Interactive Synthesis of Code Snippets

Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac*

firstname.lastname@epfl.ch
Swiss Federal Institute of Technology Lausanne (EPFL)

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

1 Introduction

Algorithmic software synthesis from specifications is a difficult problem. Yet software developers perform a form of synthesis on a daily basis, by transforming their intentions into concrete programming language expressions. The goal of our tool, InSynth, is to explore the relationship and synergy between algorithmic synthesis and developers' activities by deploying synthesis for code fragments in interactive settings. To make the problem more tractable, InSynth aims to synthesize small fragments, as opposed to entire algorithms. InSynth builds code fragments containing functions drawn from large and complex libraries. It aims to save the developers the effort of searching for appropriate methods and their compositions. InSynth is deployed within an integrated development environment. When invoked, it suggests multiple meaningful expressions at a given program point, using *type information* and *test cases*.

InSynth primarily relies on type information to perform its synthesis task. When the developer needs a piece of code that computes a value of a given type, they declare the type of this value, using the usual syntax of the Scala programming language [OSV08]. They then invoke InSynth to find suggested code fragments of this type. To find the building blocks for the code fragments, InSynth examines the scope of the given declaration. It uses Ensime [Can11], an incremental Scala compiler integrated into the editor, to gather the available values, fields, and functions. The use of type information is inspired by Prospector [MXBK05], but InSynth has an important additional dimension: it handles generic (parametric) types [DM82]. Generic types are a mainstream mechanism to write safe and reusable code in, e.g., Java, ML, and Scala. They are particularly frequent in libraries.

* Alphabetical order of authors.

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

InSynth implements a custom resolution-based algorithm to find multiple proofs representing candidate code fragments. The use of resolution is related to the traditional deductive program synthesis [MW80], but our approach attempts to derive code fragments by using type information instead of the code itself. As a post-processing step, InSynth filters out the candidate code fragments that crash the program, or that violate assertions or postconditions. This functionality incorporates input/output behavior [JGST10], but uses it mostly to improve the precision of the primary mechanism, the type-driven synthesis.

We believe that an important aspect of the software development process is that an accurate specification is often not available. A synthesis tool should be equipped to deal with under-specified problems, and be prepared to generate multiple alternative solutions when asked to do so. Our algorithm fulfills this requirement: it generates multiple solutions and ranks them using a system of weights. The current weight computation takes into account the proximity of values to the point in which the values are used, as well as user-specified hints, if any. A database of code samples, if available, could be used to derive weights, providing effects similar to some of the previous systems [SC06,MXBK05]. Given a weight function, InSynth directs its search using a technique related to ordered resolution [BG01].

Contributions. InSynth is an interactively deployed synthesis tool based on parameterized types, test cases, and weights indicating preferences. Its algorithmic base is a variation of ordered resolution. We have found InSynth to be fast enough for interactive use and helpful in synthesizing meaningful code fragments.

2 Examples

Consider the problem of retrieving data stored in a file. Suppose that we have the following definitions:

```
def fopen(name:String):File = { ... }
def fread(f:File, p:Int):Data = { ... }
var currentPos : Int = 0
var fname : String = ""
def getData():Data = □
```

The developer is about to define, at the position marked by □, the body of the function getData that computes a value of type Data. When the developer invokes InSynth, the result is a list of valid expressions (snippets) for the given program point, composed from the values in the scope. Assuming that among the definitions we have functions fopen and fread, with the types shown above, InSynth will return as one of the suggestions fread(fopen(fname), currentPos), which is a simple way to retrieve data from the file given the available operations. In our experience, InSynth often returns snippets in a matter of milliseconds. Such snippets may be difficult to find manually for complex and unknown APIs, so InSynth can also be thought as a sophisticated extension of a search and code completion functionality.

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

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

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

Using code behavior. The next example shows how InSynth applies testing to discard those snippets that would make code inconsistent. Define the class FileManager containing methods for opening files either for reading or for writing.

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

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

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

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

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

3 Foundations and Algorithm

Our main algorithm is based on first-order resolution. We therefore formalize type constrains in first-order logic. We introduce a predicate $\texttt{hasType}(v, T)$ to indicate that a value v is of a type T. We use a function symbol `arrow` to indicate the function type constructor ($\rightarrow$). First-order logic makes it possible to encode polymorphism using universally quantified variables.

The ranking of the snippets and the entire algorithm strongly rely on a system of weights. The system considers snippets of a smaller weight as preferable to those of a larger weight. The weights of terms extend to the weights of clauses, as in the multiset ordering of clauses in first-order resolution [BG01].

To begin with, we define an ordering on the symbols and assign a weight to each symbol. The user-preferred symbols have the smallest weight (highest preference). They are followed by the local symbols occurring in the current method. The remaining symbols of the corresponding class have a larger weight than the local symbols. Finally, the symbols outside the current class have the largest weight. This includes symbols from the imported libraries and APIs.

Once the ordering and the weights of the symbols are fixed, we compute the weight of a term similarly as in the Knuth-Bendix ordering. The only difference is that we additionally recalculate the weight of every term containing a user-preferred symbol. We do this so that they do not "vanish" when combined with symbols of a larger weight.

Snippet Synthesis Algorithm. Figure 1 describes the basic version of our algorithm. It takes as an input a partial Scala program and a program point where the user asks for a code snippet. Additionally, it also takes as an argument a resource bound, the maximum number of resolution steps.

The first step of the algorithm is to traverse the program syntax tree, create the clauses, and assign the weights to the symbols and clauses. We pick a minimal-weight clause and resolve it with all other clauses of a larger weight. If we derive a contradiction (empty clause), we extract its proof tree. To generate

multiple solutions, we use the proof tree to derive a blocking clause that prevents the same derivation of the empty clause in the future. We add this blocking clause to the clause set. We repeat this procedure until either the clause set becomes saturated, or the given threshold on the resolution steps is exceeded. We then reconstruct terms from the collected proof trees, and create the code snippets. We test the generated snippets by invoking any user-defined test cases and discarding the snippet for which the code crashes.

Backward Reasoning. The implementation of InSynth combines the algorithm described in Figure 1 with backward reasoning. With ? T we denote the query asking for a value of the type T. The main rule we use is

$$\frac{\texttt{hasType}(x, \texttt{Arrow}(T_1, T_2)) \qquad ?\ T_2}{?\ T_1}$$

By applying backward reasoning we were able to accelerate search for solutions compared to using purely forward reasoning.

INPUT: partial Scala program, program point, maximal number of steps
OUTPUT: list of code snippets

```
def basicSynthesizeSnippet(p : PartialScalaProgram, maxSteps : Int) : List[Snippet] = {
  var weightedClauses = extractClauses(p)
  var saturated = false
  var solutions = emptySet
  var step = 0
  while (step < maxSteps && !saturated) {
    val c : Clause = pickMinWeight(weightedClauses)
    saturated = true
    for (c' ← weightedClauses if weight(c) ≤ weight(c') && c != c')) {
      val newC = resolution(c,c')
      if !(newC in weightedClauses) {
        saturated = false
        if (newC.isEmptyClause) {
          val s = extractSolution(newC)
          solutions = solutions ∪ { s }
          val cBlock = createClausePreventingThisProof(s)
          weightedClauses = weightedClauses union { cBlock }
        }
      }
    }
    step++
  }
  return (solutions.map(proofToSnippet)).filter(passesTest(p))
}
```

Fig. 1. Basic algorithm for synthesizing code snippets

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

Fig. 2. Basic algorithm for synthesizing code snippets

4 InSynth Implementation and Evaluation

InSynth is implemented in Scala and built on top of the Ensime plugin [Can11]. It can therefore directly use program information computed by the Scala compiler, including abstract syntax trees and the inferred types. Furthermore, it can generate an appropriate pop-up window with suggested synthesized snippets and allow the user to interactively select the desired fragment.

Figure 2 gives an idea of the performance of the system. We ran all examples on Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM. The running times to find the first solution are usually bellow two milliseconds. Our experience suggests that the algorithm scales well. As an illustration, we were able to synthesize a snippet containing six methods in 0.093 seconds from the set of 320 declarations. Times to encode declarations into FOL formulas range from 0.015 (Calendar) to 0.046 (FileWriter) seconds. If the synthesized snippets need to use more methods from imported libraries, the synthesis typically takes longer, but is typically fast enough to be useful. The above examples and the system InSynth are available on the following web site: `http://lara.epfl.ch/w/insynth`.

References

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.

[Can11] Aemon Cannon. Ensime. `https://github.com/aemoncannon/ensime/`, 2011. Retrieved 20 April.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, 2010.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, 2005.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[SC06] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *OOPSLA*, 2006.