Interactive Synthesis of Code Snippets*

Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac**

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

Abstract. We describe a tool that applies theorem proving technology to synthesize code fragments that use given library functions. Our approach takes into account polymorphic type constraints as well as code behavior. 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, isynth, 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, isynth aims to synthesize small fragments, as opposed to entire algorithms. isynth builds code fragments containing functions drawn from large and complex libraries, saving the developers from a substantial effort to search for appropriate methods and their compositions. isynth is a synthesis algorithm deployed within an integrated development environment. It uses an interface that conceptually extends the familiar code-completion feature. When invoked, isynth suggests multiple meaningful expressions at a given program point, using type information and test cases.

isynth primarily relies on type information to perform its synthesis task. A user invokes isynth at the program point when defining some value. The type of the value is known but not its definition. The user is interested in getting suggestions for the value definition. To find the building blocks for code fragments, isynth examines the current scope of an incremental Scala compiler integrated into the editor and gathers the available values, fields, and functions. The use of type information is inspired by Prospector [MXBK05], but isynth has an important additional dimension: it handles not only simple but also generic (parametric) types [DM82], which are a mainstream mechanism to write safe and reusable code in languages including ML, Java, C#, and Scala.

* Note to reviewers: The authors have no prior nor concurrently submitted paper on this tool or technique in any venue. This tool submission is also available as an EPFL technical report.

** Alphabetical order of authors.

The support for generic types is a fundamental generalization compared to previous tools. The resulting set of possible intermediate types is no longer finite and the synthesis of a value of a given type becomes undecidable. isynth is based on an encoding of the synthesis problem into 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 the known connections between proof theory and type theory. In type-theoretic terms, isynth attempts to check whether there exists a term of a given type in a given polymorphic type environment. If such terms exist, the goal is to produce a finite ranked subset of them.

isynth implements a custom resolution-based algorithm to find multiple proofs representing candidate code fragments that satisfy the typing constraints. The use of resolution is related to traditional deductive program synthesis [MW80], but our approach attempts to derive code fragments by using type information instead. In addition, isynth implements a filtering functionality, which inserts the candidate fragments into the code. It then tests the program for the absence of crashes, including violations of assertions or postconditions. This functionality incorporates input/output behavior [JGST10], but uses it mostly to improve the precision of the primary mechanism, 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. 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, isynth directs its search using a technique related to ordered resolution [BG01].

Contributions. In summary, we present isynth, the first interactively deployed synthesis tool based on parameterized types, test cases, and weights indicating preferences. isynth is based on an implementation of a variation of an ordered resolution calculus. We have found isynth to be fast enough for interactive use and helpful in synthesizing meaningful code fragments.

2 Examples

As a simple first example, 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
...
def getData():Data = ■
```

There is a number of definitions in the scope. 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 isynth, 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, of types shown above, the tool 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, isynth often returns snippets in a matter of milliseconds. Such snippets may be difficult to find manually for complex and unknown APIs, so isynth 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 isynth. 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 = ■
```

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

Using code behavior. The next example shows how isynth 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 would be based only on the type inferences rules, isynth would return both **new** File(name, WRITE) and **new** File(name, READ). However, it also checks the method contract (pre- and post-conditions) and verifies whether each of the returned snippets complies with them. Because of postconditions requiring that the file is open for reading, isynth discards the snippet **new** File(name, WRITE) and returns only **new** File(name, READ).

Applying user preferences. The last example demonstrates a way in which a user can influence the ranking of the returned solutions. This is an important issue, because, due to the large number of solutions found, it is crucial to identify those that would be more valuable for the user. In this example, we define the class Calendar for managing the events.

```
object Calendar {
  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 wants to get a code snippet for remove. Running the above example without any user suggestions returns reserve(user, date) and getEvent(user, date), in this order. When invoking isynth, a user can also give a list of strings as input. Those strings are names of methods which the user wants to appear in the returned code snippet. We again run isynth, having "getEvent" as a user suggestion, and the ranking of returned snippets changed: this time getEvent(user, date) was the first returned result. isynth ranks the results based on the weight function.

3 Foundations and Algorithm

The main algorithm is based on first-order resolution and thus we formalize type constrains in first-order logic. We introduce predicate `hasType` to describe that value v is of type T: $\texttt{hasType}(v, T)$. We use Hindley-Milner type description and interpret $\rightarrow$ with the special function symbol `arrow`. First-order logic formalism makes possible to easily encode polymorphism using universally quantified variables.

We also added weights to clauses. Those weights are slightly different than the weights based on multiset ordering of clauses and used in the first-order provers [BG01]. To begin with, we define an ordering on the symbols and to each symbol we assign the weight. This ordering and weights are defined as follows: the user preferred symbols have the smallest weight and are the greatest. They are followed by the local symbols occurring in the method. The remaining symbols of the corresponding class have greater weight than the local symbols. Finally, the symbols of the greatest weight are those which do not occur in the class. Those are symbols coming from various API and library methods.

Once the ordering and the weights of the symbols are fixed, we compute the weight of a term similarly as in 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 a symbol of a greater 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 we ask for a code snippet. Additionally, it also takes as an argument 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 greater weight. If we derive a contradiction (empty clause), we extract its proof tree. Moreover, based on this proof tree we derive a new clause that prevents the same derivation of the empty clause in the future. This new clause is then added to the clause set. We repeat this procedure until the clause set becomes saturated or the given threshold on the resolution steps is exceeded. Finally, we reconstruct terms from the proof trees, and create the code snippets. They snippets are further tested by invoking a test case that involves the code and discarding the snippet if the code crashes.

Backward Reasoning. In isynth we combine 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

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

This way we managed to accelerate search for solutions.

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

```
def basicSynthesizeSnippet(p : partial Scala program, 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' <− weight(c) < weight(c') || (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 union { 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

4 isynth Implementation and Evaluation

The system isynth is implemented in Scala and built on top of the Ensime plugin, which allows the access to the Scala compiler's internal information, as for instance to ASTs. It also allows to display code snippets back to the user in interactive way.

Our intention is to use isynth within a software development environment. When invoked, isynth runs the algorithm described in Section 3 and tries to synthesize the required code. If it finds suitable code snippets, ranks them based on their weights, and it returns them in a popup menu. If the user finds the required snippet in that list, isynth inserts the chosen value in the program.

To illustrate performance, consider Figure 2. All those examples were ran on Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM. Times to execute those examples were usually less than two milliseconds. The tests that we ran indicate that our system scales well. For instance, we were able to synthesize a snippet using six methods in 0.72 seconds from the set of 442 declarations.

Program	# Loaded Declarations	# Methods in Synthesized Snippets	Time [s]
FileReader	5	4	0.001
Map	3	3	< 0.001
FileManager	3	7	0.001
Calendar	29	3	0.001
FileWriter	442	6	0.72
SwingBorder	161	2	0.02
TcpService	112	3	0.62

Fig. 2. Basic algorithm for synthesizing code snippets

The above examples and the system isynth are available on the following web site: `http://lara.epfl.ch/web2010/isynth`.

References

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

[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.

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