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).

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

60. Quotient in Polar Representation

$$\frac{z_1}{z_2}=\frac{r_1(\cos\varphi_1+i\sin\varphi_1)}{r_2(\cos\varphi_2+i\sin\varphi_2)}=\frac{r_1}{r_2}\left[\cos(\varphi_1-\varphi_2)+i\sin(\varphi_1-\varphi_2)\right]$$

61. Power of a Complex Number

$$z^n=\left[r(\cos\varphi+i\sin\varphi)\right]^n=r^n\left[\cos(n\varphi)+i\sin(n\varphi)\right]$$

62. Formula "De Moivre"

$$(\cos\varphi+i\sin\varphi)^n=\cos(n\varphi)+i\sin(n\varphi)$$

63. Nth Root of a Complex Number

$$\sqrt[n]{z}=\sqrt[n]{r(\cos\varphi+i\sin\varphi)}=\sqrt[n]{r}\left(\cos\frac{\varphi+2\pi k}{n}+i\sin\frac{\varphi+2\pi k}{n}\right),$$

where

$k=0,1,2,\ldots,n-1.$

64. Euler's Formula

$$e^{ix}=\cos x+i\sin x$$

101. $\dfrac{1}{\sqrt[n]{a}} = \dfrac{\sqrt[n]{a^{n-1}}}{a}, \; a \neq 0.$

102. $\sqrt{a \pm \sqrt{b}} = \sqrt{\dfrac{a + \sqrt{a^2 - b}}{2}} \pm \sqrt{\dfrac{a - \sqrt{a^2 - b}}{2}}$

103. $\dfrac{1}{\sqrt{a} \pm \sqrt{b}} = \dfrac{\sqrt{a} \mp \sqrt{b}}{a - b}$

2.5 Logarithms

Positive real numbers: x, y, a, c, k
Natural number: n

104. Definition of Logarithm
$y = \log_a x$ if and only if $x = a^y, \; a > 0, \; a \neq 1.$

105. $\log_a 1 = 0$

106. $\log_a a = 1$

107. $\log_a 0 = \begin{cases} -\infty & \text{if } a > 1 \\ +\infty & \text{if } a < 1 \end{cases}$

108. $\log_a (xy) = \log_a x + \log_a y$

109. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$