On Complete Completion using Types and Weights

Tihomir Gvero Viktor Kuncak
Ivan Kuraj
École Polytechnique Fédérale de Lausanne (EPFL),
Switzerland
firstname.lastname@epfl.ch

Ruzica Piskac
Max-Planck Institute for Software Systems, Germany
piskac@mpi-sws.org

Abstract

Developing modern software applications typically involves composing functionality from existing libraries. This task is difficult because libraries may expose many methods to the developer. To help developers in such scenarios, we present a technique that synthesizes and suggests valid expressions of a given type at a given program point. As the basis of our technique we use type reconstruction for lambda calculus terms in long normal form. We introduce a succinct representation for type judgements that merges types into equivalence classes to reduce the search space, then reconstructs any desired number of solutions on demand. Furthermore, we introduce a method to rank solutions based on weights derived from a corpus of code. We implemented the algorithm and deployed it as a plugin for the Eclipse IDE for Scala. We show that the techniques we incorporated greatly increase the effectiveness of the approach. Our evaluation benchmarks are derived from real code and are made available for future benchmarking of code synthesis driven by types.

1. Introduction

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 [22] and IntelliJ [11] 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 user assistance 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. 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 [15] or the pioneering work on the Prospector tool [13].

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 similarly look up T by examining the declared type appearing left of the cursor in the editor. 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 lambda calculus corresponds to provability in propositional intuitionistic logic; it is decidable and PSPACE-complete [19, 24]. We developed a version of the algorithm that is complete in the lambda calculus sense, so it is able to synthesize not only function applications, but also lambda abstractions. 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 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. 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.

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 identify several proofs determining the expressions of the desired type, and rank them according to their weight. 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. This suggests that InSynth can efficiently and effectively help the user 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 [7, 14]. 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.

2. Motivating Examples

Here we illustrate the functionality of InSynth through several examples. The first example is from the online repository of Java API examples `http://www.java2s.com/`. The second example is a real world example taken from code base of the Scala IDE for Eclipse[1]. The original code of the two examples imports only declarations from a few classes. To make the problem much harder we import all declarations from packages where those classes reside. The final example demonstrates how InSynth deals with subtyping.

Sequence of Streams. Our first 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 1. If we invoke 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 user. 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.

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) {
    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 user 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.

3. 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 and let E be a set of expressions (variables). 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 τ_i. 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 lambda calculus and in the succinct types calculus, respectively.

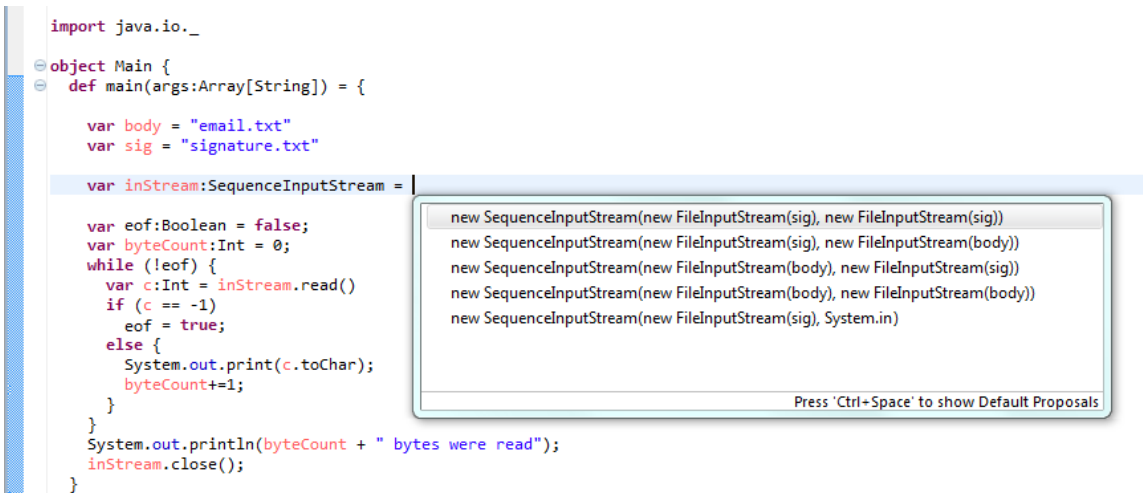
3.1 Simply Typed Lambda Calculus for Deriving Terms in Long Normal Form

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

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

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

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



```
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 1. InSynth suggesting five highest-ranked well-typed expressions synthesized from declarations visible at a given program point

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

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

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

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

We restrict the APP rule in order to derive only the terms that are in so-called *long normal form* [20]. Our main motivation is to find suitable code snippets efficiently. 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 expressivity of our calculus. Each simply-typed term can be converted to its long normal form [1, 20]. We now formally define long normal form.

DEFINITION 3.1 (Long Normal Form). *A judgement $\Gamma_o \vdash_\lambda e : \tau_e$ is in long normal form if the following holds:*

- $e \equiv \lambda x_1 \ldots x_m.f e_1 ... e_n$
- $\tau_e \equiv \tau_1 \to \ldots \to \tau_m \to \tau$
- *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 $\lambda x.fx : \tau_1 \to \tau_2$ is in long normal form.

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

3.2 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 lambda calculus this code translates to the type environment $\Gamma_o = \{a : \mathrm{Int}, f : \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int} \to \mathrm{String}\}$. 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. In this example, succinct types enable us to find an inhabitant in only one step.

DEFINITION 3.2 (Succinct Types). *Let B_S be a set containing basic types. Succinct types t_s are constructed according to the grammar:*

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

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

A type declaration $f : \{t_1, \ldots, t_n\} \to t$ 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 $\emptyset \to t$ 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 $\emptyset \to t$ equivalent.

Every type $\tau \in \tau_\lambda(B)$ can be converted into a succinct type in $t_s(B)$. With $\sigma : \tau_\lambda(B) \to t_s(B)$ we denote 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 $\sigma(v) = \emptyset \to v$. Let A (arguments) and T(type) be two functions defined on $t_s(B_S)$ as follows:
$A(\{t_1, \ldots, t_n\} \to v) = \{t_1, \ldots, t_n\}$
$T(\{t_1, \ldots, t_n\} \to v) = v$

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

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

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

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

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

It can be shown by reduction to reasoning about Venn regions 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 prove that the same holds for any union of type environments.

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 lambda 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 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.

$$\textsc{Abs}\ \frac{\Gamma \cup S \vdash_S \pi : t}{\Gamma \vdash_S S \to t}$$

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

Figure 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 $\Gamma \vdash_S \emptyset \to \text{Int}$. 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.

3.3 Soundness and Completeness of Succinct Calculus

In this section we show that the calculus in Figure 3 is sound and complete with respect to synthesis of lambda terms in long normal form.

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

- $\Gamma = \sigma(\Gamma_o)$
- $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$
- $S = \sigma(\{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = \{\sigma(\tau_1), \ldots, \sigma(\tau_n)\}$
- $\Gamma' = \sigma(\Gamma'_o) = \sigma(\Gamma_o) \cup \sigma(\{x_1 : \tau_1, \ldots, x_n : \tau_n\})$. From the last statement it follows that $\Gamma' = \Gamma \cup S$.
- $t = \sigma(\tau) = \sigma(\tau_1 \to \ldots \to \tau_n \to v) = S \to v$

THEOREM 3.3. *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then $\sigma(\Gamma_o) \vdash_S \sigma(\tau)$.*

Proof By induction on $\mathcal{L}(e)$.

Base case: If $\mathcal{L}(e) = 1$, then $e \equiv \lambda x_1 \ldots x_n.a$ and $\tau \equiv \tau_1 \to \ldots \to \tau_n \to v$. From the fact that $\Gamma_o \vdash_\lambda e : \tau$ by the (lambda) ABS rule we know that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda a : v$. Because the judgement is in long normal form, we also know that $a : v$ is an element of Γ'_o. Knowing that $a : v \in \Gamma'_o$ by the definition of σ we also know that $v \in \Gamma'$. By applying the APP rule it follows that $\Gamma' \vdash_S @\emptyset : v$. Because $\Gamma' = \Gamma \cup S$, by applying the ABS it follows $\Gamma \vdash_S S \to v$. By simple substitutions we have $\sigma(\Gamma_o) \vdash_S \sigma(\tau)$. This way we proved that for all judgements of the length 1, the theorem claim holds.

Case $\mathcal{L}(e) = k$: If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then $\sigma(\Gamma_o) \vdash_S \sigma(\tau)$.

Case $\mathcal{L}(e) = k + 1$: If $\mathcal{L}(e) = k + 1$, then $e \equiv \lambda x_1 \ldots x_n.f e_1 \ldots e_m$ and $\tau \equiv \tau_1 \to \ldots \to \tau_n \to v$. Just like before by the (lambda) ABS rule we know that $\Gamma'_o \vdash_\lambda f e_1 \ldots e_m : v$. Each e_i must have some type ρ_i, $i = [1..m]$. Then f has the type $\rho_1 \to \ldots \to \rho_m \to v$. By the (lambda) APP rule we have $f : \rho_1 \to \ldots \to \rho_m \to v \in \Gamma'_o$ and $\Gamma'_o \vdash_\lambda e_i : \rho_i$. When we apply σ we get $\sigma(\rho_1 \to \ldots \to \rho_m \to v) \in \Gamma'$ i.e $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \to v \in \Gamma'$. From the hypothesis it follows that $\Gamma' \vdash_S \sigma(\rho_i)$ because the length of e_i is less or equal to k. Now we apply the (succinct) APP rule to $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \to v \in \Gamma'$ and all $\Gamma' \vdash_S \sigma(\rho_i)$ and conclude $\Gamma' \vdash_S @\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} : v$. Finally, we apply the (succinct) ABS rule to $(\Gamma \cup S) \vdash_S @\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} : v$ and conclude $\Gamma \vdash_S @S \to v$. Again, by simple substitutions we have $\sigma(\Gamma_o) \vdash_S \sigma(\tau)$. This way we proved that for all judgements of the length $k + 1$, the theorem claim holds.

We are interested in generating any desired number of expressions of a given type without missing any expressions equivalent up to β reduction. To formulate a stronger form of completeness that captures this ability, we introduce two functions: CL and RCN. The CL function takes as arguments a succinct type environment Γ and a succinct type $S \to t$. It returns the set of all patterns $@S_1 : \tau$ derived in $\Gamma \cup S$:

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

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. The algorithmic description of the RCN function is given in Figure 4. Having CL and RCN functions, we can formalize the completeness theorem. It states that each judgement in long normal form derived in the standard lambda calculus, can be also derived by reconstructing it from the existence of type derivation of the succinct calculus.

```
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 $\cup$ $\{\lambda x_1 \ldots x_n.fe_1 \ldots e_m\}$
    return TERMS
fun Select($\Gamma_o$, $t$) := {v:$\tau$ | v:$\tau$ $\in$ $\Gamma_o$ and $\sigma(\tau) = t$}
```

Figure 4. A function that constructs lambda terms in long normal form up to given length d.

THEOREM 3.4 (Soundness and Completeness). *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form then the following holds:*

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

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

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

Our first goal is to show that $(\Gamma \cup S,\ @\emptyset{:}v) \in \mathsf{CL}(\Gamma, S \to v)$. By the CL definition this is equivalent to proving that $(\emptyset \to v) \in (\Gamma \cup S)$ and $\forall t' \in \emptyset. \Gamma \cup S \vdash_S t'$. The second statement is trivially **true**, whereas by convention the first we reduce to $v \in (\Gamma \cup S)$. From $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.a : \tau_1 \to \ldots \to \tau_n \to v$ it follows that $\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash_\lambda a : v$ by the (lambda) ABS rule. From the (lambda) APP rule we conclude that $a : v \in (\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = \Gamma'_o$. Note that if n is zero we do not need to apply the (lambda) ABS rule, but can directly apply the (lambda) APP rule. Now we apply σ and get $\sigma(a : v) \in \sigma(\Gamma'_o)$, i.e., $v \in (\Gamma \cup S)$. This shows that $(\Gamma \cup S,\ @\emptyset{:}v) \in \mathsf{CL}(\Gamma, S \to v)$.

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

We can conclude that the body of the second **foreach** loop will be executed. We select $a : v$ to be the type declaration $f : t_o$. Therefore, $m = 0$. This means that the empty tuple will execute the third **foreach** loop only once and put $\lambda x_1 \ldots x_n.a$ into TERMS. This proves that $\lambda x_1 \ldots x_n.a \in \mathsf{RCN}(\Gamma_o, \tau_1 \to \cdots \to \tau_n \to v, 1)$.

Case $\mathcal{L}(e) \leq k$: If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form and $\mathcal{L}(e) \leq k$ then the following holds:

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

Case $\mathcal{L}(e) = k + 1$: When $\mathcal{L}(e) = k + 1$, then $e \equiv \lambda x_1 \ldots x_n.he_1 \ldots e_m$ and $\tau \equiv \tau_1 \to \ldots \to \tau_n \to v$. It also holds that e_i are terms in LNF and $\mathcal{L}(e_i) \leq k$. We assume that each e_i has a type ρ_i. We have to prove that $\lambda x_1 \ldots x_n.he_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau_1 \to \cdots \to \tau_n \to v, k + 1)$. We again follow the **else** branch in RCN.

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

Let us prove the statements. From $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.he_1 \ldots e_m : \tau_1 \to \cdots \to \tau_n \to v$ it follows that $\Gamma'_o \vdash_\lambda he_1 \ldots e_m : v$, by the (lambda) Abs rule. From the (lambda) APP rule we conclude that $(h : \rho_1 \to \cdots \to \rho_m \to v) \in \Gamma'_o$. Note that if n is zero we do not need to apply the (lambda) ABS rule, but can directly apply the (lambda) APP rule. After we apply σ, we have $\sigma(h : \rho_1 \to \cdots \to \rho_m \to v) \in \sigma(\Gamma'_o)$, i.e., $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \to v \in \Gamma \cup S$. From the last application of the (lambda) App rule we have $\Gamma'_o \vdash_\lambda e_i : \rho_i$. Using Theorem 3.3 we have that $\Gamma \cup S \vdash_S \rho_i$, for $i = [1..m]$. Thus, by (succinct) App rule we conclude that $(\Gamma \cup S,\ @\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\}{:}v) \in \mathsf{CL}(\Gamma, S \to v)$ holds.

Our second goal is to prove $(h : \rho_1 \to \cdots \to \rho_m \to v) \in \mathit{Select}(\Gamma'_o, \{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \to v)$. By the Select definition this is equivalent to proving that $(h : \rho_1 \to \cdots \to \rho_m \to v) \in \Gamma'_o$ and $\{\sigma(\rho_1), \ldots, \sigma(\rho_m)\} \to v = \sigma(\rho_1 \to \cdots \to \rho_m \to v)$. We have previously prove the first statement, and the second statement follows from the σ definition.

Therefore, we conclude that the body of the second **foreach** loop will be executed, whereas we select $h : \rho_1 \to \cdots \to \rho_m \to v$ to be $f : t_o$. Finally, we have to prove that $e_i \in T_i = \mathsf{RCN}(\Gamma'_o, \rho_i, d - 1)$. This follows from the inductive hypothesis and Lemma 3.7 because $d - 1 = k$ and $k \geq \mathcal{L}(e_i)$. Therefore, one iteration in third **foreach** loop will contain $e_1, \ldots, e_m$ sub-terms in the correct order. (The argument order is preserved due to the order of argument types in t_o). Finally, we use them to construct $\lambda x_1 \ldots x_n.he_1 \ldots e_m$ and put it into TERMS. This proves that $\lambda x_1 \ldots x_n.he_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau_1 \to \cdots \to \tau_n \to v, k + 1)$. Note that if n is zero then $e \equiv he_1 \ldots e_m$.

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

Base case: When $\mathcal{L}(e) = 1$, $e \equiv \lambda x_1 \ldots x_n.a$ and $\tau \equiv \tau_1 \to \cdots \to \tau_n \to v$. We have to prove that $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.a : \tau_1 \to \cdots \to \tau_n \to v$. If there exists $\lambda x_1 \ldots x_n.a \in \mathsf{RCN}(\Gamma_o, \tau, 1)$ then there is an execution of the **else** branch in RCN that produces $\lambda x_1 \ldots x_n.a$. (Because $\mathcal{L}(e) = d = 1$ we conclude that the first execution will be the one that will produce e.) Also m must be zero because only empty tuple can produce $\lambda x_1 \ldots x_n.a$. Moreover, $t_0 = v$. It follows that there exist a pattern p and Γ'_o such that $a : v \in$ Select(Γ'_o, p). Note that by the Select definition every element that belongs to Select(Γ'_o, p) also belongs to Γ'_o, i.e., $a : v \in \Gamma'_o$. We know that there exists Γ_o such that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, and it will be exactly an environment that is passed as an argument to $\mathsf{RCN}(\Gamma_o, \tau, 1)$. Types τ_j, $j = [1..n]$ are the corresponding argument types that appear in the right order in τ. Thus we can first apply the (lambda) ABS and then the (lambda) APP rule and get $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.a : \tau_1 \to \cdots \to \tau_n \to v$, i.e., $\Gamma_o \vdash_\lambda e : \tau$. Note that if n is zero we do not need to apply the (lambda) ABS rule, but can directly apply the (lambda) APP rule.

Case $\mathcal{L}(e) \leq k$: If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form and $\mathcal{L}(e) \leq k$ then the following holds:

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

Case $\mathcal{L}(e) = k + 1$: Next we have that $e \equiv \lambda x_1 \ldots x_n.he_1 \ldots e_m$ and $\tau \equiv \tau_1 \to \cdots \to \tau_n \to v$. We have to prove that $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.he_1, \ldots e_m : \tau_1 \to \cdots \to \tau_n \to v$. If $\lambda x_1 \ldots x_n.he_1 \ldots e_m \in \mathsf{RCN}(\Gamma_o, \tau, k + 1)$ then there exist an execution of the **else** branch in RCN that produces $\lambda x_1 \ldots x_n.he_1 \ldots e_m$. (This must be the first execution of RCN because it is the only one that can produce terms with length $\mathcal{L}(e)$. All others produce smaller terms.) Also $m \neq 0$ which means that last **foreach** is executed, and that $t_o = \rho_1 \to \cdots \to \rho_m \to v$. It follows that there exist a pattern p and Γ'_o such that $h :$

$\rho_1 \to \cdots \to \rho_m \to v \in$ Select(Γ'_o, p). Note that by the Select definition every element that belongs to Select(Γ'_o, p) also belongs to Γ'_o, i.e., $h : \rho_1 \to \cdots \to \rho_m \to v \in \Gamma'_o$. We know that there exists Γ_o such that $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, and it will be exactly an environment that is passed as an argument to $\mathsf{RCN}(\Gamma_o, \tau, k+1)$. Types τ_j, $j = [1..n]$ are the corresponding argument types that appear in the right order in τ.From the inductive hypothesis and lemma 3.7 it follows that $\Gamma'_o \vdash_\lambda e_j : \rho_j$, where $j = [1..m]$. Thus we can first apply the (lambda) ABS and then the (lambda) APP rule and get $\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_n.he_1 \ldots e_m : \rho_1 \to \cdots \to \rho_n \to v$, i.e., $\Gamma_o \vdash_\lambda e : \tau$.

LEMMA 3.5. *Let Γ_o and τ be a lambda environment, a lambda type respectively. Also let $d \geq 1$ be a number, then:*

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

where $Terms_d$ is the set that contains only terms with length d.

Proof We prove this by induction on d.

Base case: When $d = 1$ we have to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1) = \mathsf{RCN}(\Gamma_o, \tau, 0) \uplus Terms_1$. Because $\mathsf{RCN}(\Gamma_o, \tau, 0)$ is empty set by the RCN definition we have to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1) = Terms_1$. In other words, we need to prove that $\mathsf{RCN}(\Gamma_o, \tau, 1)$ produces only terms with length one. This is **true** because the condition $\forall i \in [1..m]$. $T_i \neq \emptyset$ in $recon$ is always **false**, due to $T_i = \mathsf{RCN}(\Gamma_o, \rho_i, 0) = \emptyset$. Thus $\mathsf{RCN}(\Gamma_o, \tau, 1)$ may only contain terms of the form $\lambda x_1 \ldots x_n.a$ whose length is 1.

Case $d = k$: The following holds $\mathsf{RCN}(\Gamma_o, \tau, k) = \mathsf{RCN}(\Gamma_o, \tau, k-1) \uplus Terms_k$.

Case $d = k+1$: To prove that $\mathsf{RCN}(\Gamma_o, \tau, k+1) = \mathsf{RCN}(\Gamma_o, \tau, k) \uplus Terms_{k+1}$ we use the induction hypothesis to substitute $\mathsf{RCN}(\Gamma'_o, \rho_i, k)$ with $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1) \uplus Terms_k^{(i)}$. Thus, T_i sets become $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1) \uplus Terms_k^{(i)}$. Then from $\mathsf{RCN}(\Gamma_o, \tau, k+1)$ function it follows that we can choose each e_i either from $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$ or from $Terms_k^{(i)}$ (the sets are disjoint). Therefore, the Cartesian product $(T_1 \times \cdots \times T_m)$ becomes the union of 2^m Cartesian products of the form $(T_1^{(j)} \times \cdots \times T_{m'}^{(j)})$, $j = [1..2^m]$. $T_i^{(j)}$ is either $\mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$ or $Terms_k^{(i)}$. Therefore we can split $\mathsf{RCN}(\Gamma_o, \tau, k+1)$ into $2^m + 1$ subsets. We produced 2^m subsets using function with the same code as the RCN, except that $T_i^{(j)}$ replaces T_i. In these functions, an empty tuple does not trigger an execution of the body of the last **foreach** statement. Let us denote new functions with $\mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. The last set contains all terms generated by the empty tuple, and we denote it by RCN_0. Then $\mathsf{RCN}(\Gamma_o, \tau, k+1) := \mathsf{RCN}_0 \uplus \biguplus_{j=[1..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. If $\mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(1)}$ is the function where $T_i^{(j)} = \mathsf{RCN}(\Gamma'_o, \rho_i, k-1)$, for $i = [1..n]$, then it holds $\mathsf{RCN}_0 \uplus \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(1)} = \mathsf{RCN}(\Gamma_o, \tau, k)$. In other functions at least one $T_i^{(j)} = Terms_k^{(i)}$. Such a function synthesizes only terms with length $k+1$, because at least one sub-term e_i has length k. This is the maximal length of the sub-terms as well. By the definition of $\mathcal{L}$ we have that $\mathcal{L}(\lambda x_1 \ldots x_n.fe_1 \ldots e_n) = 1 + max(\mathcal{L}(e_1) \ldots \mathcal{L}(e_n)) = 1 + k$. Therefore, it holds that all terms in those functions have length $k+1$. We denote the set of all such terms by $Terms_{k+1} = \biguplus_{j=[2..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)}$. Finally, we can conclude that $\mathsf{RCN}(\Gamma_o, \tau, k+1) = \mathsf{RCN}_0 \uplus \biguplus_{j=[1..2^m]} \mathsf{RCN}'(\Gamma_o, \tau, k+1)^{(j)} = \mathsf{RCN}(\Gamma_o, \tau, k) \uplus Terms_{k+1}$.

LEMMA 3.6. *Let Γ_o and τ be a lambda environment and a lambda type respectively. Also let d and m be numbers, s.t. $d \geq 1$ and $d \geq m > 0$. Next, let $Terms_{d-m+1,d}$ be a set that contains terms with length from $d-m+1$ to d then:*

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

Proof By applying lemma 3.5 m times to $\mathsf{RCN}(\Gamma_o, \tau, d)$.

LEMMA 3.7. *If $\Gamma_o \vdash_\lambda e : \tau$ is a judgment in long normal form and $d \geq \mathcal{L}(e)$ then the following holds:*

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

Proof If $e \in \mathsf{RCN}(\Gamma_o, \tau, \mathcal{L}(e))$ then $e \in \mathsf{RCN}(\Gamma_o, \tau, d)$ because the lemma 3.6 states that $\mathsf{RCN}(\Gamma_o, \tau, \mathcal{L}(e))$ is a subset of $\mathsf{RCN}(\Gamma_o, \tau, d)$. On the other side if $e \in \mathsf{RCN}(\Gamma_o, \tau, d)$ then either $e \in \mathsf{RCN}(\Gamma_o, \tau, \mathcal{L}(e))$ or $e \in Terms_{\mathcal{L}(e)+1,d}$ by the same lemma. However, by the definition $Terms_{\mathcal{L}(e)+1,d}$ contains only terms with length greater then $\mathcal{L}(e) + 1$. Thus, e cannot be in this set and must be in $\mathsf{RCN}(\Gamma_o, \tau, \mathcal{L}(e))$.

4. Synthesis of All Terms in Long Normal Form

We next present an algorithm based on the succinct ground calculus that we use for finding type inhabitants. This algorithm is further used as an interactive tool for synthesizing expression suggestions from which 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.

In Figure 5 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 6. 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)
  Recns−n(Derived, $\Gamma_o$, $t_o$, N)
```

Figure 5. The algorithm that generates all terms with a given type τ_o and the environment Γ_o

4.1 Pattern Synthesis

In Figure 6 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.

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 6 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 A(t) is empty, or all types in 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.

4.2 Term Synthesis

The function Rcnst-n starts from the desired type and applies the following procedure:

1. Uses a type τ and corresponding patterns to create partial expressions.
2. The bound variables and head variable are instantiated, while arguments are left to be holes.
3. The partial expressions are put into a priority queue based on weights of the expressions.

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

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
```

Figure 6. The algorithm that generates all succinct expressions (patterns) with a given type $S_i \to t_i$ and the environment Γ_i

4. It removes the expression with the smallest weight from the queue.
5. If the expression is fully unfolded, returns it as a solution, and terminates if the number of found solutions is N.
6. If the expression is not fully unfolded, uses a greedy algorithm to find the expression's hole with potentially the smallest weight.
7. Finds the type of the hole τ and goes to the step 1.

5. Subtyping using Coercion Functions

We use a simple method of coercion functions [3, 12, 16] 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 $c_{12} : \{v_1\} \rightarrow v_2$. 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. 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 $\rho_1 \rightarrow \rho_2 <: \tau_1 \rightarrow \tau_2$). 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. If subtyping constraints are present, then the coercion functions are used in construction of succinct patterns. However, in the RCN function the coercion functions are omitted when deriving new lambda terms.

6. Quantitative Type Inhabitation Problem

When answering the type inhabitation problem, there might be many terms having the required type τ. A question that naturally arises is how to find the "best" term, for some 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.

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 6 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 1 we assign the least weight to local symbols declared in the same method. We assign the weight 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 7.3 below. For an imported symbol x, we determine its weight using the formula in Table 1. 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.

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

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

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

To guide the algorithm that generates patterns (in Figure 6) we use weights of succinct terms. Given $Select$ in Figure 4, a weight of a succinct type t in Γ_o is defined as:

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

7. Evaluation of the Effectiveness of InSynth

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 derive type constraints.
2. Follow the standard typing rules to derive new type declarations while directing the search towards the inhabitants of the required type.
3. Rank the found type inhabitants
4. Create the code snippets from corresponding found terms and output them to the user.

7.1 Implementation in Eclipse

We implemented InSynth as an Eclipse plugin that extends the Eclipse code completion feature for automatic generation of well-typed Scala expressions. 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 unnecessary parentheses omitting, using method name shorthands, etc.). InSynth is available for download and is currently maintained as a part of the Eclipse Scala IDE plugin.

7.2 Creating Benchmarks

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

One idea of measuring the effectiveness of a synthesis tool is to estimate its ability to reconstruct certain expressions from existing code. We arbitrarily chose some expressions from the collected examples, removed them and marked them as 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 [18]. 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.

7.3 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 6. 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. Table 2 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. 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, 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 `&&`).

7.4 Platform for Experiments

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

7.5 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 3. 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 7.3) 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`[14] and `fCube`[7]) employed for checking provability of inhabitation problems for the benchmarks, encoded as formulas in appropriate syntax.

Table 3 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.

				No weights		No corpus		All				Provers	
Benchmarks		Size	#Initial	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 3. 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.

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 7.3) 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.

8. Related Work

Several tools including Prospector [13], XSnippet [17], Strathcona [9], PARSEWeb [23] and SNIFF [4] 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. InSynth is similar in operation to Eclipse content assist proposals [22] and it implements the same behaviour. More advanced solutions appeared recently, like the Eclipse code recommenders [6], 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.

Prospector [13] 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 [17] 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 [9], 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 [23] 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 [4] tool is to use natural language to search for code examples. The authors collected the corpus of examples and annotated them with keywords, and attached them to corresponding method calls in the examples. The keywords are collected from the available API documentation. InSynth is based on a logical formalism, so it can overcome the gap between programming languages and natural language.

The synthesized code in our approach is extracted from the proof derivation. Similar ideas have been exploited in the context of sophisticated dependently typed languages and proof assistants [2]. Our goal is to apply it to simpler scenarios, where propositions are only partial specifications of the code, as in the current programming practice. Agda is a dependently typed programming language and proof assistant. Using Agda's Emacs interface, programs can be developed incrementally, leaving parts of the program unfinished. By type checking the unfinished program, the programmer can get useful information on how to fill in the missing parts. The Emacs interface also provides syntax highlighting and code navigation facilities. However, because it is a new language and lacks large examples, it is difficult to evaluate this functionality on larger numbers of declarations.

There are several tools for the Haskell API search. The Hoogle [10] 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 [8] search engine does not use types for searching functions: its search is based on function names. The main difference between Djinn [21] 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 [15]. 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.

The use of type constraints was explored in interactive theorem provers, as well as in synthesis of code fragments. SearchIsos [5] 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 [25], 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.

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 [14] and fCube [7]. 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.

9. Conclusions

We have 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.

References

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

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

[3] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93:172–221, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90055-7.

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

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

[6] Eclipse Code Recommenders. `http://www.eclipse.org/recommenders/`.

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

[8] Hayoo! API Search. `http://holumbus.fh-wedel.de/hayoo/hayoo.html`.

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

[10] Hoogle API Search. `http://www.haskell.org/hoogle/`.

[11] IntelliJ IDEA website, 2011. URL `http://www.jetbrains.com/idea/`.

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

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.

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

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

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

[17] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA*, 2006. ISBN 1-59593-348-4.

[18] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, 2007.

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

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

[21] The Djinn Theorem Prover. `http://www.augustsson.net/Darcs/Djinn/`.

[22] The Eclipse Foundation. `http://www.eclipse.org/`.

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

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

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