorithm that generates patterns (in Figure 2.2) we use weights of

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

Table 2.3: 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.

succinct terms. Given *Select* in Figure 2.1, a weight of a succinct type t in Γ_o is defined as:

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

2.4 Implementation

InSynth can be thought of as being partitioned into two main modules: 1) type inhabitation solver module and 2) reconstruction module. The overview of the design is given in Figure 2-5.

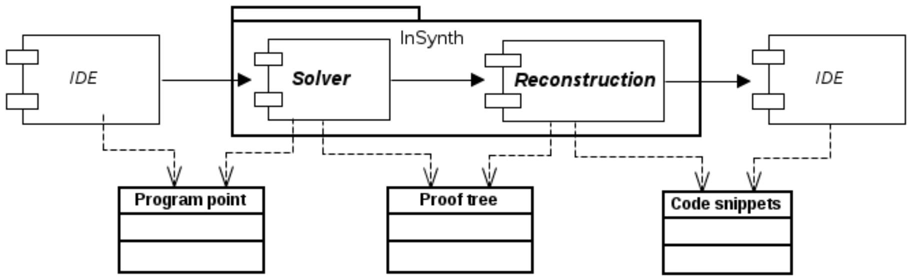


Figure 2-5: InSynth design

The type inhabitation solver module implements the first phase of the synthesis process which does the initial parsing of the given program (retrieved by the IDE) and extraction of typing information and then searches for all possible solutions to the type inhabitation problem in the *succinct types* calculus. The output is a set of succinct patterns encoded in a proof tree that witnesses type inhabitants, i.e. code snippets that can be reconstructed and then type-checked at the given program point with the desired type. Such proof trees are passed to the second, reconstruction module, which reconstructs expressions, orders them and provides them to the developer as a ranked list of code snippet suggestions within the IDE. Additionally, with the functionality provided by the IDE, it allows selection of particular code snippet and insertion of that snippet at the point of invocation in the editor buffer.

Note that the search for type inhabitants is done in the *succinct types* calculus, and as a consequence, type inhabitants need to transformed into terms in the simply typed λ-calculus (and reconstructed into Scala code snippets). Introduction of the *succinct types* calculus and the coupling of these two main phases allows solving the type inhabitation problem effectively and efficiently.

2.4.1 Type inhabitation solver

For a given type environment Γ and a succinct type τ, we address the type inhabitation problem by adding a new type declaration $\mathsf{goal} : \{\tau\} \to \bot$ (*query type*) and directing the search towards an inhabitant of type $\bot$. Symbol goal and type $\bot$ are fresh and previously unused, so an inhabitant of type $\bot$ can only be an expression of the form $\mathsf{goal}\{e\}$, where $e : \tau$. This allows directing the search for type inhabitants towards a single type (that is not usable as a standard declaration) and easier encoding of type inhabitants.

2.4.2 Pattern synthesis

In Figure 2.2 we present the algorithm that generates all succinct patterns starting from a type $S_i \to t_i$ in Γ_i, as formulated in the definition of the CL function. The Γ_i and $S_i \to t_i$ are initial succinct environment and the desired type, respectively.

INPUT:	succinct environment Γ_i and desired type $S_i \to t_i$
OUTPUT:	Derived - set of pairs (Γ, @S:t) that are derived

Figure 2-6: Description of the input and output of the algorithm

```
**fun** Core($\Gamma_i$, $S_i \to t_i$, timeout) :=
  Derived := $\emptyset$
  Inhabitants := $\emptyset$
  WorkingRequests := Requests := $\{(\Gamma_i$, NUL, $S_i \to t_i)\}$
  UninhabitedRequests := $\emptyset$
  **while** (WorkingRequests $\neq \emptyset$ and $\neg$ timeout)
    ($\Gamma$, $S \to t$, $S' \to t'$) := NextRequest(WorkingRequests)
    NewInhabitants :=
      ExplorRequest($\Gamma \cup S'$, $t'$, Derived, Inhabitants,
      UninhabitedRequests, WorkingRequests, Requests)
    PropagateInhabitants(NewInhabitants, Derived,
    Inhabitants, UninhabitedRequests, Requests)

fun ExploreRequest($\Gamma$, $t$, Derived, Inhabitants,
  UninhabitedRequests, WorkingRequests, Requests) :=
  NewInhabitants := $\emptyset$
  //find all succinct types in $\Gamma$ that return $t$
  foreach $(S \to t) \in \Gamma$
    Inhabited := true
    //See if we already have inhabitants for every type in $S$
    foreach $(S' \to t') \in S$
      if $((\Gamma, S \to t, S' \to t') \notin$ Requests)
        if $(\Gamma \cup S', t') \notin$ Inhabitants)
          Inhabited := false
          newRequest := $\{(\Gamma, S \to t, S' \to t')\}$
          Requests := Requests $\cup$ newRequest
          UninhabitedRequests :=
            UninhabitedRequests $\cup$ newRequest
          WorkingRequests := WorkingRequests $\cup$ newRequest
    //Record a new inhabitant and corresponding pattern
    if (Inhabited)
      if $((\Gamma, @S{:}t) \notin$ Derived)
        Derived := Derived $\cup \{(\Gamma, @S{:}t)\}$
        if $((\Gamma, t) \notin$ Inhabitants)
          Inhabitants := Inhabitants $\cup \{(\Gamma, t)\}$
          NewInhabitants := NewInhabitants $\cup \{(\Gamma, t)\}$
  return NewInhabitants

fun PropagateInhabitants(NewInhabitants, Derived,
                   Inhabitants, UninhabitedRequests) :=
  WorkingInhabitants := NewInhabitants
  while(WorkingInhabitants $\neq \emptyset$)
    $(\Gamma, t)$ := NextInhabitant(WorkingInhabitants)
    WorkingInhabitants := WorkingInhabitants
      $\cup$ PropagateInhabitant($\Gamma$, $t$,
          Derived, Inhabitants, UninhabitedRequests)

fun PropagateInhabitant($\Gamma$”, $t$”,
         Derived, Inhabitants, UninhabitedRequests) :=
  NewInhabitants := $\emptyset$
  foreach $(\Gamma, S \to t, S' \to t$”)
     $\in$ UninhabitedRequests and $\Gamma$” $= \Gamma \cup S'$
    UninhabitedRequests :=
      UninhabitedRequests $\setminus \{(\Gamma, S \to t, S' \to t')\}$
    //See if they can trigger new inhabitants
    if ($\forall$ $(S_1 \to t_1) \in (S \setminus \{S' \to t'\})$.
        $(\Gamma \cup S_1,\ t_1) \in$ Inhabitants)
      if $((\Gamma,\ @S{:}t) \notin$ Derived)
        Derived:= Derived $\cup$ $\{(\Gamma.\ @S{:}t)\}$
        if $((\Gamma,\ t) \notin$ Inhabitants)
          Inhabitants := Inhabitants $\cup$ $\{(\Gamma,\ t)\}$
          NewInhabitants := NewInhabitants $\cup$ $\{(\Gamma,\ t)\}$
  return NewInhabitants
```

Listing 2.2: The algorithm that generates all succinct expressions (patterns) with a given type $S_i \to t_i$ and the environment Γ_i

There are two alternating processes in the algorithm. First one explores types that are reachable from the desired type. We use our calculus rules (in backward manner) to determine what types are reachable. Therefore, this process goes from the desired type. Second process synthesizes patterns and goes in the opposite direction, towards the desired type. To form patterns we use our rules (in forward manner).

Request Exploration. The aim of this process is to discover the portion of the search space reachable from the desired type. In Figure 2.2 we use requests to mark the explored search space. Each request stores a tuple with Γ, a type $(S \to t) \in \Gamma$ already explored, and a type $(S' \to t') \in S$ that should be explored next. Let Γ_{init} and $S_{init} \to t_{init}$ be initial environment and the desired type. We start with the request (Γ_{init}, NUL , $S_{init} \to t_{init}$) that initiates WorkingRequests set. In the loop we choose the next request based on some criteria and remove it from WorkingRequests (the function NextRequest). Second, we call ExploreRequest(Γ, t) that explores the portion of the space reachable from t. It finds all succinct types $S \to t$, with T($S \to t$)=t, in Γ. It uses each $S \to t$ to create new requests if $S \neq \emptyset$. The requests record the facts that types in S should be explored in the future.

Intuitively, we start from the desired type and apply the ABS and APP rules in backward manner. Note that once we choose request (Γ, $S \to t$, $S' \to t'$) in the main loop, we pass $\Gamma \cup S$ to ExploreRequest. The ABS rule extends Γ in the same way if applied backwards. In the method ExploreRequest, the first **foreach** iterates over all types ($S \to t$). This corresponds to finding all $t_1, \ldots, t_n \to t \in \Gamma$ in the APP rule. In order for APP to succeed, we also need to check if $t_1, \ldots, t_n$ types can be inhabited. Thus in the next **foreach** loop we iterate over those types. Note that S is equal to $t_1, \ldots, t_n$. For each such a type we create a new request, that will be explored later (we put them in WorkingRequests set). The set Requests contains all created requests, which prevents re-exploration and ensures termination of the algorithm.

Another aim of the search is to reach types t that are inhabited. Type is inhabited if $\mathrm{A}(t)$ is empty, or all types in $\mathrm{A}(t)$ are inhabited. When we reach inhabited types they trigger a second process that discovers new inhabitant types. We next explain this process.

Inhabitant Propagation. The goal of this process is to discover new inhabited types. Another goal is to create and collect patterns whenever such a type is discovered. As mentioned above, in method ExploreRequest, once we reach a type $\emptyset \to t$, we know that APP succeeds. We say that t is inhabited in Γ. By the same rule, if all types in S are inhabited for some type $S \to t$, then t is also inhabited. The flag Inhabited will preserve value **true** if the type is inhabited. Once we discovered such a type, $S \to t$, we create a pair (Γ, $@S : t$) and put it in Derived. We also put the pair (Γ, t) into the set of all inhabited types, Inhabited. This set is used to preserve termination. All new inhabited types in ExploreRequest are passed to the PropagateInhabitants function. The function puts them in a working set WorkingInhabitants and process them one by one. The function stops once WorkingInhabitants is empty. PropagateInhabitant takes a new inhabitant type t" and its corresponding Γ" as inputs. The idea is to find all requests that need an inhabitant with type $S \to t$". We find them in the "foreach" loop. Those request have the following form (Γ, $S \to t$, $S' \to t$"). If we have inhabitant of type t" we also need to check if we can decompose Γ" into Γ, S'. Namely, it must hold Γ"$=\Gamma \cup S'$. This allows us to apply the ABS rule in forward manner. Thus, we can conclude that $\Gamma \vdash_s S' \to t$", i.e., $S' \to t$" can be inhabited. The set UninhabitedRequests keeps all requests without inhabitant. Once we discovered a request with inhabitant, we can remove it from this set.

The most interesting is the part that checks if new inhabitants can be derived. We use S to find all types $(S_1 \to t_1) \in S$. If they are all inhabited, then t is also inhabited. This follows from the APP rule when it is applied in the forward manner. We then create corresponding pattern. We update Inhabited and Derived in the same way like in ExploreRequest. Finally, the function collects, returns and puts new inhabitants into WorkingInhabitants set.

The function NextRequest chooses a request with the smallest weight. The weight of a request is equal to the weight of a type it needs to explore.

Output of pattern synthesis

Pattern synthesis produces succinct patterns encoded in the form of proof trees. InSynth proof trees represent "proofs" of existence of solutions to the type inhabitation problem - that is, they witness the existence of expressions that can indeed by reconstructed out of declarations from the given program environment as leafs and type-check to the desired type. These proofs bear information that such valid expressions can be derived in the *succinct types* calculus but does not directly encode how should the synthesized code snippets actually look like when given as suggestions to the developer (syntactically correct in the domain language). Therefore, InSynth proof trees need to include and propagate the information about correspondence between original program declarations and their succinct counterparts in order to allow reconstruction of valid code snippets. In addition to this, proof trees also include information about the weights of program declarations to allow ranking of reconstructed terms (see Section 2.3.4).

An example that depicts the outline of the structure of proof trees is given in Figure 2-7.

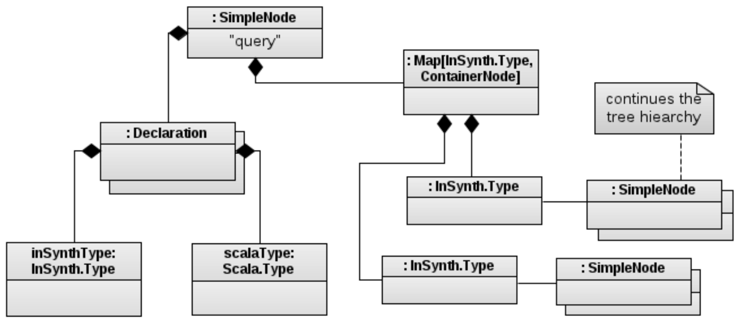


Figure 2-7: Object diagram with an InSynth proof tree example

The proof tree consists of nodes (implemented as *SimpleNode* classes) that carry the information about how to reconstruct an expression according to a program declaration (given by *Declaration* class) from sub-expressions of types made available by the parameters map. The parameters map represents information on how to construct sub-expressions of a given node and maps a type (*InSynth.Type*) to a set of subtrees (contained in the *ContainerNode* object). Each *Declaration* is associated with its succinct type *InSynth.Type* and its corresponding Scala type *Scala.Type* which stores the language-specific information that are needed for the reconstruction (and weights for ranking).

Note that each node, represented by the *SimpleNode* class, carries the information for reconstructing expressions of some type τ, that is, it encodes a set of patterns $@S_1 : \tau$ such that for each declaration in the node with succinct type τ_d, $T(\tau_d) = \tau$ holds, and for each $t \in \bigcup A(\tau_d)$ there is a corresponding child *ContainerNode* (that contains multiple *SimpleNode* nodes) that reconstructs to expressions of type t. The root node encodes the set of all patterns that can derive the artificially introduced type $\perp$ in the initial environment Γ, i.e. $CL(\Gamma, \tau \rightarrow \perp)$ where τ is the desired type.

2.4.3 Reconstruction of terms

In the following sections we will focus on the reconstruction part of the synthesis process that has the specific role of reconstructing code snippets from proof trees. The recursive function Rcnst-n starts from the desired type τ, the desired number of solutions N and applies the following steps:

1. Use type τ to find appropriate declarations of the given type
2. Construct a partial expressions by instantiating bound variables (if any), while its sub-expressions are left to be holes
3. Put partial expression into a priority queue based on their current cumulative weights
4. Remove the expression with the smallest weight from the queue
5. If the expression is fully instantiated (with no holes), count it as a solution and terminates if the number of found solutions is N
6. If the expression is not fully instantiated, recursively reconstruct its hole sub-expressions

Note that the process effectively involves a weight-directed search over the proof tree that encodes type inhabitants and needs to guarantee that the search is over when N code snippets of the highest rank (smallest cumulative weight) are reconstructed[5]. In the following sections we will explain the implementation of the reconstruction algorithm and its integration with other modules in our tool in more detail.

Overview of the reconstruction module

In this section we will describe the implementation of the reconstruction module which has the task of extrapolating and synthesizing code snippets from the proof trees obtained as a result of the resolution phase and ranking them according to their weight.

The overview of the reconstruction module is given in Figure 2-8.

The reconstruction phase starts when the type inhabitation solver phase finishes (although its design allows starting the reconstruction phase as soon as possible and running it in parallel with the resolution phase while receiving partial proof tree

[5]note that this method can terminate before inspecting the whole proof tree since weights are strictly non-negative

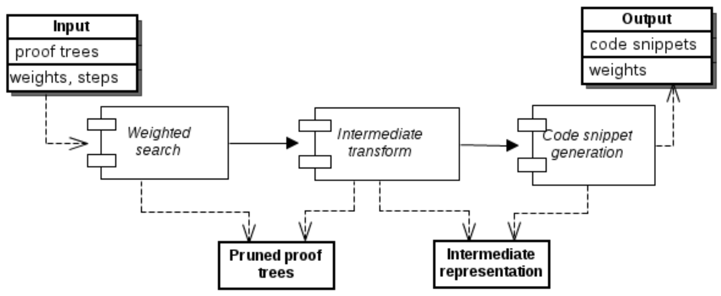


Figure 2-8: InSynth reconstruction module

updates and works on the proof tree representation. The input to the reconstruction phase besides the constructed proof trees includes the maximal execution time (defined in terms of steps in Figure 2.1, a constraint put on the InSynth for responsiveness) and the number of code snippets that should be generated and suggested to the developer.

The first step of the reconstruction phase is to extract a subtree of the proof tree (and transform it into very similar pruned tree representation) which is guaranteed to hold enough information for generation of sufficient number of code snippets that have the lowest weight. The second step takes such pruned proof trees, consults embedded information about the program environment and constructs an intermediate representation tree which holds enough information about the structure of code (encoded in simply typed λ-calculus) and program declarations (Scala-specific information). The third step takes the intermediate representation tree and applies transformations which generate a set of Scala code snippets and reports them back to the Eclipse IDE and the developer.

Weighted search

Weighted search is the first step in the reconstruction process and its goal is to prune the proof tree so that it encodes only the needed number N of the most optimal combinations in terms of associated weights. The result of this step is a proof tree that contains only a subset of nodes from the original, input proof tree such that the belonging nodes are sufficient in constructing at least $max(N, m)$ code snippets, where m is the number of encoded type inhabitants, with the lowest weight that need to be suggested to the developer [6]. The rationale behind this step is that the resolution step may, due to combinatorial explosion, output complex proof trees with

[6]due to the nature of succinct patterns, from one succinct pattern multiple expressions in the domain language can be reconstructed so the pruned proof tree is guaranteed to hold information to generate N or more code snippets

a large number of nodes and the reconstruction phase can benefit from its pruning to achieve better performance in consequent phases and overall responsiveness of the typing assist. After this step, the unnecessary nodes are removed from the proof tree and the output entails only the necessary information for construction of most optimal solutions based on the assigned weights of program declarations.

The algorithm that accomplishes the weighted search is based on the uniform-cost search which is a search algorithm used for traversing weighted tree structures [64]. Priority queue is used for storing partial expressions and directing the search according to declaration weights. The search begins at the root node and continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner recursively, until the goal state is reached. Instead of having a single global goal towards which the search is guided (as in the uniform-cost search), our version of the algorithm needed to be modified since there can be multiple solutions to the quantitative type inhabitation problem. When a subtree is fully explored, its cumulative weight serves as a weighted goal for constructing an expression corresponding to that subtree. When the root of the tree is fully explored we can compute the number of expressions that can be combined (i.e. solution snippets to reconstruct) that are solutions of the goal (*query*) type. Number of expressions any subtree can combine is equal to the sum, over all declarations, of products of number of explored combinations for each subexpression according to given declaration. The algorithm gradually explores nodes according to their weights (set of visited nodes is maintained since in general there can be cycles in the proof tree and number of traversals of recursive edges needs to be bounded), prunes the subtrees that ought to construct expressions of weight larger than optimal and finishes when the resulting tree contains nodes for construction of at least N expressions.

Note that the algorithm does not stop until it exhausts the priority queue of unexplored nodes and only examines nodes that can make new optimal subtrees (with total weight less than previously explored subtrees). This is needed since in order to be sure that optimal solutions are found, all the nodes in the tree need to be either pruned or dequeued and processed.

The weighted search algorithm is depicted in Algorithm 1.

The design of the algorithm allows implementation of incremental updates of proof trees from the resolution phase (which would add unexplored nodes to the priority queue) and it allows its implementation to support some more sophisticated policies for nodes weight calculation (instead of summing the weights of subtree node it can search for package declarations path matches, specific subtree pattern matches, etc.).

Algorithm 1 Weighted search

```
Require: root node $r$
 1: enqueue $r$
 2: while queue is not empty do
 3:   dequeue node $n$
 4:   if $n$ is not pruned then
 5:     for all nodes on the path to $r$ do
 6:       update its weight
 7:       if pruning is enabled then
 8:         prune the node if needed according to their updated weights
 9:       end if
10:     end for
11:     if $n$ is a leaf node then
12:       mark the subtree as explored and propagate this information up to $r$
13:       if number of combinations at $r$ is $\geq N$ then
14:         enable pruning of nodes
15:       end if
16:     end if
17:     for all children $c$ of $n$ do
18:       if $c$ is not visited then
19:         enqueue $c$
20:       end if
21:     end for
22:   end if
23: end while
24: return $r$
```

Intermediate transformation

In the second step of the reconstruction phase, an intermediate transform is performed on the pruned proof tree.

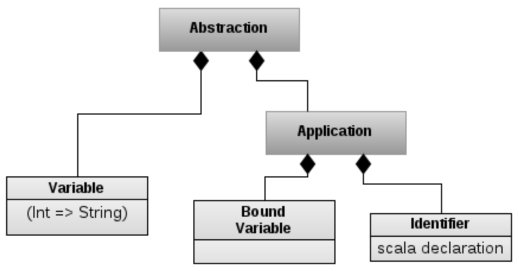


Figure 2-9: Intermediate representation tree example

The goal of this step is to produce intermediate representation trees which combine program declarations and combinations encoded in proof trees and contain enough information about the structure of the code to be synthesized. The resulting trees in the intermediate representation are structurally similar to λ-calculus terms and thus precisely encode information about abstractions and applications that form expressions in the simply typed λ-calculus[7].

An example of a tree in intermediate representation is given in Figure 2-9. The example encodes a single expression in which an identifier from the program context is applied to the function term declared in the root node abstraction.

The intermediate representation trees contain abstraction nodes which correspond to abstraction terms in the λ-calculus but which can bind multiple terms (in order to correspond to syntactically more powerful counterparts, functions and methods in Scala), application nodes which represent an application in the λ-calculus (again, the distinction is that they can include multiple parameters, like regular function invocations in Scala), identifier nodes that represent program declarations and bound abstraction variables. Important property of the intermediate representation is that each node can contain multiple nodes in the place of its sub-terms - this allows encoding multiple combinations of sub-expressions to form the expression encoded by that node and thus efficient mapping of encoded inhabitants to multiple actuall expressions in simply typed λ-calculus. Note that the intermediate representation structurally corresponds to λ-calculus and thus provides the same expressive power of λ-calculus (and thus is Turing-complete [53, 58, 75]). Therefore, it is more expressible than the succinct pattern encoding which encodes only terms in the *long normal form*.

We will now describe the algorithm for transforming InSynth pruned proof trees to intermediate representation trees.

Algorithm 2 Transformation procedure

```
Require: InSynth proof tree rooted at $r$
1: {$r$ is the query node which has type $(T \to \bot)$}
2: return Transform($\emptyset$, $r$, $\bot$)
```

The entry point to the transformation is given in Algorithm 2. Its input is the (pruned) proof tree, more specifically its root r which encodes expression of the *query type* $T \to \bot$, where T is the type of expressions we want to synthesize. The algorithm initializes a typing context that defines the current typing environment visible during the reconstruction of each subtree. It then calls the recursive *Transform* procedure given in Algorithm 3 on the root node with an empty context and with $\bot$ as the goal type. Note that if an expression of type $\bot$ is derived then its immediate sub-

[7] the intermediate representation offers an abstract way of encoding the code structure so that other programming languages can be supported as domain languages in the synthesis

expressions will be of the required type T.

Algorithm 3 Transform

```
Require: context $\Gamma$, current node $n$, goal type $t$
 1: {context $\Gamma$ is the current typing context}
 2: if $t$ is of the form $(X \Rightarrow Y)$ then
 3:     for all type $X_i$ in $X$ according to real type do
 4:         let be fresh variable $x_i$ of type $X_i$
 5:     end for
 6:     let $a$ be an abstraction that bounds all variables in $X$ {$a = (\lambda x_1 : X_1.(\lambda x_2 :$
        $X_2. \ldots (\lambda x_n : X_n.$ "_")))}
 7:     for all $t'$ in Transform declarations($\Gamma \cup (\bigcup_i x_i : X_i)$, $n$, $Y$) do
 8:         return $a$ [ "_" $\rightarrow t'$]
 9:         {plug $t'$ into the abstraction $a$ in place of "_"}
10:     end for
11: else
12:     return Transform declarations($\Gamma$, $n$, $t$)
13: end if
```

The recursive Transform, depicted in Algorithm 3 expects as inputs the current node n in the proof tree, current typing context Γ and a goal type t, to which expressions constructed from subtree n should type-check to. It simply checks the form of t and updates the typing context accordingly (at the first call, t is equal to the *query type*):

if t is not a function type according to the App rule (given in Figure 2-3), t represents the type derived from the patter $@\{t_1, \ldots, t_n\} : t$ and the transformation proceeds immediately recursively to get sub-expressions of types $t_1, \ldots, t_n$

if t is a function type $X \rightarrow Y$ t represents $S \rightarrow t$ according to the Abs rule so an abstraction terms needs to be formed in order to introduce variables of types found in S and the transformation proceeds recursively to get sub-expressions of type t under the updated context

The recursive transformation in both cases is achieved with a helper procedure given in Algorithm 4, which scans the available declarations in the current node and current context in order to transform sub-expressions.

The *Transform declarations* procedure scans declarations at the given node and the context to find suitable declarations that can transform to the goal type given as a parameter. Such declarations may be returned immediately in the case of an identifier or bound variable nodes (do not require application to them) or as applications of recursively transformed parameters of appropriate type.

Algorithm 4 Transform declarations

```
Require: context $\Gamma$, current node $n$, goal type $t$
 1: search for declarations that can return $Y$ type in declarations contained in the
    node $n$ and in $\Gamma$
 2: denote the result set of declarations as $D$
 3: for all declarations $d$ from $D$ do
 4:   if $d$ has no parameters then
 5:     return identifier or bound variable node for $d$
 6:   else
 7:     for all parameters $p_i$ of type $t_i$ in $d$ do
 8:       for all child nodes $n'$ that are contained in container node $parameters(t_i)$
          do
 9:         let $S_i$ be the result of Transform($\Gamma$, $n'$, $Y$) {set of nodes that represent
            sub-expressions of type $t_p$}
10:       end for
11:     end for
12:     return application node ($d$ $S_1$ $S_2$ $\dots S_n$) {if $d$ has $n$ parameters}
13:   end if
14: end for
```

Code snippet generation

The code snippet generation represents the final phase done in the reconstruction module and it takes a tree in the intermediate representation as input and produces code snippets in the target language as output, ranked according to their weight.

This step is based on a tree traversal transformation algorithm which traverses the intermediate representation tree and produces a set of code snippets. Since the intermediate representation tree encodes the program structure and also allows multiple sub-trees in its abstraction and application nodes, the code snippet generation step has to consider and collect every possible expression that is included in the intermediate representation tree. It outputs only the needed number of N snippets with the lowest weight, where N is the parameter to the synthesis process, in the non-decreasing order according to their weights and discards other snippets. Note that this needs to be done since intermediate representation encodes solutions in terms of combinations of subtrees and in general cannot encode the exact number of N needed solutions. Therefore the number of encoded solutions is always bounded from below by N (if sufficient number of solutions actually exists, otherwise all encoded solutions can be reconstructed and returned).

Although the intermediate representation precisely encodes the structure of code snippets to be generated in terms of λ-calculus terms, the code snippet generation step has to consult the information provided by the program declarations in order to be able to generate syntactically correct code and also to be able to simplify the resulting code as much as possible. These language-specific transformations depend on the adopted target language and in the case of Scala, include usage of necessary syntactical constructs (e.g. keyword *new* for construction of new objects, parentheses for currying), optional syntactic sugar instances (e.g. omitting dot, parentheses and *apply* in certain method calls) and general simplification steps (e.g. omitting of explicitly given types to expressions).

Resulting code snippets are encoded as a set of Scala pretty print documents (*Scala.text.Document* objects) which are then transformed by custom-indentation defined properties to strings and reported back to the IDE (the output of such objects can be then processed with Scala format and refactoring libraries in order to have visually better suggestions reported to the developer, as described in architecture section in [28]).

2.4.4 Implementation in Eclipse

We implemented InSynth as a plugin for Eclipse IDE for Scala [28] that extends the Eclipse code completion feature for automatic generation of well-typed Scala expressions [2]. It enables developers to accomplish a complex action with only a few keystrokes: declare a name and 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 (invoked 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 choices are displayed as proposals in the content assist proposal list, in the order corresponding to the snippet ranking. InSynth supports invocation at the place right after declaring 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 (in terms of omitting unnecessary parentheses, using method name shorthands, etc.). InSynth is available for download and is currently maintained as a part of the Eclipse Scala IDE plugin [2].

2.5 Correctness of the approach

In this section we define and prove correctness properties of our approach to solving the type inhabitation problem in the *succinct types* calculus and show that those properties are preserved in the implementation of InSynth.

2.5.1 Soundness and completeness of succinct calculus

The calculus defined in Figure 2-3 is sound and complete with respect to synthesis of lambda terms in long normal form. This represents an important claim for this work. In the sequel we will present two theorems that define correctness properties of the synthesis approach but omit their more formal counterparts and detailed proofs which can be found in the original paper and the associated technical report [26].

The following two important theorems are defined:

Theorem 2.5.1 (Existence lemma) *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then $\sigma(\Gamma_o) \vdash_S \sigma(\tau)$.*

This lemma states that for every derivable expression in λ-calculus an equivalent pattern will be found in the *succinct types* calculus. That is, for each judgement in the long normal form derived in the standard λ-calculus, an equivalent judgment in the *succinct types* calculus can also be derived. This effectively establishes a relation between expressions in λ-calculus and proof trees constructed with the *succinct types* calculus and allows an implementation to (efficiently) check existence of proof tree in the *succinct types* calculus to decide the type inhabitation problem.

Theorem 2.5.2 (Soundness and Completeness) *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then the following holds:*

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

This theorem is amenable to justification of soundness and completeness of the overall approach and synthesis process. With respect to the implemented reconstruction algorithm, a term will be reconstructed (and thus a candidate snippet will be offered to the developer) if and only if the term can be derived in λ-calculus.

From the fact that InSynth reasons only about the subset of Scala encodeable in the λ-calculus, the soundness and completeness of proof directly applies.

2.5.2 Soundness and completeness of the implementation

To claim that the implementation of the approach to synthesis driven by types is correct, we show that both modules that comprise InSynth as well as their coupling satisfy correctness properties given in the previous section.

The implementation of the pattern synthesis algorithm directly follows the algorithm for finding type inhabitatnts in *succinct types* calculus presented in Section 2.4.2. The algorithms incrementally searches the space of type inhabitants by applying rules given in Figure 2-3, thus it can be easily shown that it preserves both soundness and completeness.

In order to be correct, the reconstruction needs to reconstruct encoded solutions (and only those) into valid code snippets in the domain language. Generated code snippets are valid if under an assumption that proof trees obtained from the type inhabitation solver phase encode valid expressions and include correct program declarations (this follows from the correctness of the solver), the reconstruction phase synthesises code snippets that, when inserted at the given program point, type-check to the given type and the overall program compiles successfully. We need to show that soundness and completeness theorem defined and proved in Section 2.5.1 holds with respect to the implementation of the reconstruction phase.

We will claim that each step of the reconstruction phase retains the correctness properties:

weighted search This step performs pruning of the initial proof tree in a way that guarantees at least N expressions to be combined[8]. The pruned proof tree represents a subset of nodes from the original proof tree in such a way that the structure of that subtree is preserved, thus the set of expressions encoded in the pruned proof tree must be a subset of set of expressions encoded in the original proof tree.

intermediate transform The intermediate transform directly follows the definition of *succinct types* calculus rules for the transformation. Thus, the correctness of this step directly depends on the correctness of the *succinct types* calculus rules and the correspondence between encoded succinct and program declarations. Under the assumption of correctness of the extraction phase and the type inhabitation solver phase, since the transformation is done according to actual program declarations, the intermediate transform trees encode the correct expressions, i.e. type inhabitants.

[8]if at least N type inhabitants are found and encoded

code snippet generation This step relies on properties of the target programming language (Scala) and a valid transformation defined by programming language syntactic rules and syntactic sugar instances thus the correctness properties must be preserved.

Quantitative type inhabitation

An important correctness property that complements completeness and soundness is the one that states that the synthesis approach driven by types and weights solves the quantitative type inhabitation problem (as presented Section 2.3.4). With respect to the weight function that gives a non-negative value to each program declaration visible in the scope, when given number of N desired code snippets as a parameter, the synthesis indeed returns only $max(N, m)$ code snippets, where m is the total number of valid type inhabitants, that have the lowest weights and ranks them accordingly. From the observation that both the search for type inhabitants and code snippet reconstruction is directed by weights, we can conclude that this property is satisfied.

2.5.3 Completeness of synthesis of expressions in long normal form

The *succinct types* calculus rules, given in Figure 2-3, define derivation of terms in the *long normal form*. This effectively means that the set of derivable terms cannot include all terms expressible in the λ-calculus and Scala due to the inherent constraints of the *long normal form*. This includes all terms with functions, lambda terms in simply typed λ-calculus, at the top level. Note that abstractions terms can be generated, but only in places of a direct application to terms found in the initial environment.

An important question is whether we can achieve completeness of synthesized code in the Scala language when deriving new terms with *succinct types* calculus rules and whether this is true if we limit our reasoning to purely functional subset of the Scala language. Due to the limitations of *succinct types* calculus to generate only expressions in the long normal form, a straightforward conclusion is that InSynth is not complete with respect to expressive power of the Scala language. Furthermore, we can show that our approach is not complete even when we consider extensional equality between terms. Extensional equality captures the mathematical notion of the equality of functions: two functions are equal if they always produce the same results for the same arguments [75]. The extensional equality of two terms translates to equivalence of behavior of the two expressions under the η-reduction rules [58]. Although, due to *succinct types* calculus rules defined in Figure 2-2, InSynth can synthesize only terms in the *long normal form* and is not capable of synthesizing every possible expression, nor its extensional equivalent, that could be otherwise typed by the developer. The specific treatment of application terms limit the expressiveness of application terms to an identifier (or a bound variable).

We argue that InSynth can synthesize behaviorally (extensionally) equivalent expressions in the *long normal form* of all expressions otherwise typable by the developer, only when we restrict our reasoning to the purely functional subset of Scala. The simple expression $(\lambda x : Int.x)\ y$ which corresponds to Scala code snippet *((x:Int) => x)(y)* (which type-checks successfully to *Int* if *y*: *Int*) cannot be synthesized in InSynth, but its behavioral equivalent $(\lambda x : Int.x)[x \to y]$ or in Scala, just the literal *y*, can. In the cases in which we do not restrict the domain of Scala to its purely function subset, term *y* can include side effects that modify the environment (e.g. mutate a data structure or do input/output). If that is the case then the observed results in cases of evaluating *((x:Int) => x)(y)* and *y* may not be the same.

The aim of our synthesis procedure was to achieve practical value and good performance of synthesizing all code snippets that can be useful to developers in practice. Inclusion of *succinct types* calculus allowed us to achieve this but also restricted us to synthesizing expressions in the *long normal form* and remain complete only in the domain of purely functional subset of Scala. However, from our experience and from the results of the evaluation presented in section 2.6, we can conclude that we achieved these main goals even with these imposed constraints.

2.5.4 Example of S combinator synthesis

It is interesting to analyze the expressiveness of InSynth by giving it to synthesize desired types that could lead to synthesizing combinators from the SKI combinatory logic [17][9]. Combinatory logic may be viewed as a subset of lambda calculus, the theories are largely the same, becoming equivalent in the presence of the rule of extensionality. The SKI combinatory logic contains the same expressive power as lambda calculus and the logic is variable free, i.e. the abstractions are not part of the logic. Combinators from the SKI combinatory logic can be composed to produce combinators that are extensionally equal to any lambda term (and therefore to any computable function whatsoever). The process of obtaining an expression in combinatory logic from any given λ-calculus term can be achieved with the *abstraction elimination* procedure [75]. InSynth is capable to synthesize combinators from the

[9]many examples in the literature refer to the combinatory logic as SKI, in spite of the fact that combinators S and K provide completeness of the theory, while I can be expressed as I = S K K

SKI combinatory logic, when the appropriate desired type is specified.

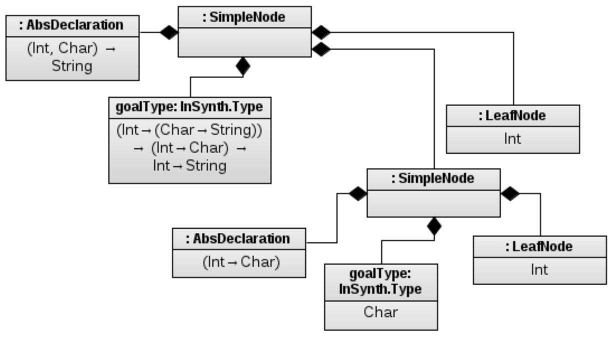
We will give an example of the reconstruction process in InSynth, show its intermediate outputs and final results, for the case of synthesis of the S combinator from the SKI logic [75]. The S combinator is defined as $(S\ x\ y\ z) = (x\ z\ (y\ z))$ and by its nature it does not require any predefined program declarations, i.e. it can be synthesized with the empty program environment[10].

The Scala declaration with type of the S combinator (instantiated appropriately with Scala types *Int, Char, String*) can be given as

```
val S:(Int=>(Char=>String))=>(Int=>Char)=>Int=>String =
```

With an empty initial environment this type has only one type inhabitant which is encoded by a succinct pattern found in three steps of the pattern synthesis algorithm.

The resulting proof tree as the output of the resolution phase is given in the following figure and it represents nested applications of terms introduced in the context (by the Abs rule).



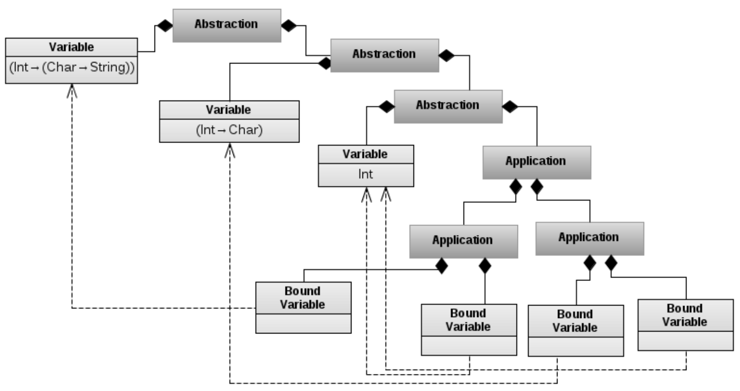
In this particular example the weighted search phase does not affect the tree since there is only one valid expression to be synthesized. The intermediate transformation phase produces a tree that in this case completely corresponds to the λ-calculus encoding of the S combinator.

The last phase of the reconstruction phase produces the following Scala code snippet which successfully type-checks when inserted as the definition of the declared value S.

2.6 Evaluation of the effectiveness of InSynth

In this section we will present results of the evaluation of effectiveness of InSynth when synthesizing code snippets in practical scenarios and by analyzing statistics

[10]the S combinator is structurally the most complex from the SKI combinator set



```
val S:(Int=>(Char=>String))=>(Int=>Char)=>Int=>String =
{
 (var_1: ((Int) => ((Char) => String)))
 =>
 {
  {
   (var_2: (Int) => Char)
   =>
   { { (var_3: Int) => { var_1 (var_3) (var_2 (var_3)) } } }
  }
 }
}
```

of obtained results, argue that InSynth is able to synthesize valid and desired snippets and can indeed be useful to developers in practical software development. This section focuses on benchmarks that measure how effective InSynth is in synthesizing code snippets that are removed from existing source code. Note that benchmarks that use InSynth for synthesis and focus on synthesizing correct code with respect to specifications is presented in Chapter 4.

2.6.1 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 the goal expressions that need to be reconstructed (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 [67]. We measure whether a InSynth can reconstruct an expression equal to the one removed, modulo literal constants (of the integer, string, and boolean type). Our benchmark suite is available for download from the InSynth web site.

2.6.2 Corpus for computing symbol usage frequencies

Our algorithm searches for those typed terms that can be derived from an initial environment and that minimize the weight function. To compute initial weights we use the technique presented in Section 2.3.4. This technique requires, among other things, an initial assignment of weights to certain terms. In order to derive the knowledge corpus which dictates this initial weight assignment, we mined declaration usage statistics from 18 Java and Scala open source projects[11]. 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. In 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 `&&`).

[11]note that these Scala projects involve heavy usage of the common Java API

2.6.3 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.

2.6.4 Measuring overall effectiveness

In each benchmark, InSynth was invoked at valid program points corresponding to the missing (goal) expressions. InSynth was parametrized with $N = 10$ and used a time limit of $0.5s$ seconds for the core type inhabitation solver and $7s$ for the overall reconstruction process . By using a time limit, our goal was to evaluate the usability of InSynth in an interactive environment (what IDEs usually are).

We ran InSynth with the aforementioned configuration on the set of 50 benchmarks. Results are shown in Table 2.5. The *Size* column represents the size of the goal expression in terms of number of declarations in its structure, as c/v, where c is the size when coercion functions are counted and v is the size with respect to visible declarations. The *#Initial* column represents the number of initial type declarations that InSynth extracts at a given program point and gives to the solver (size of the search space). The following columns are partitioned into three groups, one for each variant of the synthesis algorithm - the algorithm with no notion of term weights, the algorithm with term weights but without the knowledge corpus (presented in Section 2.6.2) and finally the full algorithm, with weights application of the knowledge corpus for initial weight assignments. In each of these groups, *Rank* represents the rank of the expression equal to the goal one, in the expression list returned by the algorithm, and *Total* represents overall execution time of the synthesis algorithm. The distribution of the execution time between two main parts of the algorithm is shown in columns *Prove* and *Recon*, for the prover and reconstruction part, respectively. The last column group gives execution times of two state-of-the-art intuitionistic theorem provers (`Imogen` [49] and `fCube` [21]) employed for checking provability of inhabitation problems for the benchmarks, encoded as formulas in appropriate syntax.

Table 2.5 clearly shows the differences in both effectiveness and execution time between the variants of the algorithm. Firstly, 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 weigh function to guide the search, the search space explodes while the reasonable solutions are not found due to maximum snippet and/or time limit. Secondly, we can see that adding weights to terms helps the search drastically and the algorithm without corpus fails to find the goal expression only in 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.

The times for `Imogen` and `fCube` provers shown in the table are the measured execution times of checking provability of benchmarks encoded as appropriate formulas. The encoding was produced from initial declarations visible at the corresponding program points (that are otherwise fed to InSynth). We can see that the difference in times spent in the *Prove* part of InSynth and those of `Imogen` and `fCube` is not negligible and in favor of InSynth - up to 2 orders of magnitude in case of `Imogen` and up to 4 orders of magnitude in case of `fCube`. Reconstruction of terms in `Imogen` was limited to 10 second and `Imogen` failed to reconstruct a proof within that time limit in all cases. The results show that, in case of the full weighted search algorithm with knowledge corpus, the goal expressions appear in the top 10 suggested snippets in 48 benchmarks (96%). They appear as the top snippet (with the rank 1) in 32 benchmarks (64%). Note that our corpus (Section 2.6.2) is derived from a source code base that is disjoint (and somewhat different in nature) with the one used for benchmarks. This suggests that even a knowledge corpus derived from unrelated code increases the effectiveness of the synthesis process; specialized corpus would probably further increase the quality of results.

In summary, the expected snippets were found among the top 10 solutions in a large number of cases and in a relatively short period of time (on average just around 145ms). These results suggest that InSynth is effective in quickly finding (synthesizing) the desired expressions at various places in source code.

	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
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 2.5: Results of measuring overall effectiveness. 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 knowledge corpus, and with weights and knowledge corpus (with the distribution of execution time between the engine and reconstruction parts). The last two columns show execution time for checking provability using Imogen and fCube provers.

Chapter 3

Lazy approach to reconstruction

In this chapter we present the idea of lazy enumeration of reconstructed type inhabitants. The idea aims to replace the eager approach implemented for the reconstruction phase of InSynth, presented in Section 2.4.3, in order to allow reconstructing terms without shortcomings introduced by the eager approach. We call the reconstruction presented in previous chapter eager, because it requires the number of desired code snippets N as a parameter and reconstructs N snippets at once, by traversing a subtree of the given proof tree that is guaranteed to encode N type inhabitants of the highest rank. Note that the traversal requires the number of desired snippets as a parameter and always tries to reconstruct a list of snippets eagerly. This may pose a problem if an appropriate parameter N is not known a priori or if there is an infinite number of encoded type inhabitants to reconstruct.

Lazy enumeration of reconstructed expressions is implemented within the InSynth reconstruction phase in order to allow integration into progressive synthesis and verification steps that is used in the core of our approach to synthesizing correct code with respect to specifications, presented in Chapter 4. It represents an important modification to synthesis driven by types and weights and is required in order to make the synthesis driven by specifications practically feasible with InSynth as an underlying synthesizer.

We present two flavors of enumeration of reconstructed type inhabitants: unordered and ordered. Both approaches guarantee that a type inhabitant is eventually enumerated if it is encoded in the proof tree, while the ordered flavor additionally allows imposing an ordering on the enumeration of inhabitants according to an arbitrary weight function.

3.1 Lazy enumeration of lambda calculus terms

This section presents techniques for achieving lazy enumeration of reconstructed type inhabitants. Using techniques for building a stream that traverses the proof tree incrementally, on demand, only when the next element is enumerated, we define an approach that can be used for the implementation of reconstruction module, presented in Section 2.4, to allow lazy enumeration of reconstructed code snippets.

3.1.1 Motivation

The reconstruction phase of the synthesis algorithm presented in Chapter 2 uses the set of found patterns (proof trees that witness succinct terms) and the original type environment to reconstruct lambda terms that these pattern encode with respect to the original type environment. The reconstruction algorithm is presented in Section 2.4.3. This algorithm is eager, in the sense that it tries to reconstruct a certain number of solutions while being bounded by a certain input parameter (the maximal depth of the tree traversal during reconstruction or the needed number of terms to reconstruct).

Although InSynth succinct trees can encode infinitely many lambda terms that are in long normal form, the algorithm presented in Section 2.4.3 needs to terminate regardless of how many lambda terms are encoded (and thus can be reconstructed). The termination of the reconstruction algorithm is guaranteed by specifying the (finite) depth d for the reconstruction process so that it terminates after following at most d edges from the root of the proof tree (that is the chain of function invocations is at most d) or after reconstructing a (finite) number n of terms (as it is actually implemented). Both of these parameters that ensure termination incur the same limitation to the algorithm.

Due to the eager nature of the reconstruction algorithm, there is a significant limitation, inherent to it. The reconstruction algorithm enumerates all lambda terms that can be constructed by traversing succinct trees up to the specified depth (or until the specified number of terms is found) thus the reconstruction is done eagerly and can only be bounded by the given input parameter. This means that in cases where the parameter is not specified (or the bound is large), the reconstruction algorithm reconstructs will try to reconstruct all encoded solutions. In the cases in which succinct trees encode infinitely many solutions, this procedure will not terminate. For practical purposes the eager approach is usually sufficient and this inherent limitation does not impose problems in finding useful code snippets (in Section 2.6 it was shown that the reconstruction algorithm achieves very good results). However, there are cases in which there is no predefined number that is expected to lead to reconstructing (enough) useful terms and in those cases this poses a serious limitation. The idea behind the approach to code generation driven by specifications can produce such cases (and it usually does since the motivation is to synthesize recursive functions which incur infinitely many expressions for synthesis). It relies on the ability to progressively reconstruct terms regardless of how many terms are encoded in the proof tree.

In the following section we will present an example of a synthesis problem on practical example in which this limitation prevents reconstructing useful expressions.

Motivating example

We will present an example in which the eager reconstruction algorithm cannot efficiently reconstruct the required number of terms. For the following program, the produced proof tree contains many recursive edges. This usually encodes that an application term f, f can be applied to itself, so that e.g. the result $f(f(f(\ldots)))$ is a valid term of the right type. The eager algorithm searches for a proof subtree that encodes sufficient number of terms, and in this example, it needs to traverse recursive edges to inspect terms encoded by them. As we will show, with traversing recursive edges, the number of terms eligible for reconstruction from the visited nodes can dramatically explode.

The following example represents code for concatenation of two lists, where lists are represented with their common algebraic data type representation usually found in functional programs (functions *Cons* and *Nil*).

Algebraic data type of lists (which is recursive) can be given in Scala as:

```
sealed abstract class List
case class Cons(head: Int, tail: List) extends List
case class Nil() extends List
```

Listing 3.1: Definition of lists

Next, we define a function that returns content of a list as:

```
 def content(l: List) : Set[Int] = l match {
   case Nil() => Set.empty
   case Cons(head, tail) => Set(head) ++ content(tail)
 }
```

Listing 3.2: Content of a list

And define a function for concatenation of two lists as[1]:

```
def concat(l1: List, l2: List) : List = ({
  (l1, l2) match {
    case (Nil(), _) => l2
    case (_, Nil()) => l1
    case (c1: Cons, c2: Cons) =>
      Cons(c1.head, concat(c1.tail, c2))
  }
}) ensuring(res => content(res) == content(l1) ++ content(l2))
```

Listing 3.3: Concatenation of two lists

Notice that the function in this example is implemented as a (complex) control flow (*match*) expression and represents the class of programs that we aim to synthesize in our approach.

Now, if we try to synthesize the expression of the third case statement, i.e. invoke InSynth at the placeholder of *hole* in the following code:

```
def concat(l1: List, l2: List) : List = ({
  (l1, l2) match {
    case (Nil(), _) => l2
    case (_, Nil()) => l1
    case (c1: Cons, c2: Cons) => ▌
  }
}) ensuring(res => content(res) == content(l1) ++ content(l2))
```

Listing 3.4: Code with a cursor for synthesis invocation

The code snippet that we removed, $Cons(c1.head, concat(c1.tail, c2))$ cannot be reconstructed because of the explosion in the number of (distinct) terms that need to be included in the set of reconstructed terms. Although the eager reconstruction does pruning of proof trees based on the number of terms already found in the search and their weights, this does not help because the explosion occurs before the pruning can take effect.

In spite of this example being simplified (declarations were removed to narrow down the initial environment), it reflects the issues of eager reconstruction in cases of recursive programs and the explosion of number of terms to reconstruct.

The proof tree produced as a solution to this instance of the type inhabitation problem in succinct calculus is given in the following figure.

[1]concatenation function could be defined with only two cases, but the given one can return early if either of the two argument lists is Nil

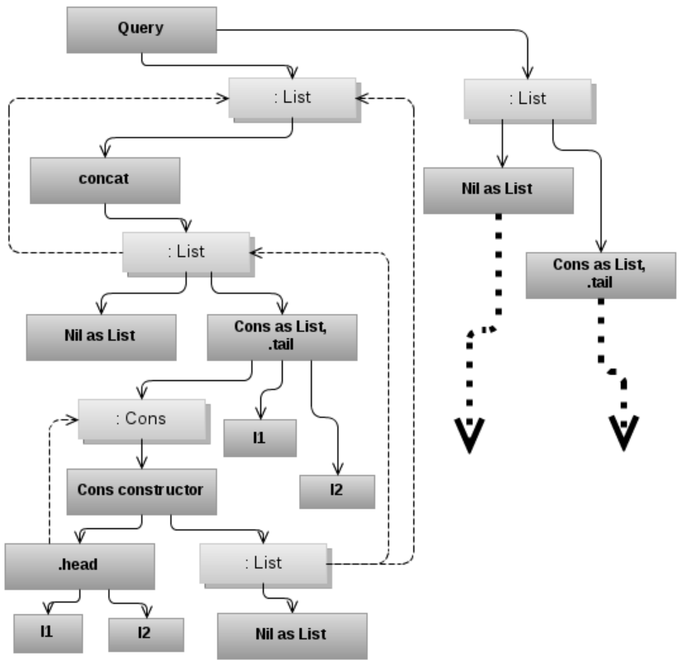


Figure 3-1: Proof tree produced for synthesis in the context as shown in Listing 3.4. Recursive edges are depicted as thin dashed edges, while thick dashed edges represent ommited subtrees.

Although the tree looks relatively small, we can see that a part of the proof tree contains a lot of edges. The proof tree encodes the information that in order to inhabit the ⊥ type we need to apply a term to the *Query* (a unique declaration introduced for the synthesis algorithm, as shown in Section 2.4.1). This term needs to be of type *List* and there are three subtrees that can reconstruct a term of type *List* - the ones that apply *concat*, coercion from *Nil* and coercion from *Cons* at top level. Only the first subtree is denoted in full in the figure, while the other two subtrees are omited for brievity. We can see that the first subtree can be used to reconstruct terms such as *concat(Nil, l1.tail)* without following recursive edges and *concat(Nil, concat(Nil, l1.tail))* when recursive edges are followed. Note that although some subtrees represent application of terms (e.g. the figure shows two nodes that apply coercion function *Cons* as *List* and *tail*), they cannot be merged into a single node in general since they are used in different contexts which may have different type environments. Also note that some two reconstructed terms may be different but behave equivalently (e.q. $Cons(l1.head, l1.tail).tail$ and $l1.tail$).

We can get a feeling of the explosion of the terms to be reconstructed by counting the number of different terms encoded by the proof tree presented in Figure 3-1. Without following recursive edges, we can reconstruct 90 different terms in total. When following each recursive edge only once, we can reconstruct 47568 terms. When following recursive edges two times this goes to staggering 11370283916 terms! If for reconstructing the desired snippet we need to follow recursive edges two times (e.g. in cases of expressions with two recursive calls chained) the synthesis may become practically unfeasible.

We conclude that the eager approach to reconstruction of terms is not suitable in cases when we do not have a fixed, specific number of solutions in mind and even practically unfeasible in some cases. The approach to code synthesis driven by specification has the idea of progressively getting code snippets from a source, such as the synthesis process driven by types and weights used in this example, and inspecting them to determine whether they are useful or not. It therefore cannot predetermine any particular (finite) number to pass to the eager reconstruction algorithm (even if it could, the approach would not be practical). A different approach to term reconstruction is needed. This leads to an idea to reconstruction approach in which reconstructed terms could be lazily enumerated, such that no term is reconstructed (and corresponding subtrees of the proof tree are visited) if they are not explicitly asked for, i.e. if they are not enumerated. Regardless of the limitation of eager reconstruction, addressing this problem by allowing lazy traversal and enumeration represents an interesting (and by no means easy) problem from both the theoretical perspective and implementation.

3.1.2 Algorithms

The goal of lazy enumeration is to provide a systematic way of ordering resulting terms of the reconstruction process in a way such that the terms can be progressively reconstructed and enumerated. While traversing the proof tree, the reconstruction is performed only on the those subtrees that need to be used for reconstructing of the term being enumerated. This means that terms can be enumerated one by one, in some predefined order, and the actual reconstruction is performed only when needed.

The idea of lazy enumeration is closely related to the lazy evaluation in programming language theory. Lazy evaluation delays evaluation of procedure arguments until the last possible moment (e.g., until their values are required by a primitive operation) and which allows avoiding of repeated computations (i.e. sharing) [3]. As we will see next, the implementation of lazy enumeration of expressions from proof trees is natural and easier to define with the lazy evaluation semantics (which Scala language supports). Lazy evaluation corresponds to call-by-name argument passing defined in the operational semantics [58]. Lazy evaluation combined with memoization is sometimes referred to as call-by-need argument passing, in contrast to call-by-name argument passing (implementations of call-by-name are similar to non-memoized lazy evaluation) [3].

Two most important benefits of lazy evaluation that are amenable to our approach to lazy enumeration of reconstructed terms are the increase of performance due to avoiding repeated reconstruction of same subtrees of the proof tree and the ability to construct an enumerable lazy stream (sometimes called lazy list [3]) that can encode infinite number of reconstructed terms.

Scala provides the possibility to construct lazy streams [53]. In Scala, streams can be constructed with a value (or a sequence of values that represent the intially enumerated elements) together with a function that is invoked when new elements need to be enumerated. Having such mechanism for constructing streams is very useful and provides means for construcing stream that enumerate inifinite sequences of values (the provided function can generate infinitely many values). The *Stream* class in Scala incorporates these mechanism and also employs memoization such that previously computed values can be stored and converted from Stream elements to concrete values [53].

The main goal is to allow traversal of any subtree of the proof tree and reconstruction of partial expressions only when needed and their memoization, so that those partial expressions can be reused later. This allows progressive traversal and reconstruction process when the values need to be enumerated and amortization of the cost of each subterm reconstruction over total number of times the subterm appears in the reconstructed terms.

Stream utilities

In order to define the reconstruction process that produces a stream of terms which allows lazy enumeration, we present few simple objects that are used in the reconstruction process. The goal of these algorithms is to construct stream objects from the given parameters which encode appropriate streams. Classes of these objects in Scala implement a *Streamable[T]* trait which produces a stream of values of type T.

We will denote a stream that enumerates values $a, b, c, \dots$ in that order with $\langle a, b, c, \dots \rangle$.

Singleton stream This stream encapsulates a single value into a stream (of finite size 1) and serves for transforming leaf terms in the reconstruction.

Algorithm 5 Singleton stream

```
Require: value $v$
1: return $\langle v \rangle$
```

Round robbin stream Produces a stream of values out of streams received as inputs, such that every value that could be enumerated by any of those input streams will also be enumerated by the produced stream, at some point of enumeration (eventually).

Algorithm 6 Round robbin

```
Require: array of streams $s_1, \ldots, s_n$
 1: for each $s_i, i = 1, \ldots, n$ make a stream iterator $it_i$
 2: $ind = 0$, values = $\langle\rangle$
 3: while at least one iterator has next do
 4:     if $it_{ind}$ has next then
 5:         add next value of $it_{ind}$ to values
 6:         forward iterator $it_{ind}$
 7:     end if
 8:     $ind = (ind + 1) \bmod n$
 9: end while
10: return ⟨ values ⟩
```

For the sake of simpler presentation, stream of values computed by Algorithm 6 that belong to the stream are given as being computed eagerly (and such that this computation may not halt), but the actual implementation constructs a Scala *stream* with a function that gets one value. This process is repeated on each value enumeration and the values are streamed lazily, on demand.

Mapper stream Mapper stream takes a stream s of type T and a mapping function $f : T \to U$ and produces a stream with values from s mapped with f.

Algorithm 7 Mapper

```
Require: stream $s$, mapping function $f : T \to U$
 1: { apply $f$ to each enumerated value of $s$ to produce a lazy stream }
 2: return ⟨ f(s(0)), f(s(1)), f(s(2)), ... ⟩
```

It represents a simple but necessary algorithm which is needed for constructing the solution stream in the reconstruction phase.

Binary stream Binary stream takes as input two streams, enumerating values of types T and U, and a function, a binary operator f of type $(T, U) \to V$ and produces a stream of values of type V which represent application of f to each combination of values that can be enumerated from these two input streams.

The algorithm is given in a high level for the sake of simpler presentation. The main idea is to split the construction of the resulting stream into construction of

Algorithm 8 Binary stream

```
Require: non-empty streams $s_1, s_2$, binary operator $f$
1: let $n_1, n_2$ be the lengths of streams $s_1, s_2$ respectively
2: let $s_a$ be a stream of values $\langle f(s_1(1), s_2(1)),\ f(s_1(1), s_2(2))\ ,\ f(s_1(2), s_2(2)), f(s_1(2), s_2(3)), \ldots\rangle$, i.e. values $f(s_1(i), s_2(j))$ where $i = 1, \ldots, n_1,\ j = i, \ldots, n_2$ {combinations of all values $s_1(i)$ and $s_2(j)$ where $i \leq j$}
3: let $s_b$ be a stream of values $\langle f(s_1(2), s_2(1)),\ f(s_1(3), s_2(1))\ ,\ f(s_1(3), s_2(2)), \ldots\rangle$, i.e. values $f(s_1(i), s_2(j))$ where $i = 1, \ldots, n_1,\ j = 1, \ldots, i-1$ and $j < n_2$ {combinations of all values $s_1(i)$ and $s_2(j)$ where $i > j$}
4: return stream of values $\langle s_a(1), s_b(1), s_a(2), s_b(2), \ldots\rangle$
```

two streams - s_a and s_b in the algorithm. Let us denote ranks of enumerated values from s_1 and s_2 with i and j. Stream s_a contains results of f applied to combinations of all enumerated values from s_1 and s_2 where $i \leq j$. Similarly stream s_b contains combinations of all enumerated values where $i > j$. Due to these conditions on i and j are disjunctive, the two streams s_a and s_b enumerate disjunctive combinations of values from s_1 and s_2 and alternating between enumerating values from them produces the needed resulting stream. The resulting stream enumerates applications of f to all combinations of values with ranks i and j (where $i \leq j$ together with $i > j$) so each needed value can be enumerated eventually. Note that we denote lengths of streams by n_1 and n_2. Even though streams may be infinite, the resulting stream is such that values are enumerated lazily so this does not impose an issue in the implementation.

Binary streams are used in reconstruction of function applications, where we want to enumerate all possible combinations of parameters applied to a function. Note that this stream can be infinite (if a parameter can be an application of the same function) that is why its values have to computed lazily.

3.1.3 Reconstruction using streams

To achieve the goal of streaming reconstructed terms, the reconstruction phase of InSynth needs to be altered to construct lazy streams. The process of reconstruction using streams is done after the intermediate transformation step (Section 2.4.3), i.e. it takes as an input the intermediate representation of proof trees (like the example given in Figure 2-9) and produces a stream of reconstructed lambda terms. An intermediate representation tree represents a proof tree translated to encode terms λ-calculus. It can encode many terms and the size of produced stream is the number of those encoded terms. Now, instead of eagerly traversing the intermediate trees and producing exact number of code snippets, the process returns a stream of trees (that represent terms in λ-calculus) which can be easily transformed to a stream of code snippets by mapping that stream with a function that translates an individual lambda term to its corresponding code snippet in the domain language.

The algorithm that transforms a subtree of intermediate representation tree and constructs a stream of terms reconstructed from that subtree is given in Algorithm 10.

Algorithm 9 Recursive stream construction (*rec*)

```
Require: intermediate representation node $r$
1:  switch (term type of $r$)
2:  case leaf term $t$:
3:      {$t$ is an identifier from environment or a bound variable}
4:      return single stream of $t$
5:  case abstraction term, where $v_1, \dots, v_n$ are variables abstracted from a term:
6:      {note that our abstraction nodes can encode multiple variables}
7:      let $b_1, \dots, b_m$ be inner terms that encode body terms of the abstraction
8:      let $f_{abs}$ be a function that takes a term $t$ and returns an abstraction term in
        the form of $(\lambda v_1. \, (\lambda v_2., \, \dots, \, (\lambda \, v_n. \, t)))$
9:      for all inner node $b_i$ from $b_1, \dots, b_m$ do
10:         $s_{bi} = rec(b_i)$ {recursively call $rec$ on $b_i$ and collect the resulting stream}
11:     end for
12:     let $p_s$ be a round robbin stream made out of collected streams $\{b_{s1}, \dots, b_{sm}\}$
13:     return mapper stream that applies $f_{abs}$ to values of $p_s$
14: case application term, where $P_2, \dots, P_n$ are sets of inner parameter nodes applied
    to set of terms, represented by the set of nodes $P_1$:
15:     {note that our application can apply multiple parameters}
16:     for all inner set of nodes $P_i$ that encodes parameter terms do
17:         for all parameter node $t_i$ from $P_i$ do
18:             $ps_{i,j} = rec(t_i)$ {recursive transformation of the $j$-th element of $P_i$}
19:         end for
20:         let $ps_i$ be a round robbin made out of all streams $ps_{i,j}$ {from all streams $ps_i$,
            where $j = 1, \dots, l_i$ and $l_i$ is the cardinality of $P_i$}
21:     end for
22:     let $comb = ps_1$ {start accumulating stream of term combinations}
23:     for $ind = 2, \dots, n$ do
24:         make a binary stream $b$ out of $comb$ and $ps_{ind}$
25:     end for
26:     let $f_{app}$ be a function that takes a combination of terms $t_1, \dots, t_k$ and makes
        an application term $(t_1 \; t_2 \; \dots \; t_k)$
27:     return mapper stream that applies $f_{app}$ to values of $comb$
28: end switch
```

Stream utilities algorithms are invoked from this algorithm for constructing the of streams (for the sake of brevity, call of these algorithms are denoted in a simple manner and we always take the stream from the returned object). Note that the algorithm takes a tree in the intermediate representation, in which a node contains sets of inner nodes to encode multiple terms, and produces a stream that enumerates trees that encode single individual λ-calculus terms.

This recursive procedure constructs an appropriate stream by cases based on the type of the input term.

- In the case of a leaf node, the corresponding transformed term also represents a leaf term in the whole expression tree so only a stream with that single term is returned (line 4).

- In the case of an abstraction node, we have to enumerate an abstraction term for each body term encoded by that node. This is accomplished with a round robbin stream constructed out of all streams made by recursive reconstruction of all body term nodes (line 12) and mapping its value with a function that makes the appropriate abstraction term (line 13).

- In the case of an application node where the node is of the form $(P_1 P_2 \ldots P_N)$ (where P_i are sets of nodes) we have to enumerate all terms $(t_1 t_2 \ldots t_n)$ where t_i is a term enumerated by a stream of terms ps_i reconstructed from P_i. Since for each place of a single subterm in λ-calculus, our intermediate representation can have a set of terms and each gets reconstructed to a stream of terms, we reconstruct a set of streams for each parameter node in P_i (line 18). The solution of enumerating all possible terms that can occur in place of P_i is to construct a round robbin stream our of all those streams (line 20). All combinations of streams are then ensured by constructing a chain of binary streams out of those round robbin streams (lines 22-25). For a given value of ind, the for loop construct a stream that streams all possible combinations of terms $1, \ldots, ind$. The resulting stream is constructed with a mapping function that creates the appropriate application term (line 26) applied to values from a stream that enumerates all combinations of terms (line 27).

One important and subtle remark, which is not regarded in the case of application parameter nodes, is dealing with node links that create cycles, i.e. application terms which can have themselves or their ancestor terms as parameters (e.g. an application node $(fxf(\ldots))$ where $x, f(x), f(f(x)), \ldots$ can be applied to f). This can lead to a scenario in which when reconstructing an application term, a parameter stream can enumerate that application term. In order to avoid non-terminating enumeration due to such cases, the parameter streams must first enumerate all terms without following such recursive links (and this enumeration is guaranteed to terminate) and afterwards the recursive ones. Lazy enumeration allows us to represent infinite streams build in such a way and is required in our algorithm.

The main reconstruction algorithm calls the subterm transformation algorithm on the root node and applies a function that translates a λ-calculus term into a valid code snippet in the domain language (for Scala it is similar to the one described in 2.4.3).

Algorithm 10 Reconstructing stream of code snippets

```
Require: intermediate representation tree with root node $r$
1: let $f_{dom}$ be a function that translates a $\lambda$-calculus term into corresponding term
   in the domain language
2: return map values of $rec(r)$ with $f_{dom}$
```

Synthesis of exactly n solutions can be achieved by simply taking first n values from the result stream.

3.1.4 Soundness and completeness

We will use terms soundness and completeness for defining properties of this approach to reconstruction terms with lazy enumeration[2]. When this approach is used in the reconstruction phase in the synthesis process driven by types and weight, these properties directly affect the soundness and completeness properties of the whole synthesis process (presented in Section 2.5.1).

Let T be the set of terms encoded by an input tree, in the intermediate representation, with a root node r. Let s be the resulting stream from calling Algorithm 10 on r, with length n, and t_i be the i-th term in the enumeration of s.

We define two theorems to state the soundness and completeness properties:

Theorem 3.1.1 (Soundness) *No term encoded in the input tree is enumerated twice from the stream produced by the lazy stream reconstruction algorithm. More specifically, the following holds:*

$$\forall i, j.\ i \neq j \rightarrow t_i \neq t_j$$

Theorem 3.1.2 (Completeness) *Every term encoded in the input tree will be eventually enumerated. More specifically, the following holds:*

$$\forall t \in T.\ \exists i \leq n.\ t_i = t$$

[2]these words are being used frequently and sometimes recklessly for defining various properties

It can be shown by reasoning about the properties of the stream utilities and then reflecting them to the main reconstruction algorithm, given in Algorithm 10 that the reconstruction procedure produces a stream that respects properties defined by these two theorems.

The Completeness theorem, Theorem 3.1.2, directly affects Completeness of the synthesis driven by types, if the reconstruction phase is done with lazy stream reconstruction. It guarantees that all terms encoded in intermediate proof trees, thus by discussion in Section 2.5.1 and 2.5.2 all terms derivable in the given environment by λ-calculus rules, will be enumerated from the resulting stream at some point during enumeration. Note that the Completeness theorem also implies fair enumeration between multiple sources of infinite number of reconstructed terms (e.g. if there are two mutually recursive functions which application represents a valid reconstructed term, the enumeration process may not enumerate infinite stream of solutions where applications to the first function are reconstructed, otherwise the Completeness theorem does not hold).

3.1.5 Evaluation

It can easily be shown that for reconstructing a single term, i.e. enumerating one solution, the complexity is $O(n)$ where n is the size of the input tree - in the worst case the whole input tree needs to be visited for its reconstruction. However, due to the memoization of reconstructed subterms, the cost for traversing the tree is amortized over the number of enumerated elements. More precisely, the complexity of enumerating m terms from the reconstructed stream is $O(mn)$, but gets amortized according to memoized reconstructed subterms. This means that the asymptotic complexity of getting a specific number of reconstructed terms is greater than the complexity of eager reconstruction, which is $O(m+n)$[3]. Nevertheless, the lazy enumeration approach offers significant advantage over the eager one, in cases where only a certain a priori unknown number of terms is required. The eager approach would need to reconstruct a specific number of terms regardless of the actually needed number.

The evaluation results and comparison between eager reconstruction and reconstruction using lazy enumeration are deferred to Section 3.2.5. They show performance of reconstruction in couple of examples, when various number of terms are given as parameters to the reconstruction. These results witness the practical benefits of using lazy enumeration for the reconstruction of terms.

There are two main practical implications that the lazy enumeration approach

[3]with a conservative assumption that the input tree encodes exactly m terms

to term reconstruction brings. First one is that the approach allows imposing more control and limits on the search space of term reconstruction - unnecessary traversal of subtrees of the input tree can be avoided, while the reconstruction is done only when needed. The second one is that the enumeration imposes an ordering on the set of synthesized terms. This can make enumeration of a finite sequence of reconstructed terms faster and even feasible in cases of infinite number of encoded terms.

3.2 Ordered lazy enumeration of lambda calculus terms

This section will present a modification of the idea of lazy enumeration of reconstructed lambda calculus terms that restricts the order in which reconstructed terms are enumerated. This idea is implemented within the InSynth reconstruction to achieve enumeration of reconstructed terms ordered by term size and used to speed up the synthesis approach presented in this chapter.

3.2.1 Motivation

Although lazy enumeration of reconstructed terms provides the possibility for progressive synthesizing of terms according to some order, this order is not strictly defined and so that the order in which terms are reconstructed is arbitrary and depends on the input tree given to the reconstruction process. This may be insufficient to produce good results in cases where we need to synthesize (that is, enumerate) some desired snippet as fast as possible.

In our approach to synthesizing programs according to specifications, we use the idea of lazy enumeration to progressively synthesize terms and then to examine them and determine whether they can be useful (for construction of a correct program). Therefore, the number of terms that is enumerated and examined directly affects the performance of the overall synthesis. The goal is to be able to enumerate the desired reconstructed term as soon as possible, i.e. the desired term should have as low rank as possible.

If we return back to the motivating example for the lazy enumeration, we can evaluate the lazy enumeration technique on the code given in Listing 3.4. After implementing the lazy enumeration reconstruction algorithm and integrated it into InSynth, we ran the synthesis at the place given by the cursor. The desired code snippet was the one that we removed, $Cons(c1.head, concat(c1.tail, c2))$. After running several tests, this snippet was enumerated with the different ranks ranging from more than 3500 up to more than 10000 (note that the ordering is not deterministic and completely depends on the input tree, thus we can get different rankings). This means that a lot of examples needed to be examined, and as we will show in Section 4.3.7, this incurred high execution times of the synthesis process and made it unpractical.

3.2.2 Algorithms

The idea of improving the synthesis approach is to impose a stricter ordering on the enumeration that would result in an ability to enumerate terms that are more likely to be useful early. We observed that in the majority of scenarios the size of desired expressions (the one needed for construction of more complex expressions correct with respect to given specifications) is relatively small. This brought us to the idea of imposing ordering on the enumeration of reconstructed terms based on their size.[4]

The idea presented in this chapter extends the general idea of lazy enumeration with the flexibility of defining ordering of the enumeration. It is important that the properties that hold for the general idea of lazy enumeration, namely soundness and completeness defined in Section 3.2.4, also hold in the case of ordered lazy enumeration. Although the algorithms that achieve ordered lazy enumeration provide flexibility that allows an easy way of defining custom ordering of terms, for the purpose of the improving performance of synthesis, we will focus on the ordering based on size of terms.

We will extend the notion of stream of reconstructed terms by including an additional stream that enumerates values that are used to define the order of enumeration. We denote these values as weights. Conceptually, each stream of terms is now associated with a stream of weights. This additional stream of weights completely defines the ordering imposed on enumeration of terms thus should be defined carefully. In the case of ordered lazy enumeration of reconstructed terms by their size, the additional stream represents the appropriate stream of term sizes. This effectively means that we have a pair of streams, such that for each term enumerated from the first stream the size of that term is enumerated from the second (thus it only makes sense to enumerate these paired streams simultaneously). In the following sections we will focus on lazy enumeration of reconstructed terms that enumerates terms in a non-decreasing order of their sizes.

[4]note that such ordering may not be deterministic

Ordered stream utilities

We present a few simple objects that are used in the reconstruction process of ordered lazy streams (they are similar to their unordered counterparts, presented in Section 3.1.2). Their implementations in Scala implement same interfaces thus make the reconstruction process with streams easily configurable (one easily change between flavors of lazy enumerated streams). These objects provide an additional stream that evaluates weights and thus are able to represent pairs of streams as described in the previous section.

The algorithms presented in this section are general enough to allow imposing other ordering besides the ordering on term size. Generally, we can think of ordered streams as streaming values and weights. In our case values are reconstructed terms and weights are their sizes. We will denote an ordered stream that enumerates values $a, b, c, \ldots$ and weights $w_1, w_2, w_3, \ldots$ in that order with $\langle a : w_1, b : w_2, c : w_3, \ldots \rangle$.

In order to guarantee a non-decreasing order of enumeration, each stream is constructed such that it enumerates its terms in a non-decreasing order of their weights. This allows our algorithms to compose streams into more complex ones that respect the same guarantees on ordering.

Singleton stream This stream is analog to the unordered counterpart with an addition that it takes a single weight.

Algorithm 11 Ordered singleton stream

```
Require: value $v$, weight $w$
1: return $\langle v : w \rangle$
```

Round robbin stream Round robbin stream takes as an input a set of ordered streams and produces an ordered stream that enumerates all values from the set of input streams, ordered by their corresponding weights.

As in Algorithm 6 we describe that values are computed eagerly while in the actual implementation they represent a stream that can be lazily enumerated. Note that breaking ties is especially important for the completeness property - without it, it could happen that we have an infinite stream of values with the same, minimum weight and that stream would be always enumerated even if we had other streams with values of the same weight. Changing priorities allow us to break ties fairly in such cases.

Algorithm 12 Ordered round robbin

```
Require: array of ordered streams $s_1, \ldots, s_n$
 1: for each $s_i, i = 1, \ldots, n$ make a stream iterator $it_i$
 2: $ind = 1$, values = $\langle\rangle$
 3: while at least one iterator has next do
 4:     {let $v_i : w_i$ be the next element of $it_i$}
 5:     choose $j$ such that $w_j = \min_{k=1,\ldots,n} w_k$ where $k$ belongs to indices of iterators
        $it_k$ which have next element
 6:     {ties are handled by letting $(k - ind) \bmod n$ as a priority function}
 7:     add $v_j : w_j$ to values
 8:     forward iterator $it_j$
 9:     $ind = (j + 1) \bmod n$ {priorities change}
10: end while
11: return ⟨ values ⟩ {stream of pairs}
```

Mapper stream Mapper stream is analog to its unordered counterpart with an additional requirement that the parameter mapping function $f : T \to (U, W)$ produces pairs of values $(u, w) : (U, W)$ such that values of W are non-decreasing (function f is monotonic on weights [10]). When applied, the function must produce pairs that respect the ordering.

Algorithm 13 Ordered mapper

```
Require: stream $s$, mapping function $f : T \to U$
 1: { apply $f$ to each enumerated value of $s$ to produce a lazy stream }
 2: return ⟨ f(s(0)), f(s(1)), f(s(2)), ... ⟩
```

Binary stream Binary stream takes as input two ordered streams and a binary operator that takes two pairs of values and produces a new pair. For the binary ordered stream, it is also important but harder to guarantee a stream of values with non-decreasing weights. Combinations of values are defined by the binary operator f. f must be monotonic on weights and must produce pairs of values that respect the ordering.

Since our focus is to order reconstructed terms by their sizes, the purpose of ordered binary streams is to provide means for combining streams of parameters into a stream of their combinations. With respect to weights, the resulting stream projected on weights represent additions of weights from both streams. We will sacrifice generality for better presentation and assume that the binary operator for combining weights is fixes to be the addition operator, +, while f operates only on terms.

Algorithm 14 Binary stream

```
Require: streams $s_1 = \langle v_1(1) : w_1(1), \ldots, v_1(n_1) : w_1(n_1)\rangle, s_2 = \langle v_2(1) : w_2(1), \ldots, v_2(n_2) : w_2(n_2)\rangle$, binary operator $f$
 1: let $q$ be a queue of pairs of indices
 2: $q = (0, 0)$, values $= \langle\rangle$
 3: while $q$ is not empty do
 4:     let $(i, j) = min_{(a,b)\in q}(w1_i + w2_j)$ {pick $(i, j)$ from q such that sum of weights of those indexes from $s_1$ and $s_2$ is the least}
 5:     remove $(i, j)$ from $q$
 6:     add the pair $(f(v1_i, v2_j) : w1_i + w2_j)$ to values
 7:     if $i = j$ then
 8:         enqueue $(i + 1, j)$, $(i, j + 1)$, $(i + 1, j + 1)$
 9:     else if $i > j$ then
10:         enqueue $(i + 1, j)$
11:     else
12:         enqueue $(i, j + 1)$
13:     end if
14: end while
15: return $\langle values\rangle$
```

Note that function on weights is a simple addition operator but could be generalized with the only requirement that it needs to be monotonic on weights.

We denote accessing $i-th$ value from stream j with $v_j(i)$ while the similar notation we adopt for weights ($w_j(i)$). The algorithm starts by initializing a queue with the heading pairs from both input streams, s_1 and s_2. The algorithm proceeds by scanning the queue and picking a pair of indexes (i, j) such that sum of weights at s_1 and s_2 at given indexes is minimal. This guarantees that the produced stream respects the non-decreasing ordering on weights. For the sake of simplicity, the accesses to streams are described to made with indexes while in the actual implementation they are progressively enumerated (with iterators). Afterwards, these indexes are incremented such that all combinations of two streams are examined and the appropriate non-decreasing ordering is satisfied. For example, if we have two infinite streams in non-decreasing order, s_1 and s_2, one can always that the minimal sum of values is $s_1(1) + s_2(1)$ (i.e. the sum of head weights). The next sum by this ordering could be $s_1(1) + s_2(2)$ or $s_1(2) + s_2(1)$ depending on the actual weights. If the next two sums are $s_1(1) + s_2(2)$ and $s_1(2) + s_2(1)$ in that order, to determine the next sum, one must examine sums $s_1(1)+s_2(3)$, $s_1(3)+s_2(1)$ and $s_1(2)+s_2(2)$, etc. This process continues and guarantees, given two streams in non-decreasing order of their weights, a resulting ordered stream on sums of weights.

3.2.3 Reconstruction using ordered streams

The algorithms used in the reconstruction using ordered streams are identical to the ones presented in Section 3.1.3 (the implementation of the reconstruction relies on a common interface so that reconstruction using different flavors of streams can be changed easily and even combined). The algorithm does not have to be changed and only requires that for each leaf term in the input trees an appropriate weight of the term is associated (size 1 in the case of ordering by term size) and that the algorithm at each step constructs streams that enforce ordering. An important remark is that a special care needs to be made in cases where we have recursive function calls. As described in Algorithm 9 parameters to function application are encapsulated within an ordered round robbin stream. Since the ordered round robbin needs to check all input streams to determine the minimal weight, weights should be examined without actual enumeration of parameter streams. Otherwise, enumerating a parameter stream that contains a recursive call could lead to non-termination of the algorithm.

The output of the reconstruction phase is a stream which enumerates λ-calculus terms, in the non-decreasing order on the term size (with respect to a standard definition of the size of a term in λ-calculus [58]).

3.2.4 The ordering property

As we mentioned in the previous sections, the idea of ordered lazy enumeration of reconstructed terms should retain properties of lazy enumeration and add the restriction of ordered enumeration.

In addition to Theorem 3.1.2 and 3.1.1, the following theorem also holds in the case of ordered streams constructed by the reconstruction algorithm (we use the same notation as the one in Section 3.2.4):

Theorem 3.2.1 (Ordering) *Reconstructed terms are enumerated in a non-decreasing order on term sizes. More specifically, the following holds:*

$$\forall i, j \in \{1, 2, \ldots, n\}. i < j \rightarrow size(t_i) \leq size(t_j)$$

By reasoning on individual ordered stream constructions and the reconstruction algorithm with ordered stream, it can be shown that the resulting stream indeed represents a lazy stream of reconstructed terms for which soundness, completeness and ordering hold. An important property of all intermediate streams constructed during the whole reconstruction process is that they all enumerate terms with non-decreasing order on term sizes. An interesting remark holds for recursive calls within the input tree that can be enumerated as parameters - they inherently guarantee enumeration of strictly larger terms than any of the parameters without recursive calls. Moreover, combinations with their terms are monotonic with respect to the term size (e.g. applying stream $x, f(x), f(f(x)), \ldots$ to f results in $f(x), f(f(x)), f(f(f(x)))$).

3.2.5 Evaluation

Similarly as in the case of reconstruction using unordered streams, enumerating a single term in this case can take $O(n)$ time, where n is the size of the input tree. This is due to the fact that in the worst case, the whole tree needs to be traversed to reconstruct a term. Interestingly, in the case of reconstruction using ordered streams when a term is constructed from subterms, the algorithm needs to examine minimal weights of all subterm streams in order to determine which values should be used for the reconstruction of the next term.

The ordered round robbin needs to examine the minimal weight of each input parameter stream. This leads to the complexity of $O(p)$, where p is the number of input streams, for enumerating a single element. Moreover, the ordered binary stream algorithm combines two streams into the resulting ordered stream of their sums. This algorithm may need to examine sum of weights for $O(k)$ combinations of subterms, where k is the number of enumerated elements from the ordered binary stream. More specifically, the complexity of enumerating a combination of elements from an ordered binary stream is linear in the number of already enumerated combinations. The binary stream is used for enumerating application terms (combining application parameters) while the round robbin is used for enumerating parameters thus the upper bound is $O(p+k) = O(m)$, where m is the total number of terms possible to reconstruct from the given input tree, an upper bound of both p and k.

This means that the worst-case complexity for enumerating a single term is $O(n+m)$, that is, bounded by the size of the tree and total number of terms encoded. Enumeration of k terms from such stream raises the complexity bound to $O(k(n+m))$. Although the memoization of reconstructed subterms is employed, this does not help the worst case - the cost for traversing the tree and inspecting weights of subterms is amortized over the number of enumerated terms that are constructed from them, but in the worst case, all subterms need to be inspected.

The asymptotic complexity of getting a specific number of reconstructed terms is greater than for the cases of eager or unordered lazy reconstruction. However, the complexity in the average case is far away from the pathological worst case and memoization drastically improves performance. We can conclude that performance in practice correspond to average case complexity from the results of the evaluation presented in the next section.

Evaluation on benchmarks

Table 3.1 presents comparison of performance between the eager reconstruction and reconstruction using two presented flavors of lazy enumeration, unordered and ordered. The platform that we used for running the experiments was identical to the one described in Section 2.6.

Example code	Reconstruction method	# reconstructed expressions 10	100	1000	10000	100000	average speedup
List concatenation	Eager	73/544	513/825	N/A	N/A	N/A	1.00
	Unordered	174	238	1093	5688	165975	3.30
	Unordered (mem.)	16	58	356	3439	101505	24.11
	Ordered	30	41	208	1279	14093	19.13
	Ordered (mem.)	13	27	93	658	7841	36.20
Finite combinations	Eager	140/144	135/139	164/168	642/647	N/A	1.00
	Unordered	24	83	654	7392	95069	2.00
	Unordered (mem.)	14	43	358	4613	64725	3.53
	Ordered	19	29	115	1201	12894	3.59
	Ordered (mem.)	17	22	64	1078	9158	4.50
Recursive calls	Eager	38/84	5082/5084	N/A	N/A	N/A	1.00
	Unordered	7	62	946	N/A	N/A	47.00
	Unordered (mem.)	4	33	677	N/A	N/A	87.53
	Ordered	2	15	989	N/A	N/A	190.47
	Ordered (mem.)	1	9	407	N/A	N/A	324.44

Table 3.1: Results of the evaluation of the lazy enumeration approach to reconstruction.
The first column partitions the results of three examples in the evaluation and gives their names. The second column denotes the reconstruction method (note that mem. denotes enumerating from a stream with memoized elements). The following columns represent the execution time of the reconstruction process in milliseconds. The last column denotes the average speedup of each reconstruction method when having the eager approach as baseline.
Note that *N/A* denotes that reconstruction could not finish and the average speedups do not consider these values. The times for eager evaluation are given as x/y, where x and y are execution times with and without inclusion of the time spent in the combinator phase, respectively.

There are three examples that were used in the evaluation. They were chosen such that each reflects a category of cases that can occur when reconstructing terms during a synthesis process:

List concatenation This example reflects the majority of cases that occur in practice. It is explained in more detail in Section 3.1.1 and represents one of the examples that motivated the approach to reconstruction using lazy enumeration. The synthesized terms for this examples are encoded by the proof tree that is not negligible in size, neither in terms of height nor width. It contains recursive edges that represent recursive calls in multiple places thus it encodes an infinite number of terms.

Finite combinations This example reflects an extreme in which the tree that encodes terms is relatively small but the number of encoded terms is big due to many possible combinations of subterms that can be used in for reconstruction of the resulting term. The number of terms that can be reconstructed in this example was set to 100000.

Recursive calls This example reflects the case in which recursive calls are encoded. The proof tree is small and contains only one non-recursive and one recursive edges. This examples directly implements the case in which we can synthesize $x, f(x), f(f(x)), f(f(f(x))), \ldots$ as solutions. It presents to what extent can reconstruction approaches deal with encodings of recursive calls.

We measured time for 5 reconstruction approaches: eager approach (presented in Section 2.4.3), unordered and ordered lazy enumeration (presented in last two sections) together with their modifications in which a half of the total number of terms to reconstruct are enumerated before the measured experiment (denoted with (mem.) in Table 3.1). The idea for last two approaches is to evaluate the effects of memoization and benefits it provides in cases a stream enumeration is stopped at some point and then restarted. Note that in case of eager reconstruction this is not possible and restarting the reconstruction comes with no benefits.

We measured time needed for reconstruction of 10, 100, 1000, 10000, and 100000 terms. The resulting times are given in milliseconds. With N/A we encoded cases in which the reconstruction phase failed to return (the reason in all cases was exceeded memory limits during the reconstruction of the program, which was set to 2GB). Note that the total exectuion time for the eager approach is presented as x/y, where x and y are execution times with and without inclusion of the time spent in the combinator phase, respectively. This was done to emphasize the impact that the combinator phase, that explores the initial proof trees and prunes them, has on the whole reconstruction. Additionally, in the last column we give average speedups of all approaches to reconstruction with respect to times needed for the eager approach, including the combinator step. Note that average speedups were calculated over all specified number of terms to reconstruct for which the eager approach terminated (the time is not denoted with N/A).

We can conclude that employing lazy enumeration for reconstruction allowed us to achieve reconstruction in several cases in which the eager approach was not feasible. Furthermore we can see that the lazy enumeration approach outperforms the eager one in almost all cases. This is due to two facts: one is that the combinator step, which is not needed in case of lazy enumeration, can take significant amount of time since it needs to traverse the proof tree and prune it; and the second one is that requiring a specific number of combinations can result in producing more terms than the required number, thus incurring greater overhead. Note that for the finite combinations example, in which the tree does not require pruning but encodes a lot of combinations, once we go over 100 terms to reconstruct, the eager phase is faster since when it reaches a certain point, it quickly combines many subterms. On the other hand, the overhead for invoking reconstruction per each enumerated term becomes significant in the case of lazy enumeration. Finally, the recursive calls example shows poor performance of eager reconstruction which is due to the fact that it effectively represents a generation and traversal of a degenerate tree of height equal to the number of terms to reconstruct.

An interesting remark is that the ordered lazy enumeration, while employing mechanisms that incur greater worst-time complexity and more overhead than the unordered counterpart, outperforms the unordered counterpart in almost all cases. This is due to the fact that ordering terms by their size results in less time needed for the actual reconstruction (which directly depends on the size of the term that is reconstructed).

We can see that ordered lazy enumeration justifies its motivation and not only provides ordering of terms by their size, that can be useful for the synthesis approach but also offers better performance because of this ordering.

Ordered lazy enumeration approach offers the same advantages over the eager reconstruction and makes a lot of reconstruction instances feasible in practice, as we can see from the presented results. Moreover, the additional imposed restriction on the order of the enumeration adds an important value since it enforces (some degree of) determinism (and predictability) into the enumeration. This ordering restrictions bring significant improvement for the synthesis approach, mainly to its performance, and offer flexibility for reusing various techniques and heuristics that affect ordering and dictate the reconstruction (even filter out many solutions).

Chapter 4

Code generation driven by specifications

This chapter presents ideas behind the code synthesis approach driven by specifications. By specifications we mean formal specifications - precise statement of properties that a program should exhibit (what the system should do, not necessarily how the system should do it) [10]. Besides formal specifications we also consider specifications of program behavior with input/output examples.

Instead of synthesizing only simple expressions that type-check at a given place in the code (as described in Chapter 2), this approach has more ambitious goal - it focuses on synthesizing whole functions that are correct according to certain correctness properties associated with them. This approach utilizes existing tools for code generation (e.g. a tool like InSynth that generates code driven by types), together with tools that check satisfiability (or validity) of program correctness properties. For that purpose, the approach relies on Leon, a framework that operates on a functional subset of Scala and offers a semi-decision procedure for checking satisfiability of expressive correctness properties of recursive first-order functional programs [71]. Within this approach, Leon is used for verification and evaluation of generated code.

The domain language of synthesis is again a subset of Scala language. Special construct, *hole*, that marks the body of a function to be synthesized has been made available to the developer. Specifications that drive the synthesis process are also written in the Scala language itself. Formal specification can be given as annotations[1] in terms of preconditions and postconditions, with *require* and *ensuring* constructs, respectively.[2]. Additionally, input/output example specification can be given with *passes*, a construct that takes a mapping of values and evaluates to true or false in

[1]not to be mistaken with general programming language annotations

[2]these two constructs are part of the Scala language standard library

the context of the given function. This approach is implemented as a plugin for the Scala compiler, which internally uses InSynth and Leon. It processes Scala source files and tries to synthesize correct functions that can be implemented as arbitrarily complex expressions that follow a certain control flow structure (*if* or *match*). We describe implementation of our system and from the evaluation of synthesis on several examples that represent widely-used algorithms and practical tasks, we conclude valuable insights into the potential of this approach.

4.1 Checking satisfiability of correctness properties using Leon

In this section we present Leon, a tool that is employed for checking satisfiability of correctness properties of programs (i.e. for checking validity of programs) and evaluation of programs according to given input/output examples. We will describe how Leon is implemented and organized and how can we benefit from it for the purpose of our synthesis approach. Additionally, we make a brief introduction to software verification, its motivation, applicability and limitations in order to achieve better understanding of how can, and in what extent, Leon be useful to our synthesis approach.

4.1.1 Background in formal verification

Due to the increase of complexity of modern designs, quality and reliability of software (and hardware) was becoming harder to achieve. In order to remedy this, researchers started to study formal verification techniques which have the goal of proving or disproving correctness of intended algorithms underlying a system, with respect to a certain (formal) specifications.

We will present some key techniques and results that lie at the foundations of many techniques used in verification of modern software. Understanding these can help understanding the principles and limitations of checking satisfiability of program correctness using Leon.

Formal specification

Formal specification is a mathematical description expressed with precise statements of properties that a program should exhibit [10]. Given such a specification, it is possible to use formal verification techniques to demonstrate that a candidate system design is correct with respect to the specification.

The formal specification had a great impact on the software engineering. A formal approach to software engineering that relies on formal specification during implementation, design by contracts, emerged [51]. It has its root in formal verification (and Hoare logic [32]). The idea is that a software component should have a very precise interface, expressed by its contract. Each contract can then be individually tested or automatically verified. In the case of purely functional programs, a component is comprised of a single function that is annotated with a contract. Our approach to synthesis is defined and mainly driven by specified contracts of individual functions.

Such contract has two main parts[3]:

Precondition a boolean expression composed of the function parameters

Postcondition a boolean expression composed of the function parameters and the function returned value

The goal of the verification process is then to check that if the precondition holds, then the postcondition also holds. More specifically, to get an answer if the result of executing the code for each possible input, that satisfies the precondition, satisfies the postcondition. This property alone is sufficient to prove that the function is correctly implemented.

More specifically, if we denote formulas for precondition and postcondition with P and Q respectively and compute the formula that expresses the function implementation F_c, then the verification needs to check validity of the following:

$$(P \wedge F_c) \to Q$$

Note that the resulting formula is quantifier free and the sets of free variables of P, Q, F_c need to be disjoint.This approach reduces a formally specified function to a finite set of formulas, called verification conditions, such that their validity implies the correctness properties of the function [10]. Traditionally, verification conditions are denoted with Hoare triples,

$$\{P\}\ F_c\ \{Q\}$$

We adopt the expression that if Hoare triple holds then the denoted verification is valid. More specifically, if $\{P\}\ F_c\ \{Q\}$ holds then the implementation reduced to F_c is indeed correct with respect to P and Q. We will extensively use verification condition in terms of Hoare triples in the subsequent sections.

[3]in case of imperative programs an additional part exists, an invariant, which defines unchanging correctness properties

One important remark is that this way, correctness is checked for a function individually and in order to guarantee correctness of the whole programs, all the invocation of functions must respect their preconditions. Function invocation which does not respect the precondition does not respect the contract and its result cannot be guaranteed to satisfy the postcondition.

Formal verification

Although many advances in the field of formal verification were made, the question of whether a program meets its specifications cannot be answered (with certainty) in the general case. Since the formal specifications of a program are written as a mathematical description, to answer the question whether a proram meets its specifications an appropriate mathematical proof needs to be constructed. Theorem provers are specialized software that automate mathematical proofs. Depending on the underlying logic, the problem can vary from easy to undecidable. This means that in some cases such question cannot be answered with certainty. Additionally, in some cases there exists only a semi-decision procedure for answering the question (an effective procedure that will always say 'yes' if the answer to the question is positive, or it will say 'no' or 'I do not know' otherwise [10]). One way to work around the undecidability is to limit the underlying logic the theorem prover reasons about to decidable fragments that are of interest. Perhaps the most studied class of provers is the SAT solvers, that only consider propositional logic [10]. SAT is one of the quintessential NP-complete problems.

An important class of solvers, that present important drivers of advances in verification of modern software and hardware, are the solvers that reason about Satisfiability Modulo Theories (SMT) which is a generalization of SAT to other theories [18,5]. These solvers reason about specialized theories specific to certain language properties (integers, data structures, ...) and thus can be more efficient in the domain of those theories. Leon uses an existing SMT solver and extends its supported theories. It reasons about theories supported by the solver with addition of recursive functions.

4.1.2 Leon

Leon is an automated system for verifying functional Scala programs and finding counterexamples to the validity of user-specified properties [72]. Leon uses existing semi-decision procedures for verification of purely functional programs. It builds upon an existing SMT solver (Z3 [18]) to provide a procedure for handling recursive function definitions. Thus, Leon can be used for all tasks where SMT solvers are used, including verification, synthesis, and test generation [71]. These programs are written in purely functional subset of Scala and their formal specifications can be expressed using existing Scala language constructs[4].

Leon can check satisfiability of expressive correctness properties of recursive first-order functional programs. Recall that recursive functions are expressive enough to give full Turing-completeness power to a programming language, without the need for loops [3]. Leon uses a procedure for checking the satisfiability of formulas modulo recursive functions. The procedure is based on successive unrolling of definitions of a recursive function which adds more information about the behaviour of the function. A top level loop alternates under-approximation and over-approximation of the formula, asking the underlying solver each time, until it converges to a solution (or it loops forever). In the phase in which the formula represents an under-approximation, all function invocations appearing in the formula are replaced with uninterpreted symbols before the formula is sent to the solver. Since the solver has the freedom to assign any meaning to such functions, we can check with certainty if the formula is unsatisfiable. Otherwise, the procedure proceeds to the over-approximation phase. In the phase in which the formula represents an over-approximation, it is made to force taking only branches that correspond to terminal cases of recursive functions (if terminal cases do not exist, the procedure may not terminate). If the solver says that the formula is satisfiable then we accept this answer with certainty. The requirement that each function terminates is very important and directly affects the correctness properties of Leon itself, thus our approach to synthesis must address the issue of synthesizing expression that could lead to non-terminating recursive calls.

An interesting remark is the technique of "lucky tests" that can speed up the verification process in Leon. If in the over-approximation phase solver returns that the formula is satisfiable, it could be that the solver guessed a valid assignment. Since the evaluation in Leon is fast (it amounts to executing the specification) this can be check and the satisfiable answer can sometimes be reported early. Leon was evaluated on verification of 60 functions from implementations of practical algorithms and data structures. The results show that Leon can be effective and efficient in verification and with average execution time below half a second represents an excellent fit for the verification tool in our synthesis approach.

Leon framework

Leon verification system has at its core an implementation of the procedure described in the previous section. Before invoking the procedure, Leon takes the input a pro-

[4]with transformations presented in [7], Leon can be used also for imperative programs

gram, written in a purely functional subset of Scala, and produces verification conditions for all specified postconditions, calls to functions with preconditions, and match expressions in the program [71]. Leon is written as a Scala compiler plugin which operates after the early stages of the compilation process, including parsing and type-checking. Immediate advantages of this approach directly benefit from these early phases and include parsing of Scala code and its constructs, type-checking and type inference.

Leon supports a purely functional subset of Scala which is Turing-complete. It supports core notions such as integers, booleans, arithmetic and comparison operations, maps, sets as well as case classes for expressing recursive datatypes together with pattern-matching expressions over such types [71]. Additional support in terms of lists, tuples, arrays and imperative programming construct has been described in [7]. A Scala program suitable as input to Leon is written as an object with collection of case classes and functions. Contracts can be expressed with *require* and *ensuring*[5]. After initial phases of the Scala compiler, Leon employs its own parsing step which produces abstract syntax trees specific to Leon. These abstract syntax trees effectively mirror the original Scala code but are tailored for efficient reasoning about the program within Leon. This includes verification (generation and analysis of verification conditions), counterexample generation, and evaluation[6].

Leon defines a lot of useful constructs and annotations for its operation in its extensible library. This library can be extended with additional constructs meant for interaction with the developer. We incorporate constructs needed for our synthesis approach into it.

The aim of the synthesis approach presented in this chapter is to utilize the current Leon infrastructure and integrate Leon into the synthesis process (together with the synthesis approach driven by types, described in Chapter 2) with a goal of reusing the available techniques, tools and framework for synthesizing complex programs according to their specifications.

4.2 Adapting code synthesis to code verification

Since InSynth supports synthesizing Scala code, the integration with Leon can be straightforward - InSynth would synthesize the code, this code would be inserted into appropriate places in Scala source files after which Leon could be invoked to process

[5] *invariant* construct is supported for annotating loops in the extension for imperative programs

[6]recently this was extended to support compilation of code to JVM and synthesis, effectively making an infrastructure around the Leon core

the sources. Although this approach could in principal work (and it was actually implemnted and tested), it cannot offer enough flexibility in invoking tools in the Leon framework. More importantly, since that way the Scala compiler needs to invoked each time we want to either synthesize code with InSynth or verify it with Leon and that incurs too much overhead.

In order to integrate the approach to code synthesis driven by types effectively, we needed to modify InSynth to conform to the Leon framework. This effectively means that the domain language of InSynth needed to be changed. Reasoning of the synthesis process needed to be changed from Scala code to Leon abstract syntax trees. By modifying InSynth and changing its domain language we also modified its expresiveness. We modified InSynth in such a way such that it can reason about the whole functional subset of Scala supported in Leon (i.e. that can be encoded by Leon abstract syntax trees) with the exception of control flow expressions. The aim is to reuse InSynth for generating only leaf and condition of control flow expressions that are sufficient for representing implementations of practical algorithms. The synthesis procedure itself would use these expressions to determine the structure and construct a correct control flow expression. Note that this approach is able to synthesize recursive code and thus retains Turing completeness of the code it can synthesize.

Instead of parsing directly Scala source files, InSynth is modified to parse Leon abstract syntax trees. It scans all functions and class definitions withing the Leon program and adds appropriate declarations to the initial succinct type environment. Additionally, regardless of the program given as input it adds all primitive operations and constants supported by Leon, such as arithmetic, comparison operators, boolean constants etc. As will be described in Section 4.3.1 the special *hole* construct which extends the Leon library determines the body of the function to be synthesized so that the function parameters are also added to the environment. Note that all fucntion declarations are visible to the synthesis process so the recursive calls can be syntheized without special reasoning about them. Instead of denoting a Scala declaration (from the given program) each declaration now represents a partial expression encoded in the Leon abstract syntax trees. Due to design flexibility of InSynth (as mentioned in Section 2.4) it was indeed enought to modify only the parsing phase and the last step of the reconstruction phase (the code generation) in order to fully adapt InSynth to the Leon framework.

Although InSynth is implemented as an interactive tool (an Eclipse IDE plugin, see Section 2.4.4), due to this integration it is modified to be used within the Scala plugin implementation of this synthesis approach.

4.3 Synthesis driven by specifications

In this section we will present the approach to synthesizing correct programs with respect to specifications. We will refer to previous sections and describe how does previously presented techniques and tools combine together within the synthesis process that is driven by specifications.

4.3.1 Motivation

In Section 2.6 we saw that the approach to code generation driven by types and weights (and its implementation in InSynth) can be very useful and effective in assisting developers in practice. The motivation behind InSynth is that the developer would benefit from having offered a list of code snippet suggestions while developing in a context which large number of API calls is exposed. As we observed, the usual scenario is that the developer knows which type of object he needs while he is unsure of the exact combination of API calls that are used to compose the desired, and relatively small, code snippet. Weights mined from a corpus of projects can significantly contribute to the quality of synthesized code snippets. As it can be concluded from Section 2.6, the more specialized corpus is, the more benefits weights can have to the quality of suggested code snippets.

InSynth aims at synthesizing short code snippets but not entire full-fledged functions that accomplish a complex task or an algorithm. But what happens if switched focus of InSynth and try to use it for synthesizing whole algorithms? What happens when the exposed API is relatively small but number of possible expressions to synthesize explodes due to their combinations (such example was introduced for the motivation for lazy enumeration approach described in Section 3.1.3)? Can we reuse (fast) synthesis driven by types and weights for more ambitious synthesis goals? After all, the synthesis approach focuses on synthesizing functional code, while even the most complex algorithms are expressed as single, self-contained expressions in pure functional languages.

Evaluation of correctness properties of code synthesized with InSynth

We will present an evaluation of InSynth in the context of synthesizing small examples that express simple and practical algorithms in a purely functional subset of Scala, the one used for writing programs for Leon (see Section 4.1). The results of this evaluation have given very useful insights and represent the main motivation behind the approach to code synthesis that employs both InSynth and Leon.

The experiments were done on multiple examples that have their programs written in Scala, together with their formal specifications (mainly just postconditions), such that their validity can be checked with Leon. In all experiments, we took valid programs with multiple functions (e.g. a code implementing list operations with objects that implement list algebraic data types together with methods that return size of the list, insert element, etc., as in the example given in Section 3.1.1), removed bodies of certain functions and marked them for synthesis. For all examples, prior to their modification for this experiment, Leon could verify all functions according to their formal specifications. At the marked places (holes), we invoked InSynth.

The synthesized code snippets were put back at the place of invocation while the rest of the code was left unchanged. This allowed us to automatically synthesize many code snippets and check their validity in the original program code. We implemented the approach that synthesizes a set of snippets for a single hole and plugs each snippet into the code that is otherwise intacted. The inherent limitation of this approach prevents us from testing InSynth in scenarios where multiple code snippets can be synthesized in different functions that are directly related for proving validity[7]. Interestingly, in some cases, synthesized snippets were different from the removed ones but still lead to a valid program.

The examples were taken and implemented from various collections of widely used and well known examples. Some of those include functional program examples found in [54], publicly submitted "tasks" found at [16] and existing Leon benchmarks [72].

Synthesizing whole function bodies

In the first batch of experiments, for certain functions, we removed the body of the function and replaced it with a hole, i.e. InSynth synthesized bodies of functions. Since most of these examples involve control flow statements in their code (*if* and *match*) and InSynth does not explicitly reason about *if* statements (it can synthesize only conditions and branches of an *if*), we needed to include them explicitly.

One limitation of InSynth is that it cannot synthesize *match* statements - it cannot "refine" declarations to appropriate subtypes based on case patterns (nor can it extract fields). These statements can be expressed with *if* statements while preserving the semantics of pattern-matching [71]. If that is done, the context can be enriched inside the case statements (e.g. matching a standard abstract data type representation of *List*, as presented in Section 3.1.1, incurs that in one case statement the matched expression is Nil, while in the other it is $Cons$, and additional *head* and *tail*

[7]besides such an approach being cumbersome, an attempt to perform it was made, and lead to bad (and expected) results

fields can be visible).

The experiments did not produce favorable results for using this approach in practice. Even when the initial environment was extended to encode the "refining" of declarations, InSynth was able to synthesize valid code in just a very few examples. InSynth was able to synthesize bodies of few very simple, trivial non-recursive functions (e.g. checking if a list is empty) and very few recursive ones, when the "refinement" was done explicitly (e.g. returning size of the list). In all cases the rank of valid code snippets was very high (even three orders of magnitudes higher than results presented for synthesis of individual holes, as will be shown later) thus making such synthesis approach practically unusable.

Synthesizing "holes" within function bodies

In the second batch of experiments we evaluated InSynth in similar circumstances, but here, instead of synthesizing whole function bodies, we removed certain (simple) expressions such that when removed, the "structure" of code remains unchanged. We synthesized branches (and conditions) of *if* and case statements of *match* statements. In case of a *match* we removed case statements and introduced additional boolean conditions that would be needed if we were to do the conversion from that *match* to *if* (express that *match* with an appropriate *if* statement). The following example shows the experiment setup in the case of code for the list concatenation:

```
def concat(l1: List, l2: List) : List = ({
  (l1, l2) match {
    case (Nil(), _) => l2
    case (_, Nil()) => l1
    case (Cons(l1Head, l1Tail),
        Cons(l2Head, l2Tail)) =>
      Cons(l1Head, concat(l1Tail, l2))
  }
}) ensuring(res => content(res) ==
     content(l1) ++ content(l2))
```

Listing 4.1: Valid code for concatenation of two lists

```
def concat(l1: List, l2: List) : List = ({
val cond1: Boolean = ▮
l1 match {
  case Nil() => ▮
  case Cons(l1Head, l1Tail) =>
    val cond2: Boolean = ▮
    l2 match {
      case Nil() => ▮
      case Cons(l2Head, l2Tail) => ▮
    }
}) ensuring(res => content(res) ==
     content(l1) ++ content(l2))
```

Listing 4.2: Transformed code with "hole" expressions removed

The example illustrates that instead of matching a expression with a *match*, equivalent code can be written as *if* with an appropriate condition[8]. Note that this approach does not suffer from the limitation of "refining" mentioned in the previous section since the case patterns remain and the fields of matched expressions are introduced into the environment. The goal is not to solve the "refining" issue but rather to see if valid expressions can be synthesized under the assumption that the "refining" issue is not a concern (it presents one of the challenges that need to be solved when using this technique, as it is described later, in Section 4.3.2).

We evaluated this approach on the set of multiple examples. We strove to remove expressions inside *match* and *if* statements of functions that implement the core functionality. InSynth was able to synthesize the desired code snippet in more than 90% of cases. The parameters to the synthesis process were as follows: 1 second time limit and 500 code snippets to synthesize. The synthesis invocations in which the desired code snippet was not found were mainly due to the lack of reasoning about arithmetic operations and certain integer constants (e.g. the division operator / and constant -1). Explicitly introducing such operators and constants improved the results significantly. Table 4.1 shows results for examples of 5 chosen algorithms that implement both well-known and simple algorithms, and that reflect the overall experiment results.

Table 4.1 represents results of synthesizing holes in the structure of 6 representable functions. We can see that InSynth managed to synthesize the desired code snippet in almost all cases without explicit assistance, and in all cases in which the context of synthesis was altered to aid the synthesis. Since InSynth cannot reason about arithmetic operations and all constants, some code snippets could not be synthesized regardless of the parameters that drive the synthesis. However, for all those examples it was possible to modify the context of synthesis (by adding necessary declarations with appropriate weights) to make the desired snippet found with a rank lover than 50 (although by doing so, the quality of returned snippets could drop in the general case).

This shows that application of the synthesis approach driven by types and weights looks promising, especially if the context can be modified (specialized) for the synthesis task at hand. Note that no corpus was used to affect the weights in these examples.[9] For some examples, such as the *sort* function of the insertion sort algorithm, all the needed snippets for holes (in isolation) are found with rank 15 or less (less than 6 on average), while for some more complicated ones, such as the *insert*

[8] *if* expressions can be more general than matching thus approach to synthesizing *if* expressions can itself synthesize more general algorithms

[9] this offers room for potential improvement for this synthesis approach

Test	**Method**	**Rank**
Insertion Sort	*sortedInsert*	51.8
	isEmpty(list)	0
	Cons(e, Nil)	13
	*X <= e	N/A (47)
	Cons(x,sortedIns(e, xs))	161
	Cons(e, l)	38
	sort	5.67
	isEmpty(list)	0
	Nil	2
	sortedIns(x, sort(xs))	15
List search	*linearSearch*	15.8
	isEmpty(list)	0
	**-1	N/A(38)
	*LHead = c	N/A (26)
	linearSearch(lTail, c)	10
	size(l)	5
Merge sort	*sort*	85.33
	isEmpty(list)	0
	split(list,length(list))	3
	merge(mergeSort(p.fst), mergeSort(p.snd))	253
List concatenation	*concat*	22.2
	isEmpty(l1)	0
	l1	1
	isEmpty(l2)	1
	l1	0
	***Cons(lHead, concat(lTail, l2))	109
Red black tree	*insert*	239.25
	isEmpty(tree)	1
	balance(c, ins(x, a), y, b)	405
	Node(c,a,y,b)	144
	balance(c,a,y,ins(x, b))	407
Average		**70.01**

Table 4.1: Results of measuring effectiveness of InSynth in terms of synthesizing correct code. The first column denotes the description of the algorithm used in that example. The second column gives name of the function in which the code is synthesized and the desired code snippet. The third column denotes the ordinal of the desired snippet or average over ordinals for corresponding set of examples. The ordinal in parentheses represents the result after we modified the context.
* denotes examples in which an additional declaration had to be introduced (>=, =)
** denotes the example in which a general constant for that type was found (by default 0 in case of Int)
*** example in which a code snippet different than the removed one was found that satisfies the correctness properties

function for the red black tree the rank upper bound is 407 (and the average is less than 240).

For all these examples, we removed all simple expressions that alone do not comprise a control flow statement (in our examples most of them are *if* and *match* statements) but are sufficient for construction a correct function if filled in at right places. Therefore, if we knew the control flow structure of a function (or we had a way for finding it out), together with places where synthesized expressions need to be put, we would be able, in principle, to synthesize entire functions for all these examples.

As we saw in the previous section, the approach where whole functions are synthesized does not give good results. By intuition, if we think about control flow expressions, the results match the expected. If we consider the simple *if* term, it is comprised by a combination of three expressions (with appropriate types, one of them being boolean). If we assume a simple procedure that enumerates sub-expressions and combines them to make a larger expression, and denote the expected rank of the desired expression in the case of synthesis of all three expressions in *if* with r_1, r_2, r_3, without loss of generality, we can conclude that the expected rank of the desired *if* expression is $O(max(r_1, r_2, r_3)^3)$. This can lead to a logical explanation of bad results and the practical limitations of this approach.

Since these results witness that the synthesis driven by types can synthesize sub-expressions ("hole" expressions) within a control flow expression efficiently, an idea for a technique that progressively synthesizes one sub-expression at a time in order to construct a more complex one with control flow, naturally emerges. If we consider the level of performance of the type inhabitation problem in succinct calculus, presented in Section 2.6, in which proof trees that encode thousands of expressions are found in less than 50ms, we can get an idea of an approach to use this synthesis for a quick and gradual "construction" of complex expressions.

4.3.2 Synthesize and verify approach

Up to this point in this document, we presented tools and techniques that can:

- synthesize code (InSynth, driven by types and weights, Chapter 2)
- enumerate synthesized expressions (lazy enumeration, Section 3.1.3)
- verify correctness properties of code (Leon, Section 4.1)

Additionally, we presented a set of examples that represent implementations of various commonly used algorithms that demonstrated a motivation for an approach to synthesize valid programs by constructing complex control-flow expressions from more simple ones (Section 4.3.1). These examples are written in a functional subset of Scala that lies at the intersection of the ones used by InSynth and Leon, and for which appropriate modifications were made in order to enable all three mentioned techniques to be combined and used for a common domain language (Section 4.2).

Having all this in mind a straightforward idea that comes up is to use the widely-familiar "generate and test" approach. This method, sometimes called "trial and error" or "guess and check", has been successfully applied in various areas of science including computer science (it lies in the basis of many techniques, including ones from the field of artificial intelligence [64]).

Generate-and-test search algorithm is a very simple algorithm and boils down to:

1. generating a candidate for a solution

2. test to see if this is the expected solution
3. if the solution has not been found, repeat from the first step

In the scenarios where more than one solution can be found, instead of quiting when finding the solution, we can proceed to find other solutions. Generate-and-test algorithm is guaranteed to find a solution if done systematically and there exists a solution.

This approach seems as a straightforward solution whenever we have means to generating a candidate and testing it. Interestingly, the previous work on the synthesis driven by types, using quantitative type inhabitation [30,31], describes an implementation in which the developer can specify correctness properties (with Scala *assert* and *ensuring*) and provide test cases which serve to filter out synthesized code which is not valid[10]. In our experiments done in Section 4.3.1, we used InSynth to synthesize code snippets which were then inserted into programs and verified with Leon. Note that both of these examples effectively use the "generate and test" technique to some extent.

For the purpose of synthesis of correct programs, the aim is to do the generation part (synthesis) with InSynth and the testing part (verification) with Leon- thus, on some occasions we refer to this approach as, "synthesize and verify".

From the experiments presented in Section 4.3.1, we concluded that the synthesis driven by types, by itself, is not sufficient for effective and efficient synthesis of complex expressions. Although we presented a systematic way for enumerating synthesized terms, and theoretically, for each problem out there, regardless of the complexity of its solution, the desired expression can be enumerated at some point, this "brute force" approach is far from being practical. Moreover, results have shown that synthesizing individual, simple expressions that make a correct program when inserted at certain points ("holes"), can be done effectively and due to presented performances of used tools, also efficiently.

The idea of abducing conditions

The main idea that allows the synthesis approach to combine mentioned techniques comes from the area of abductive reasoning [37]. Abductive reasoning, sometimes also called "inference to the best explanation", is a method of reasoning in which one chooses a hypothesis that would, if true, explain the found evidence in the best way. In the context of logical reasoning, it is analogous to the inductive reasoning

[10]after the search algorithm returns candidate snippets, they are inserted one by one in the user code and tests are run - if at least one test fails, it is discarded [30]

(inferring consequence b from some argument a, where b does not necessarily follow from a) applied in the opposite way (inferring, or "abducing", a as an explanation of consequence b). The type of reasoning was studied and applied in various areas of science including philosophy and computer science. Abductive logic programming exists as an extension of the logic programming paradigm and allows specifying some predicates in an incomplete manner and requiring the computation system also to explain certain observations [38].

Since we are able to effectively synthesize valid "hole" expressions that belong to the appropriate control flow structure, they can be viewed as parts of the whole algorithm that are executed only in certain cases. These cases are unambigously defined by the conditions and case patterns in appropriate *if* and *match* terms. If we could apply the idea of abductive reasoning to allow guessing ("abduce") the appropriate condition (the explanation for our code), we could synthesize a program which behaves correctly in certain cases. Such program could effectively encode certain correct branches (or equivalent match case expressions) of the complete program. More specifically, if we have the goal to synthesize a function with a given postcondition, we can introduce an appropriate condition as the precondition of the function, and afterwards verify the resulting function for correctness (as described in Section 4.1). If verified, such function satisfies its postcondition only in cases encoded by the introduced precondition - the function effectively encodes the behavior of the correct function in those cases. Progressively applying this technique of abducing individual preconditions can result in incremental construction of a program which behaves correctly in more and more cases, eventually constructing the correct program.

Constructing partial solutions We will give an illustrative example of how can abducing appropriate condition lead to the synthesis of programs that behave correctly in certain cases.

Let us consider the greatly exploited example that presents an implementation of concatenation of lists, presented in Listing 3.3. As we can see, the postcondition of the *concat* function is `res => content(res)== content(l1)++ content(l2)`, while the precondition is omitted (it is just **true**). Let us assume that we have a source that can generate expressions of type *List* and *boolean*.

Having the goal to synthesize the correct body of *concat*, if we try to plug expression `l1` as the body, the function would of course not be correct. But we may observe that `l1` actually represents the correct result in the cases where `l2` is equal to `Nil()`.

More specifically, the following function can easily be verified:

```
def concat(l1: List, l2: List) : List = ({
  require(l2 == Nil())
  l1
}) ensuring(res => content(res) == content(l1) ++ content(l2))
```

This effectively means that the expression l1 behaves correctly in the cases which satisfy the precondition l2 == Nil(). Thus, the expression covers a certain partition of the space of inputs and may represents a valid branch in the complete program (as in this case, l1 corresponds to a case expression). From the formal verification point of view, this expression captures a subset of basic paths in a valid program [10].

Note that after verifying such function we can proceed to cover more cases by solving the same problem in which the goal is not to synthesize the body of the function, but rather an expression that produces a correct function when inserted at the place of hole as in the following code:

```
def concat(l1: List, l2: List) : List = ({
  if (l2 == Nil()) l1
  else hole
}) ensuring(res => content(res) == content(l1) ++ content(l2))
```

We can think of the body b in this case as a partial expression that represents correct behavior of the function in some cases and can be given an expression e so that it constructs a correct function body, $b(e)$. More specifically, if we denote abduced conditions as $p_1, p_2, \ldots, p_n$ and the body and postcondition of the function with r and q, then the following Hoare triple holds $\{p_1 \lor p_2 \lor \ldots \lor p_n\}\ b(e')\ \{q\}$, regardless of the expression e' put instead of hole. Furthermore, if a correct function exists then exists an expression c such that $\{true\}\ b(c)\ \{q\}$ holds (it must exists since plugging the body of such correct function must produce another correct function).

4.3.3 The synthesis algorithm

In this section we will describe a general version of the synthesis algorithm that is given a collection of expressions and knows how to verify the program but does not explicitly use any particular tools[11]. The algorithm directly follows from the "synthesize and verify" technique described in the previous section. It is presented in Algorithm 15.

The algorithm applies the idea of abducing conditions to progressively synthesize and verify branches of a correct function. The input to the algorithm is a function f with a precondition p and a postcondition q, and a collection of expressions s. For the sake of brevity, the algorithm gets a single collection of expressions which

[11]the algorithm can easily be changed to work with a concrete code synthesizer

Algorithm 15 Synthesize a correct program

```
Require: function $f$ with a precondition $p$ and postcondition $q$, a collection of expressions $s$
 1: $sol = (\lambda x.x)$ {maintain a partial solution}
 2: repeat
 3:    get an expression $b$ from $s$
 4:    assign $e$ as the body of $f$
 5:    if program is correct then
 6:       assign $p$ and $(sol\ b)$ as the precondition and body of $f$
 7:       return $f$ {a correct program is synthesized}
 8:    else
 9:       try to synthesize a branch {calls Algorithm 16 with $f, s$}
10:       if a branch with condition $c$ is synthesized then
11:          update $sol$ to $(\lambda x.\ (sol\ (\text{if } c \text{ then } b \text{ else } x)))$
12:          assign $p \wedge \neg c$ as a precondition of $f$
13:       end if
14:    end if
15: until $s$ is not empty
```

contains all necessary expressions that are tested at the place of both condition and a branch expression (as described in the previous section). The formal description of the implementation will follow so that the algorithm here can be presented as simple as possible.

The initial precondition of f in the algorithm is p and it gets refined by adding clauses by a conjunction (line 12). Let p' denote conjucted clauses at any given time in the algorithm, or more specifically, let the precondition at any given time be $p \wedge c_1 \wedge \ldots c_n = p \wedge p'$. The algorithm maintains the partial expression (solution) in the variable *sol* (line 1). *sol* is encoded as a function in λ-calculus and when applied to a term t such that $\{p \wedge \neg p'\}\ t\ \{q\}$ holds, then $\{p\}\ (sol\ t)\ \{q\}$ also holds. This means that *sol* effectively encodes a partial solution of all synthesized branches in the algorithm at any given time. If *sol* is a applied to a term that represents a solution for all cases not covered by the partial solution, the resulting term is an expression that represents a correct function body. Note that *sol* encodes the correct program for all cases which satisfy $p \wedge \neg p'$, while the precondition of f is $p \wedge p'$ (the precondition defines a partition of the space of inputs).

The algorithm repeats enumerating all possible expressions b from the given collection (line 3). For each b it plugs b as the body of the function, f (line 4) and checks the correctness of the resulting function, i.e. $\{p \wedge p'\}\ b\ \{q\}$ (lines 5-14). If the resulting function is verified, then, from the previous discussion, we know that we found a needed part of the solution and that $(sol\ b)$ represents a correct function body (i.e. $\{p\}$ $(sol\ b)$ $\{q\}$ holds). The algorithm constructs the function and stops (lines 6-7). Otherwise, the algorithm tries to find an expression that represents the needed precondition c in order for b to be a correct body, i.e. for $\{c\}$ b $\{q\}$ to hold (it tries to abduce the precondition that could lead to correctness of the given body) (lines 9-13). In effect it synthesizes a branch of an *if* expression that corresponds to valid behavior of the whole function in certain cases.

The branch guessing is done as described in Algorithm 16. If the branch is found, the partial solution is updated to include the additional branch (lines 11-12). *sol* now represents a function such that when applied to a term, produces a function that is correct for all inputs that satisfy c together with preconditions of previously found branches. The current precondition is updated with the negation of c so that the solution in the next iteration covers cases where c does not hold. The algorithm repeats (at line 15) and it will eventually, given the appropriate terms in the collection s, find an expression b that forms complete body of a correct function.

Algorithm 16 tries to guess a branch of the desired function.

Algorithm 16 Synthesize a correct branch

```
Require: function $f$ with a body $b$, precondition $p$ and postcondition $q$, a collection
    of expressions $s$
1: for all expressions $c$ from $s$ do
2:   assign $p \wedge c$ as the precondition of $f$
3:   if program is correct then
4:     return $c$ {return the "abduced" precondition}
5:   end if
6: end for
```

The algorithm enumerates expressions from s and searches for a valid condition expression - that is, an expression c such that the Hoare triple $\{p \wedge c\}$ b $\{q\}$ holds. If the function is called as explained for Algorithm 15, this effectively means that a branch of an expression that represents a correct function body, with respect to initial given precondition and postcondition, is found and its condition extends the space of inputs covered by the partial solution. This condition c is returned as a result.

Note that both algorithms return the first found solution. Instead of returning the first solution, algorithms could collect and return multiple solutions (this may be useful if such solutions can be compared and ranked).

4.3.4 The synthesis problem formalized

The rough definition of the synthesis problem is: in the context of a given program, given a formal specification of a function in terms of its postcondition (and optional precondition[12]), return an expression that satisfies it. More specifically, if we denote the given precondition and postcondition with p and q, the returned expression r has to satisfy $\{p\}\ r\ \{q\}$. Setting the context of a given program means that formal specification and the returned expression (which represents the body of the function) can use all appropriate declarations given in the program (types, functions).

Up to this point we did not address in detail the fact the synthesis can be specified with input/output examples. Effectively, with input/output examples, the specification of the function to synthesize can be extended. As mentioned in Section 4.3.2, evaluation of the synthesized code on the specified input/output examples can filter out invalid solutions thus replace or strengthen the formal specification. As we will present in Section 4.3.5, specified input/output examples (or even input examples alone) can also provide means for improving the performance of the synthesis process.

The following definition of the synthesis problem includes the (optional) specification with input/output examples.

Definition 4.3.1 (The problem of synthesizing a correct program) *In the context of a given program, given formal specifications of a function f, in terms of its postcondition q, optional precondition p and optional input/output example pairs IO, return an expression r such that $\{p\}\ r\ \{q\}$ holds and outputs of evaluating r on example inputs from IO match the according example outputs from IO.*

The previous definition has been made general so that it does not describe any particularities of the approach that solves the defined problem. It formalizes the general requirements of the problem we want to solve. Since our approach to synthesis focuses on reusing existing techniques and tools for synthesis and verification of code, the problem we want to solve is more specific and includes the availability of existing techniques and tools for generating (synthesizing) expressions, verifying them according to formal specifications and testing against specified input/output examples. Such specific context of the problem allows using the algorithms defined in Section 4.3.3.

4.3.5 Implementation

Although the generality of used algorithms allows using different techniques and tools for code generation, verification and testing, the techniques and tools presented so far fit nicely into the requirements. These are mentioned in Section 4.3.2 and include two main tools that are used in this approach: InSynth and Leon[13].

[12]if the precondition is omited we can consider it to be *true*

[13]the lazy stream enumeration technique emerged as a necessity for this approach due to inherent limitations of the eager enumeration

With the intention to use InSynth and Leon we needed to adopt a common framework and a domain language for these tools. As described in Section 4.2 and 4.3.1 InSynth was modified to conform to the internal abstract syntax tree representation of Leon. This allows the synthesis approach to work with a functional subset of Scala that lies at the intersection of the ones used in InSynth and Leon. InSynth still solves the type inhabitation problem in the *succinct types* calculus but now reconstructs terms in the form of abstract syntax trees in Leon. The synthesis process is implemented as a Scala compiler plugin that adds a separate phase to the compilation process. The synthesis phase occurs after the initial phases of the compilation process, similarly as in Leon, and invokes both InSynth for synthesis and Leon for verification and testing, repeatedly during the synthesis process.

The developer can write his programs in the Scala programming language and specify formal specifications the same as it is described in Section 4.1. This inherently implies all benefits and limitations of Scala programs imposed by Leon.

Introducing of additional constructs for the synthesis

Additional language constructs needed to be defined in the Leon library for the purpose of specifying synthesis problems.

Hole The notion of a "hole" resembles the cursor that specifies the program point of synthesis in InSynth (see Section 2.4.4). A *hole* construct is written in place of a body of a function which we want to synthesize, that is, it specifies the function which we want to synthesize. This function needs to be correctly defined in terms of its signature and the contract (precondition can be omitted). The return type of the function which we want to synthesize needs to be known. If the return type is not specified in the signature, then *hole* needs to be parametrized by the desired return type of the function (either explicitly or with a parameter[14]).

Note that *hole* designates the function for which we want to synthesize a correct body and thus denotes the place where an arbitrarily complex expressions can be put and thus differs from "hole" expressions mentioned in Section 4.3.1.

Specification in terms of input/output examples Since besides giving the formal specification with precondition and postcondition we allowed specifying the function with input/output examples, we introduced a construct that allows the developer to do this. For each synthesis problem, an additional construct *passes* is

[14]in which case the desired type can be infered by the Scala compiler

available which takes three parameters: an input/output example map and two variables binding inputs and outputs during the evaluation of tests, respectively. The input/output example map maps from types A to B where A corresponds to the tuple defined by the function arguments, while B is the return type of the function. Each mapping defined in this map, map an input example to corresponding output example that will be checked during test evaluation. The binding variables specify expressions to be evaluated, and determine input values and reference output values in the evaluation. This allows flexibility in controlling the testing process (e.g. it allows specifying properties that should hold for certain fields of the result value). If invalid parameters are passed to this construct, i.e. these three parameters do not have appropriate types, the type-check phase of the compilation process will fail.

An example of a program in which a function is marked to be synthesized and provided with formal specification and input/output examples for the synthesis process, is given below:

```
object ListOperations {
  val mapping = Map[List, List](
    (Nil(), Nil()) −> Nil(),
    (Cons(0, Nil()), Cons(1, Nil())) −> Cons(0, Cons(1, Nil()))
  )

  def concat(l1: List, l2: List) = ({
    hole[List]
  }) ensuring(res => content(res) == content(l1) ++ content(l2) &&
  size(res) == size(l1) + size(l2) && passes(mapping, (l1, l2), res))
}
```

Listing 4.3: An example of code that sets up the synthesis

The example shows the definition of the function *concat* which should concatenate two lists[15]. The main *ListOperations* object represents the highest level of encapsulation currently supported (and required) by Leon. We omit the import statement necessary for the Leon constructs, declarations of types (*List*, *Cons* and *Nil*) and definitions of *size* and *content* functions that are defined within the same object for brevity. *hole* marks the function body to be synthesized and declares the returning type of the function. A precondition is omitted while the *ensuring* construct specifies the postcondition, $content(res) = content(l1) \cup content(l2) \wedge size(res) = size(l1) + size(l2)$. The behaviour of the function is also specified with input/output examples. *passes*

[15]with respect to usual and natural implementation of *size* and *content* on lists

construct specifies that keys from *mapping* serve as inputs and values from *mapping* serve as reference outputs for when evaluating the behavior of the synthesized function. An interesting remark is that the input/output example map actually makes the specification of the behavior of the function stronger, since its second input/output example pair restricts concatenation of two lists of size 1 to have a particular order of elements.

Algorithms that use InSynth and Leon, that are used in the actual implementation of the synthesis approach, are very similar and based on algorithms given in Algorithm 15 and 16, but different in a couple of subtle details due to issues inherent of their practical realization. Two things need to be taken care of in the implementation of these algorithms:

- dealing with infinite number of reconstructed terms: Since most of the motivating examples implement algorithms with recursive functions, the number of synthesized terms that have the function's return type are infinite. When using the reconstruction with lazy enumeration, they can be enumerated from a stream that does not have a finite size[16]. Using the Algorithm 15 and 16 without modifications, could thus lead to non-terminating executions.

- dealing with refinement of types: As mentioned in Section 4.3.1, without refining types of declarations in appropriate branches in the code, the desired terms cannot be synthesized. In the implementation, this is solved by explicitly tracking possible types for each declaration in every branch during the synthesis process.

  After a branch is synthesized, in some cases, refining can be done automatically (e.g. in the else branch of, *if (list == Nil()) then* { ...} *else* { ...}, we can be sure that list has the actual type of *Cons*[17]) If the synthesis approach cannot progress with current declarations in the scope, types of certain declarations are refined and the process is repeated.

The presentation of these algorithms is deferred to the following section, which also includes integration of the technique for filtering with input/output examples, together with other optimizations and subtleties.

[16]this actually presents the main motivation behind lazy enumeration of reconstructed terms, as described in Section 3.1.1

[17]with respect to usually adopted implementation of the list abstract data type

Filtering with input/output examples

This section describes the implementation of the synthesis approach, and presents various techniques and heuristics that are applied in order to improve its performance. Motivations behind these optimizations can be gathered from analyzing the evaluation results in Section 4.3.7.

The technique that promises the most benefits to performance of the synthesis process is filtering (or ranking) of expressions with the set of input/output examples. The rationale behind filtering with input/output examples has the assumption that executing an expression on a given input, and evaluating the result, can be faster than the verification of the expression, in the context of a given function. The goal is to use such examples to quickly test expressions before forwarding them for verification. Such early evaluation of expressions based on examples can determine which expressions are unlikely to be the correct ones and avoid their verification. The idea was motivated by good results that resulted from filtering expressions when finding an appropriate branch expression, as given in Algorithm 15. In cases where complex branch expressions needed to be synthesized, aignificant speedups were achieved by filtering out many expressions quickly - expressions that otherwise took a lot of time for verification.

Implemented algorithms

We now present algorithms that lie at the core of the synthesis approach and closely resemble its implementation. They solve the synthesis problem as formalized in Section 4.3.4 and employ InSynth for synthesis of expressions and Leon for checking satisfiability of correctness properties and evaluation of tests. They resemble algorithms given in Algorithm 15 and 16, but had to be modified to overcome practical limitations and allow using input/output examples.

Algorithm 17 takes a function f with assigned precondition and postcondition, a set of input/output examples io and the desired type T of the function which we want to synthesize. The main difference with respect to Algorithm 15 is that in this case the algorithm maintains a priority queue of expressions, that are ordered based on the number of input/output example pairs that evaluated correctly. More specifically, an example input evaluates correctly if it satisfies p, while the evaluation result satisfies q and is equal to the appropriate pair example output[18]. This effectively means that in each iteration of *repeat* at line 4, a finite number of expressions is evaluated from

[18] outputs can be omitted and in that case inputs serve solely for evaluation in order to improve performance, but not for additional specification

Algorithm 17 Synthesize a correct program, implementation

```
Require: function $f$ with a precondition $p$ and postcondition $q$, set of input/output
    examples $io$, desired type $T$
 1: initialize an empty priority queue $q$
 2: initialize declaration refinement mapping
 3: $sol = (\lambda x.x)$
 4: repeat
 5:    invoke InSynth with desired type $T$ and to stream of expressions $s$
 6:    if $q$ is empty then
 7:       for $i \leftarrow 1$ to $n$ do
 8:          let $e$ be the next expression removed from $s$
 9:          evaluate the expression on the set of examples from $io$
10:          enqueue $e$ in $q$ with number of passed examples
11:       end for
12:    end if
13:    dequeue an expression $b$ from $q$ and assign it as the body of $f$
14:    check satisfiability of correctness properties of $f$ with Leon
15:    if  program is correct  then
16:       restore initial precondition $p$, and assign $sol$, $(sol\ b)$ as the body of $f$
17:       return  $f$ {a correct program is synthesized}
18:    else
19:       try to synthesize a branch {calls Algorithm 18 with $f$}
20:       if a branch with condition $c$ is synthesized then
21:          update $sol$ to $(\lambda x.\ (sol\ (\text{if } c \text{ then } b \text{ else } x)))$
22:          assign $p \wedge \neg c$ as a precondition of $f$
23:          refine types of declarations and empty $q$
24:       end if
25:    end if
26: until $s$ is not empty or timeout
```

the stream and ranked according to evaluation results. The rest of the algorithm is familiar from Algorithm 15 - expressions are now dequeued from q and assigned as the body of f. The function is checked for satisfiability of correctness properties (line 15). If the check passes, the expression is used to form the full synthesized function and the function is returned.

Otherwise, we did not synthesize a correct function, and the algorithm for synthesizing a branch is called (line 19). If the algorithm synthesizes a branch successfully, the queue is emptied and postcondition of f is updated so that synthesizing appropriate expression can continue in the next iteration. Additionally, variable refinements are maintained and the declaration refinement is done each time a branch is synthesized and its condition can enable variable refinement. If the algorithm cannot synthesize a branch with expression b, the expression is discarded.

Algorithm 18 Synthesize a correct branch, implementation

```
Require: function $f$ with a body $b$, precondition $p$ and postcondition $q$
 1: initialize an empty set of models $cm$
 2: invoke InSynth with desired type of boolean and to stream of expressions $s$
 3: for all finite number of expressions $c$ streamed from $s$ do
 4:   if $c$ prevents all models in $cm$ then
 5:     assign $p \wedge c$ as the precondition of $f$
 6:     check satisfiability of correctness properties of $f$ with Leon
 7:     if program is correct then
 8:       return $c$ {return the “abduced” precondition}
 9:     else
10:       add new counterexample model to $cm$
11:     end if
12:   end if
13: end for
```

The algorithm for synthesizing branches is given in Algorithm 18. The main addition to this algorithm with respect to the one given in Algorithm 16 is the filtering of condition examples based on counterexample models derived when checking satisfiability with Leon. After each check with Leon, if it was unsuccessful a counterexample model that witnesses the unsatisfiability is retrieved and added to to set of counterexample models cm. A condition expression c is considered only in the case such that, if c is satisfied then all models from cm cannot be satisfied (i.e. $\forall m \in cm.\ c \rightarrow \neg m$). This is somewhat similar to the counterexample guided iterative refinement approach (as described in Chapter 5) but used for the purpose of filtering out already synthesized expressions.

Due to practical reasons, similarly as in Algorithm 17, the algorithm enumerates only a certain finite number of expressions n, and if none of them can make a correct branch, the search stops. This imposes an inherent limitation to the process - if a correct boolean condition can be enumerated, but with a rank higher than n, our process will never check it and will give up. This issues can be remedied by organizing the search similar to the breadth-first search[19].

Note that input/output example pairs are often used way for specifying behavior of the program. Although such specification is naturally given in input/output example pairs, our approach can benefit solely from given input examples, as the evaluation of input examples can help ranking of expressions according to the satisfaction of the postcondition. The more basic paths in the program they exercise, the more useful

[19]this is considered as future work

the input examples are. It will shown in the Section 4.3.7 that giving right input examples, can significantly improve performance of synthesis.

Note that further extension to these algorithms are possible. The algorithm could be modified to return found partial solutions in the case of exceeding the imposed time limit. The approach progressively constructs solutions to the synthesis problems and in each iteration maintains the invariant that the partial solution is correct when certain precondition is met, as discussed in Section 4.3.2 (initially, such precondition is $false$ and the solution does not cover any input to the function).

4.3.6 Correctness properties of the synthesis

In this section we define correctness properties of the approach to synthesis driven by specifications and show that those properties hold. An important property that naturally emerges in order for this kind of synthesis to be sensible, is that code that does not satisfy given specifications should never be synthesized. This means that the developer should never get a program that behaves badly since that clearly defeats the purpose of the problem that we want to solve and can bring bad and unpredictable consequences. It is imperative that we guarantee that all synthesized code will be correct with respect to given specifications and allow the synthesis process to terminate without returning a solution (this is similar to the definition of the semi-decision procedure [10] and represents a common behaviour of various algorithms used in practice). Moreover, since the approach can be modified to return partial solutions working for only certain cases, we should prove that if a partial solution is returned, it behaves correctly according to given specifications.

The additional important property that can be stated for this synthesis approach, is that the approach should be able to synthesize a correct solution if one exists. This in general represent a property that can be hardly satisfied in practice. In our approach we will restrict this property to reason about the underlying tools used which may not be sufficient to find a solution even if one exists.

The two mentioned statements resemble semantics of two classes of properties, frequently used in mathematics and computer science - soundness and completeness [10] (these are also used in Section 2.5). We will refer to the mentioned correctness properties as the soundness and completeness property, respectively.

Soundness and completeness

Note that our synthesis approach directly relies on correctness of tools that are used for generation, verification and evaluation of code. In order to be able to guarantee correctness, the correctness of underlying tools must also be guaranteed. Similar statement holds for the domain of operation and expressiveness of the underlying tools. It is reasonable to expect that the technique used for this approach can synthesize correct code only if it applies correct underlying tools and the the expressiveness of those tools is sufficient for the given problem.

In our case, the used algorithms imply that all programs that can be synthesized, can be represented as control flow expressions in which leaf and condition expressions themselves can be synthesized by the underlying synthesizer. Similarly, all programs that can be synthesized according to used algorithms can be verified for correctness and evaluated against input examples. The correctness properties need to take into account these remarks.

Theorem 4.3.2 (Soundness) *If the synthesis returns a function f as a result, then the function f must satisfy given formal specification and input/output examples, with respect to the verification tool used.*
More specifically, $\{p\}$ r $\{q\}$ has to be verifiable with the underlying verifier, where b is the body of f and p, q are the precondition and postcondition defined by the formal specification of f, respectively. In addition, if a set of input/output example pairs is defined, all given inputs must evaluate to appropriate outputs and the evaluation result must satisfy q, with respect to the underlying evaluator.

Theorem 4.3.3 (Completeness) *If there exists a correct program to the given synthesis problem that represents an expression encodable with a control flow term, and all leaf and condition expressions of this term can be synthesized with the underlying synthesizer when given enough time, then a solution to the synthesis problem is returned.*

The proof that these two properties hold directly stems from observation of algorithms that are used for the implementation. It is easier to first show that general algorithms, given in Algorithm 15 and 16 respect these properties and then by building upon these results, show that the more specific algorithms, the ones given in Algorithm 17 and 18, also hold. An important remark is that the main algorithm will always gradually add only correct branches so that the partial solution respects the invariant that it encodes the correct behavior with respect to conditions that guard it. Note that since Leon can verify only terminating programs, the evaluation of input/output examples on enumerated expressions is important and it filters out the ones that can lead to non-terminating executions. The branch synthesis algorithm will try but never return a condition that does not guard an expression that represents a correct behavior. Since an exhaustive enumeration is done, eventually all expressions of the right type will be checked, including the correct one, if one can be synthesized and enumerated. These remarks together with the assumption that underlying tools are correct in their behavior can lead to explanation that both soundness and completeness properties hold for our synthesis approach driven by specifications.

4.3.7 Evaluation

We evaluated our synthesis approach on several examples that represent implementations of various widely-known and practical algorithms and data structures, including the examples introduced in Section 4.3.1. The examples are written in a purely functional subset of Scala and specifications are given with constructs *hole* and *passes* (which together represent the domain language of our synthesis, as described in Section 4.3.5).

Platform and settings for our experiments We used the same platform for conducting all experiments presented in this work, and the platform is described in Section 2.6. The algorithm that drives our implementation closely resembles the one given in Algorithm 17. This algorithm depends on and can be tuned by a couple of input parameters. The most influential parameter values that we adopted for the experiments were:

- Algorithm 18 tries to synthesize a branch and gives up after examining boolean snippets 20
- initially, 5 counter-example models are derived and put into the *cm* set, before algorithm for branch synthesis is invoked
- after performing synthesis of potential body expressions of type T, 50 expressions were tested and enqueued according to the input/output examples
- we ran the procedure without imposing a time limit

We selected these particular values of parameters by the results we got after series of experimental, trial and error runs.

Code synthesis and results

For each synthesis problem we will present:

- code of the original implementation

- code on which the synthesis process was invoked
- code after the synthesis

We will include the code from the original implementation so that it can serve as a comparison to the resulting, synthesized code. The original implementation was verified to be correct with Leon, prior to running the experiments[20]. The code on which the synthesis process was invoked illustrates how we specified synthesis problems and characterize our experiments. The code shows the formal specification and input/output example pairs that were used to drive the synthesis. Note that, for the purpose of readability, we gave specifications with input/output examples in a separate commented section that follows the main code, rather than with the *passing* construct. We will use a simple way do denote lists - e.g. with [x, y, z] we denote a list with elements x, y, z or more precisely $Cons(x, Cons(y, Cons(z, Nil())))$ in our examples. Lastly, the code after synthesis represents the whole annotated function after its body was synthesized and inserted at the appropriate place in the code.

We will introduce a couple of representable examples in more detail and afterwards present a table that shows results and statistics of experiments ran with these examples.

List concatentation

The algorithm for concatenation of two lists (Section 3.1.1) was frequently exploited for demonstrative purposes in this chapter. Here, we will use it to examine the performance of our synthesis approach. The code for this example uses the previously adopted algebraic data type list representation and functions *size* and *content*.

The original code from the example for the function that performs list concatenation is given in Listing 3.3.

The following code served as the input to our synthesis:

```
def concat(l1: List, l2: List) = ({
  hole[List]
}) ensuring(res => content(res) == content(l1) ++ content(l2) &&
  size(res) == size(l1) + size(l2))
/* passes: ([], []) −> []; ([1], []) −> [1]; ([1], [3]) −> [1, 3];
([1, 3], [5]) −> [1, 3, 5] */
```

As the result, the following code was synthesized in 5784ms:

```
def concat(l1: List, l2: List) = {
  if (l1.isInstanceOf[Nil]) l2
```

[20] note that not all examples could be verified, e.g. the linear search algorithm

```
  else if (l2.isInstanceOf[Nil]) l1
  else Cons(l1.head, concat(l1.tail, l2))
}
```

It is interesting to note that the second branch is not necessary, but does not hurt the correctness of the code and even results in faster execution in cases in which second input list is empty. Also note that a different solution, that would respect the given postcondition but allow permutation of elements between the lists is prevented from synthesizing due to the specified input/output examples. The execution time was almost 6 seconds and this shows that the generate-and-test approach can incur a lot of time while searching for a solution even with the help of counter-example evaluation and ranking.

Insertion sort

This example represents a purely functional implementation of the insertion sort algorithm [15]. A version of the insertion sort algorithm implemented in this example sorts lists encoded as algebraic data structures[21] with integer elements (there is no loss of generality in storing integers as opposed to values of other types inside the list, as long the values can be compared with ¡= as given in the example below).

The code includes definitions of functions *size*, *content* (introduced in Section 3.1.1), appropriately annotated with formal specifications, together with the function *isSorted* that represents a predicate for sorted lists (it returns true if the order of elements in the list is non-decreasing):

```
def isSorted(l: List): Boolean = l match {
  case Nil() => true
  case Cons(x, Nil()) => true
  case Cons(x, Cons(y, ys)) => x <= y && isSorted(Cons(y, ys))
}
```

The sorting algorithm is implemented with two main functions:

sortedIns This function takes a list and an element to be insert and inserts the element at the right place in the input list. The implementation of this function that can be verified for correctness is:

```
def sortedIns(e: Int, l: List): List = {
  require(isSorted(l))
  l match {
    case Nil() => Cons(e,Nil())
```

[21]the usual representation we referred to multiple times in this work

```
    case Cons(x,xs) =>
          if (x <= e) Cons(x,sortedIns(e, xs))
          else Cons(e, l)
    }
  } ensuring(res => contents(res) == contents(l) ++ Set(e)
  && size(res) == size(l) + 1 && isSorted(res))
}
```

The precondition requires that the input list is sorted while the precondition requires that the new list is equal to the input list with the element inserted which respects the right order of its elements.

sort This is the main function that does the sorting. It takes a list (unordered) as an argument and returns a new list that represents the input list with its elements sorted. The function uses *sortedIns* function introuced previously.

```
def sort(l: List): List = (l match {
  case Nil() => Nil()
  case Cons(x,xs) => sortedIns(x, sort(xs))
}) ensuring(res => contents(res) == contents(l)
  && isSorted(res) && size(res) == size(l))
```

The function recursively goes through all elements of the input list and uses recursive calls to eventually achieve unfolding that inserts each element into a sorted, gradually filled list. The postconition states that the list should be sorted, while it should not change in terms of its size and contents.

In our experiments we tried to synthesize functions *sortedIns* and *sort*. In the following text we will present the code that is given to the synthesis process and the resulting, synthesized code, in the case of both of these functions.

The sorting function, *sort* The code given to the synthesis process is:

```
def sort(l: List): List = ({
  hole(l)
}) ensuring(res => contents(res) == contents(l)
&& isSorted(res) && size(res) == size(l))
/* passes: [] -> []; [1] -> [1]; [1, 3] -> [1, 3];
[1, 3, 5] -> [1, 3, 5]; [10, 7, 5] -> [5, 7, 10] */
```

The following code was synthesized in 4304ms:

```
def sort(l: List): List = ({
  if (l.isInstanceOf[Nil])
    l
  else
    sortedIns(l.head, sort(l.tail))
})
```

Note that the code exactly corresponds and encodes the body in the original code. This effectively means that process had two iterations of the main loop of the algorithm given in Algorithm 15 and that in the first, a correct branch was found (the one that represents Hoare triple, $\{l.isInstanceOf[Nil]\}\ l\ \{q\}$, where $q = contents(l) = contents(Nil) \wedge isSorted(l) \wedge size(l) = size(Nil)$).

The sorted insert function, *sortedIns* The code given to the synthesis process is:

```
def sortedIns(e: Int, l: List) = {
    require(isSorted(l))
    hole[List]
  } ensuring(res => contents(res) == contents(l) ++ Set(e)
  && isSorted(res) && size(res) == size(l) + 1)
/* passes: (3, []) -> [3]; (9, []) -> [9]; (3, [5, 7, 10]) -> [3, 5, 7, 10];
(6, [5, 7, 10]) -> [5, 6, 7, 10]; (9, [5, 7, 10]) -> [5, 7, 9, 10]; */
```

The following code was synthesized in 19763ms:

```
if (l.isInstanceOf[Nil]) Cons(e, l)
else if (e == l.head) Cons(e, l)
  else if (e < l.head) Cons(e, l)
    else sortedIns(l.head, sortedIns(e, l.tail)))
```

It is interesting that in this case the synthesized body of the function does not exactly correspond to the one found in the original code with respect to the branching structure of *if* terms. Note that it represents a correct body, moreover implements the same behavior as the one in the original, with respect to given specifications. More specifically, our tool synthesized an additional branch, i.e. two branches with conditions `e == l.head` and `e < l.head` and the same branch expression `Cons(e, l)`. This is due to the fact that after synthesizing the first branch, while synthesizing with the precondition *!l.isInstanceOf[Nil]*, the term `e == l.head` was enumerated and verified as correct before the term `e <= l.head`. Of course, in the following iteration, the precondition was updated to *!l.isInstanceOf[Nil]* $\wedge$ *!e == l.head* and the process found that the same branch expression `Cons(e, l)` is valid under the precondition `e < l.head`. Afterwards the else expression, `sortedIns(l.head, sortedIns(e, l.tail))`, was enumerated and validated.

Merge sort

The following example represents a purely functional implementation of the merge sort algorithm [15]. It operates on the same representation of lists as the previous example.

The original code includes functions *size*, *content* and *isSorted* that are implemented in the same way as in the previous example. These functions are used to specify the postcondition of the *mergeSort* function to be synthesized. The usual functional implementation of merge sort has two additional functions: *merge* that takes two sorted lists and merges them into a sorted list and a function that does the splitting of the input list in half [16]. Note that splitting is usually implemented with the help of built-in functions (like *take*) if they are available (which does not hold in our case, thus we introduced explicitly).

We introduced functions *merge* and *split* in our program with the behaviour according to the previous description, together with their formal annotations. For the sake of brevity, we will present these functions without their bodies:

```
def merge(a : List, b : List) : List = {
    require(isSorted(a) && isSorted(b))
    ...
  } ensuring(res =>
    isSorted(res) && contents(res) ==
      contents(a) ++ contents(b))
```

Listing 4.4: *merge* function

```
// Pair encodes (List, List)
def split(list: List): Pair = {
    ...
  } ensuring(res => contents(list) ==
    contents(res.fst) ++ contents(res.snd))
```

Listing 4.5: *split* function

The code given to the synthesis process is:

```
  def mergeSort(list : List) = ({
      hole[List]
  }) ensuring(res => contents(res) == contents(list) && isSorted(res))
/* passes: [] -> []; [10] -> [10]; [5, 10] -> [5, 10]; [10, 5] -> [5, 10];
[5, 10, 15] -> [5, 10, 15]; [15, 10, 5] -> [5, 10, 15]; */
```

The following code was synthesized in 79937ms:

```
def mergeSort(list : List) =
  if (list.isInstanceOf[Nil]) list
  else if (isSorted(list)) list
    else merge(mergeSort(split(list).fst), mergeSort(split(list).snd)))
```

An interesting remark about the synthesized solution is that it indeed represents correct code but an additional, unnecessary branch was synthesized, once more. The branch checks if the list is sorted and if true, returns the list without recursively calling the sort function. This is in fact interesting since the given code even performs faster than the original code on input examples where the list is already sorted. There are 3 input examples that represent already sorted lists, so the unnecessary synthesized branch got favored in the second iteration of the algorithm. After the expression was dequeued, the synthesize branch algorithm managed to find the appropriate condition for the branch. Finally, the last, most complex expression was synthesized.

Comparison of the results

Table 4.2 presents the overview of the synthesis results on the four previously described, representative examples.

Example name and function	Synthesized branches	Unnecessary branches	Total size	Given examples	Execution time (s)		
					Ver.	Test.	Overall
Concatenation	3	Yes	15	4	3.8	0.37	5.39
Insertion sort, sortedIns	4	Yes	27	5	14.63	0.72	19.76
Insertion sort, sort	2	No	10	5	3.35	0.23	4.30
MergeSort	3	Yes	16	6	51.46	5.1	79.94

Table 4.2: Results of the evaluation of synthesis of correct programs with respect to specifications. First column gives the example name. The following two columns state how many branches the synthesized solution contains and whether some of those were unnecessary for satisfying correctness. The following two columns give the cumulative size of the synthesized *if* expression together with how many input/output examples were provided as the input specification. Last three columns give execution time spent for verification, execution of tests and overall (synthesis and overhead).

The table shows that in all four examples our approach managed to synthesize correct code successfully. Note that the total size of synthesized *if* expressions was calculated by summing up sizes of all contained branch and condition expressions. Additionally, note that we did not explicitly measure synthesis time (and execution time of the enumeration of reconstructed terms) but left it as the cumulative execution time together with overheads in the whole process.

From the overview of the complexity of synthesized solutions, in terms of synthesized branches and total size of the expression, we can see that relatively complex, with size ranging from 10 to 27, but still correct expressions were synthesized with providing relatively little input/output example pairs to the synthesis process. Synthesis of the *sortedIns* function produced an expression of the largest size, namely 27, while the synthesis of *mergeSort* has proved to be the most difficult and took almost 80s to produce a solution. The classification of the execution time shows that the verification takes the largest portion of the execution time, while the execution of tests is relatively fast. This is expected since only a few of them is being executed for each enumerated expression.

Although we did not generalize our implementation to the extent to be able to provide an evaluation on a larger set of examples and to tune and analyze how individual techniques and parameter values affect the synthesis process, the results presented in this section can serve to get valuable insights into practical possibilities, limitations and further improvements of our approach to generating correct programs.

Chapter 5

Related work

Previous work on the type driven synthesis This line of work was started with the initial versions of our approach driven by types and the InSynth tool [60, 31, 29]. In the demo tool a theorem prover was used for classical logic for synthesis. Based on an extensive evaluation and various implementation improvements, the important conclusion was that the code completion problem is more related to the type inhabitation problem. Our current approach also provides a method to mine initial weights of declarations, which was very important for obtaining useful results.

Code snippet search Several tools including Prospector [46], XSnippet [66], Strathcona [34], PARSEWeb [74] and SNIFF [14] 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 initiate multiple queries. In contrast, we generated expressions at once. We could not effectively compare InSynth 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. Prospector [46] 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 the method calls that starts at $type_{in}$ and ends at $type_{out}$. Prospector ranks solutions by the length, preferring shorter solutions. On the other hand, 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 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 [66] 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 [34], 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, those examples can not be fitted into the code without additional interventions. PARSEWeb [74] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. In InSynth the length of a returned snippet also plays an important role in ranking the snippets but InSynth also has an additional component by taking into account also the proximity of derived snippets and the point where InSynth was invoked. The main idea behind the SNIFF [14] 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.

There are several tools for the Haskell API search. The Hoogle [56] 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 [59] search engine does not use types for searching functions: its search is based on function names. The main difference between Djinn [63] 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 [57]. In their tool Perelman et al. generate 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.

An approach that develops a search-engine that answers semantic code-search queries, deals with how an API is used, in a way that consolidates, distills, and ranks matching code snippets was presented in [50]. The presented tool, Prime, receives a query in a form of partial code, does a textual search of similar code on the web and uses consolidation techniques to merge multiple code snippets and rank the results. In Prime, the developer must have a general idea about the implementation in the desired code snippet in order to be able to provide a sensible query. While Prime uses semantic search to synthesize potentially complex algorithms it does not guarantee a correct code or even a code that type-checks.

Exsting support in IDEs InSynth is similar in operation to Eclipse content assist proposals [73] and IntelliJ [35], and it implements the same behaviour. More advanced solutions appeared recently, like the Eclipse code recommenders [12, 20], which use and expand knowledge base of API calls statistics in order to find the appropriate expressions and offer them to the developer with appropriate statistical confidence value. InSynth is fundamentally different from this approach (it even subsumes it) and can synthesise code fragments that never occurred in code previously. Such solutions can explore only the provided code snippets from the repository and do not perform synthesis of source code in the developer's run-time environment.

Proof assistants 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 [8]. 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.

Type driven synthesis and logical frameworks The use of type constraints was explored in interactive theorem provers, as well as in synthesis of code fragments. SearchIsos [19] 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 [78], but there is a large gap between a theoretical result and an effective tool. Furthermore, InSynth can not only enumerate terms but also rank them and return a desired number of best-ranked ones.

Grammatical frameworks Our work has a couple of related points with grammatical frameworks [61, 4]. A relation between abstract and concrete syntax used in grammatical frameworks can be observed in the way the synthesis driven by types represents and produces code snippets. The intermediate representation can be seen as the abstract syntax and the domain language as concrete, while extracting program declarations and reconstructing reduce to parsing and linearization of syntax trees, respectively. Techniques such as partial evaluation and tree transformations used in grammatical frameworks may be applied to proof trees in the reconstruction of type inhabitants.

Comparison with intuitionistic provers As having a witness term that a type is inhabited is a vital ingredient of our tool, one could think of InSynth as a prover for propositional intuitionistic logic (with substantial additional functionality). Among recent modern provers are Imogen [49] and fCube [21]. 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, which is not entirely surprising because these tools are not necessarily optimized for the task that we aim to solve, 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 if we only 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, which is essential for our application: using synthesis as a generalization of completion.

Synthesis with input/output examples One of the first works that addressed synthesis by examples, put inductive synthesis on a firm theoretical foundation and largely influenced our work is the one by Summers [70]. It is an explanation based generalization approach that initially constructs a non-recursive program (in terms of Lisp datatype S-expression) from traces and conditions that explain each input/output example and then generalizes the program according to similarities in traces. The approach thus widely relies on algorithmic processes and only partially on search. Our approach infers traces implicitly by constructing recursive programs with if conditions and recursive calls, instead of S-expressions. While it drastically relies on searching and usually synthesizes branches that cover multiple example pairs, it lacks the generalization step and thus is not effective in cases in which overspecialization of the synthesized program may occur.

Several more recent works present extensions of the classical approach to induction of functional Lisp-programs [39, 33, 40]. They present extensions which include synthesizing a set of equations (instead of just one), multiple recursive calls and systematic introduction of parameters. Interestingly, many of the shortcomings of these approaches, like reasoning about arbitrary datatypes, multiple parameters in I/O examples, nested recursive calls and user-defined declarations, are not present in our work. Reasoning about more than structural problems, specifically in the generalization part, is interesting extension of these approaches that can also be incorporated in our approach for solving the overspecialization issue.

Inductive programming and programming by demonstration Inductive (logic) programming, which explores automatic synthesis of (usually recursive) programs from incomplete specifications, most often being input/output examples, represents an interesting field of research in machine learning [23, 52]. Our approach strongly relates to approaches in inductive programming that generate all correct programs and test against the specification.

Although they focus on solving conceptually different and more specific problems, more specifically synthesis of string manipulation operations, a couple of approaches from the area of programming by demonstration influenced the ideas behind our approach [24, 44]. They present interesting ideas for learning from input/output examples that may be generalized and incorporated in our approach to synthesis driven by specifications.

Synthesis procedures and frameworks Several works address the approach of generalizing decision procedures into synthesis procedures that synthesize functions mapping inputs to outputs in decidable theories, such as linear arithmetic and sets [42] and systematic derivation of synthesis procedures with inference rules for combinations of theories [36,41]. Synthesis driven by specifications presented in our work does not require limiting the synthesis to decidable theories and can be incorporated into a synthesis procedure framework as a rule that searches for solutions in run-time. Our approach is thus related to the run-time approach of constraint solving from the area of constraint (logic) programming [43].

Counter-example guided iterative synthesis Due to recent advances in the field of SMT solvers, numerous papers begin to emerge that consider using for synthesis approaches based on the counter-example guided iterative synthesis [25]. The counterexample guided iterative synthesis technique provides a procedure for dealing with quantifier alternation in formulas. That effectively means that if the right underlying logic is supported, an effective procedure for synthesis in that logic exists. We were motivated by this technique for optimization purposes in our approach that synthesized correct code. The main difference in our approach is that we use counterexamples derived from models returned by the solver to filter out already synthesized expressions.

Chapter 6

Conclusions

We presented the design and implementation of a code completion inspired by complete implementation of type inhabitation for simply typed lambda calculus. Our algorithm uses succinct types, an efficient representation for types, terms, and environments that takes into account that the order of assumptions is unimportant. Our approach generates a graph representation of all solutions, 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. Our 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. Our tool and additional evaluation details are publicly available.

Additionally, we presented an approach to synthesizing correct programs with respect to specifications provided by the developer. Synthesizing correct code with respect to specifications of its behavior is a very challenging tasks. Our approach was motivated by works that laid foundations of software synthesis with examples and theorem proving. Our observation was that techniques and tools presented in those works are too restrictive in their expressiveness to be useful in practical modern software development. Our goal was to start from an approach that is able to solve the code synthesis problem in the general case and use it as the core for a more sophisticated procedure that would provide necessary flexibility for refinement and improvements until it converges to the appropriate practical value. The technique that lies at the core of our approach is based on few well-known and simple concepts but goes beyond the scope of possibilities of existing techniques and presents a great potential if utilized in a right way. Our approach has the aim enhance the technique by utilizing the existing state-of-the art tools for synthesis and verification.

We implemented a Scala compiler plugin that allows developers to describe behavior of an arbitrarily complex program both in terms of formal specifications and input/output example pairs and goes through the provided source code and fills out missing bodies of functions marked for synthesis. The technique uses state-of-the art synthesizer and verifier, namely InSynth and Leon. From the intial results of the evaluation of our tool on synthesizing several practical algorithms with a range of complexities, we can get a feeling of the potential and practical value that this approach can bring. The results show that our approach is able to synthesize code in those examples and while not showing great efficiency it provides insights into the potential for improvements.

Bibliography

[1] S. Gulwani. Dimensions in program synthesis. In T. Kutsia, W. Schreiner, and M. Fernndez, editors, *PPDP*, pages 13–24. ACM, 2010. 9

[2] Insynth, type-driven interactive synthesis of code snippets, official website, `http://lara.epfl.ch/w/insynth`. 45

[3] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Massachusetts, 2nd edition, July 1996. 62, 63, 85

[4] K. Angelov. *The Mechanics of the Grammatical Framework.* PhD thesis, Ph. D. thesis, Chalmers University of Technology, Gothenburg, Sweden, 2011. 120

[5] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer, 2010. 84

[6] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. II.* Oxford University Press, 2001. 24

[7] R. W. Blanc. Verification of Imperative Programs in Scala. 2012. 85, 86

[8] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - a functional language with dependent types. In *TPHOLs*, 2009. 119

[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA*, pages 123–133, 2002. 10

[10] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification.* Springer, 2007. 14, 15, 73, 81, 82, 83, 84, 96, 106

[11] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93:172–221, July 1991. 30

[12] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222, 2009. 119

[13] R. M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 1969. 15

[14] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. FASE '09, pages 385–400, 2009. 117, 118

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. 110, 113

[16] Rosetta code, a programming chrestomathy website. `http://rosettacode.org/wiki/Rosetta_Code`. 14, 89, 113

[17] H. B. Curry and R. Feys. *Combinatory Logic, volume 1.* North-Holland, Amsterdam, 1958. 25, 49

[18] L. M. de Moura and N. Bjrner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. 84

[19] D. Delahaye. Information retrieval in a Coq proof library using type isomorphisms. In *TYPES*, pages 131–147, 1999. 120

[20] Eclipse code recommenders. `http://www.eclipse.org/recommenders/`. 119

[21] M. Ferrari, C. Fiorentini, and G. Fiorino. fCube: An efficient prover for intuitionistic propositional logic. In *LPAR (Yogyakarta)*, 2010. 13, 53, 120

[22] P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *J. Symb. Comput.*, 15(5/6):778–805, 1993. 10, 15

[23] P. Flener and D. Partridge. Inductive programming. *Automated Software Engineering*, 8:131–137, 2001. 121

[24] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *POPL*, pages 317–330. ACM, 2011. 10, 121

[25] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 62–73. ACM, 2011. 10, 122

[26] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. On complete completion using types and weights. Technical report, EPFL, 2012. 30

[27] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. On Fast Code Completion using Type Inhabitation. Technical report, 2012. 30

[28] Scala ide team. scala ide developer documentation. 45

[29] T. Gvero, V. Kuncak, and R. Piskac. Code completion using quantitative type inhabitation. Technical Report EPFL-REPORT-170040, EPFL, July 2011. `http://infoscience.epfl.ch/record/170040`. 12, 117

[30] T. Gvero, V. Kuncak, and R. Piskac. Code Completion using Quantitative Type Inhabitation. Technical report, 2011. 94

[31] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011. 12, 94, 117

[32] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12(10):576–580, 583, October 1969. 83

[33] M. Hofmann. Igor2 - an analytical inductive functional programming system: tool demo. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 29–32, New York, NY, USA, 2010. ACM. 121

[34] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. ICSE '05, pages 117–125, 2005. 117, 118

[35] Intellij idea website, `http://www.jetbrains.com/idea/`, 2011. 11, 119

[36] S. Jacobs, V. Kuncak, and P. Suter. Reductions for synthesis procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013. 10, 122

[37] J. R. Josephson. *Abductive inference: Computation, philosophy, technology.* Cambridge Univ Pr, 1996. 94

[38] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming, 1993. 95

[39] E. Kitzelmann and M. Hofmann. Igor2 – an inductive functional programming prototype. In *System Demonstrations of the 18th European Conference on Artificial Intelligence*, 2008. 121

[40] E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Mach. Learn. Res.*, 7:429–454, Dec. 2006. 10, 121

[41] V. Kuncak. On interpolation for synthesis procedures. epfl technical report., 2012. 122

[42] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 316–329. ACM, 2010. 10, 122

[43] A. S. Kksal, V. Kuncak, and P. Suter. Constraints as control. In J. Field and M. Hicks, editors, *POPL*, pages 151–164. ACM, 2012. 10, 122

[44] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003. 121

[45] Z. Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008. 30

[46] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM. 10, 11, 117

[47] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992. 10

[48] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, Mar. 1971. 10, 14

[49] S. McLaughlin and F. Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In *CADE*, 2009. 13, 53, 120

[50] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 997–1016. ACM, 2012. 10, 119

[51] R. Mitchell and J. C. McKim. Design by contract, by example. In *TOOLS (39)*, pages 430–431. IEEE Computer Society, 2001. 83

[52] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994. 121

[53] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition edition, Nov. 2008. 19, 42, 63

[54] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999. 14, 89

[55] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM. 10

[56] Hoogle api search, `http://www.haskell.org/hoogle/`. 118

[57] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286, 2012. 10, 11, 119

[58] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. 25, 28, 42, 48, 62, 75

[59] Hayoo! api search, `http://holumbus.fh-wedel.de/hayoo/`. 118

[60] R. Piskac. Decision procedures for software verification and code synthesis. In *Eight-Minute Presentations at POPL'11, ACM SIGPLAN Symp. Principles of Programming Languages*, January 2011. 12, 117

[61] A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004. 120

[62] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, pages 211–258, 1980. 30

[63] The djinn theorem prover, `http://www.augustsson.net/Darcs/Djinn/`. 118

[64] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, third edition, Dec. 2009. 40, 93

[65] The eclipse foundation. the eclipse project, `http://www.eclipse.org/`. 19

[66] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA*, 2006. 117, 118

[67] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodk, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 167–178. ACM, 2007. 10, 52

[68] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67 – 72, 1979. 12

[69] K. Støvring. Higher-order beta matching with solutions in long beta-eta normal form. *Nordic J. Computing*, 13(1), 2006. 23, 24

[70] P. D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, Jan. 1977. 10, 121

[71] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, pages 298–315, Berlin, Heidelberg, 2011. Springer-Verlag. 14, 81, 85, 86, 89

[72] Leon project at lara, website, `http://lara.epfl.ch/leon/`. 16, 84, 89

[73] The Eclipse foundation, eclipse ide, website, `http://www.eclipse.org/`. `http://www.eclipse.org/`. 11, 119

[74] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *ASE*, 2007. 117, 118

[75] J. Tromp. Binary Lambda Calculus and Combinatory Logic. Technical report, 2006. 42, 48, 49, 50

[76] P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In *TLCA*, 1997. 10, 12

[77] R. J. Waldinger. *Constructing programs automatically using theorem proving.* PhD thesis, Carnegie Mellon University Department of Computer Science, 1969. 10

[78] J. B. Wells and B. Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, pages 262–277, 2004. 120