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 (MPI-SWS), Germany
piskac@mpi-sws.org

Abstract

Developing modern software 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 inhabitation 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 code examples from programming practice; we make them available for future comparisons.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming–Program synthesis; D.2.6 [*Software Engineering*]: Coding Tools and Techniques–Program Editors; D.2.13 [*Software Engineering*]: Reusable Software–Reuse Models

General Terms Languages, Algorithms

Keywords program synthesis, type inhabitation, code completion, type-driven synthesis, ranking

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 [26] and IntelliJ [15] recommend methods applicable to an object, and allow the developer to fill in additional method arguments. Such completion typically considers one step of computation. IntelliJ can additionally compose simple method sequences to form a type-correct expression, but requires both the receiver object as well as assistance from the developer to fill in the arguments. These efforts suggest a general direction for improving modern IDEs: introduce the ability to synthesize entire type-correct code fragments and offer them as suggestions to the developer.

In this paper we describe a tool for automated synthesis of code snippets. The tool generates and suggests a list of expressions that have a desired type. 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 object) 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 pioneering work on the Prospector tool [18], or than the recent work of Perelman et al. [20]. A general idea of our approach and a first prototype implementation was demonstrated already in [10].

Finding a code snippet of the given type 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. Formally, find e such that $\Gamma \vdash e : T$. In our deployment, the tool computes Γ from the position of the cursor in the editor buffer. It similarly looks 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. When there are multiple solutions, the tool prompts the developer to select one, much like in simpler code completion scenarios.

The type inhabitation in the simply typed lambda calculus corresponds to provability in propositional intuitionistic logic; it is decidable and PSPACE-complete [24, 28]. We developed a version of the algorithm that is complete in the lambda calculus sense (up to $\alpha\beta\eta$-conversion): it is guaranteed to synthesize a lambda expression of the given type, if such an expression exists. Moreover, if there are multiple solutions, it can enumerate all of them. If there are infinitely many solutions, then the algorithm can enumerate any desired finite prefix of the list of all solutions. Note also that each synthesized expression is a complete in that method calls have all of their arguments synthesized. Because of all these aspects of the algorithm we describe our technique as *complete completion*.

We present our algorithm using a calculus of *succinct types*, 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.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

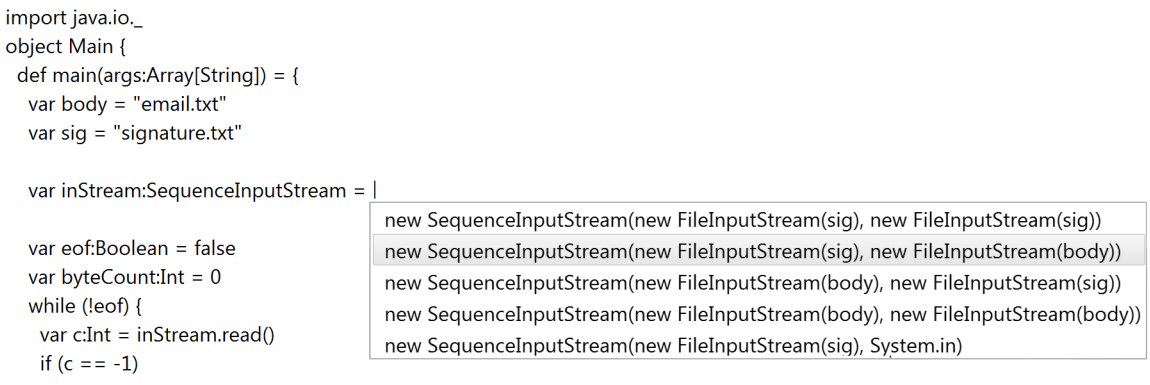

Figure 1. InSynth suggesting five highest-ranked well-typed expressions synthesized from declarations visible at a given program point

Moreover, our algorithm generates a representation of *all* solutions, from which it can then extract any desired finite subset of solutions.

Given a possibility of an infinite number of type inhabitants, it is natural to consider the problem of finding the *best* one. To solve this problem, we introduce *weights* to guide the search and rank the presented solutions. Initially we assign the weight to each type declaration. Those weights play a crucial role in the algorithm, since they guide the search and rank the presented solutions. The weight is defined in a way that a smaller weight indicates a more desirable formula. 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. In addition, 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 implemented our tool, InSynth, within the Scala Eclipse plugin. 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. 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. The running times of InSynth were always a fraction of a second. In the great majority of cases we found that the expected snippets were returned among the top dozen solutions.

Furthermore, we evaluated a number of techniques deployed in our final tool and found that all of them are important for obtaining good results. We also observed that, even for checking existence of terms InSynth outperforms recent propositional intuitionistic provers [9, 19] on our benchmarks. Our overall experience suggests that InSynth is effective in providing help to developers.

2. Motivating Examples

We illustrate the functionality of InSynth through three examples. The first example is taken from the online repository of Java API usage samples `http://www.java2s.com/`. The second example is a real-world fragment of the code base of the Scala IDE for Eclipse, `http://scala-ide.org/`, and requires invoking a higher-order function. For these two examples, the original code imports only declarations from a few specific classes; to make the problems more challenging and illustrate the task that a programmer faces, we import all declarations from packages where those classes reside. The third example illustrates that InSynth supports subtyping.

2.1 Sequence of Streams

In this example the goal is to create a SequenceInputStream object, which is a concatenation of two streams. Suppose that the developer has the code shown in the Eclipse editor in Figure 1. If she invokes InSynth at the program point indicated by the cursor, in a fraction of a second InSynth displays the ranked list of five expressions. Seeing the list, the developer can decide that e.g. the second expression in the list matches their 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 developer. InSynth is able to use both imported values (such as the constructors in this example) and locally declared ones (such as body and sig). InSynth supports methods with multiple arguments and synthesizes expressions for each argument. In this example InSynth loads over 3000 declarations from the context, including local and imported values, and finds the expected solution in less than 250 milliseconds.

The effectiveness of InSynth is characterized by both scalability to many declarations and the quality of the returned suggestions. InSynth ranks the resulting expressions according to the weights and selects the ones with the lowest weight. The weights of expressions and types guide the final ranking and also make the search itself more goal-directed and effective. InSynth derives weights from a corpus of declarations, assigning lower weight to declarations appearing more frequently, and therefore favoring their appearance in the suggested fragments over more exotic declarations.

2.2 TreeFilter: Using Higher-Order Functions

We demonstrate the generation of expressions with higher-order functions on real code from the Scala IDE project. The example shows how a developer should properly check if a Scala AST tree satisfies a given property. In the code, the tree is an argument of the class TreeWrapper, whereas the 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 p is a predicate function that takes the tree and returns **true** if the tree satisfies it. In order to properly use p inside filter, the developer first needs to create an object of the type FilterTypeTreeTraverser. If the developer calls InSynth at the place ▮, the tool offers several expressions, and the one ranked first turns out to be precisely 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.

2.3 Drawing Layout: Using Subtyping

The next example illustrates a situation often encountered when using java.awt: implementing a getter method that returns a layout of an object Panel stored in a class Drawing. To implement such a method, we use code of the following form.

```
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.

For more experience with InSynth, we encourage the reader to download it from:

`http://lara.epfl.ch/w/insynth`

The rest of the paper describes a formalization of the problem that InSynth solves as well as the algorithms we designed to solve it. We then describe the implementation and the evaluation, provide a survey of related efforts, and conclude.

3. Type Inhabitation Problem for Succinct Types

To answer whether there exists 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 the synthesis of code snippets.

Let T be a set of types. A *type environment* Γ is a finite set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of pairs of the form $x_i : \tau_i$, where x_i is a variable of a type $\tau_i \in T$. We call the pair $x_i : \tau_i$ a *type declaration*.

The type judgment, denoted by $\Gamma \vdash e : \tau$, states that from the environment Γ, we can derive the type declaration $e : \tau$ by applying rules of some calculus. The type inhabitation problem for a given calculus is defined as follows: given 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 type rules for the standard lambda calculus restricted to normal-form terms. We denote the corresponding type judgment relation $\vdash_\lambda$. We then introduce a new *succinct* representation of types and terms, with the corresponding type judgment relation $\vdash_c$.

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

As background we present relevant rules for the simply typed lambda calculus, focusing on 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 $\tau_\lambda(B)$.

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

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

Figure 2 shows the type derivation rules used to derive terms in long normal form. This calculus is slightly more restrictive than

$$\textsc{App}\ \frac{(f : \tau_1 \to \ldots \to \tau_n \to \tau) \in \Gamma_o \qquad \Gamma_o \vdash_\lambda e_i : \tau_i,\ i = 1..n \qquad \tau \in B}{\Gamma_o \vdash_\lambda f e_1 \ldots e_n : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma_o \cup \{x_1 : \tau_1, \ldots, x_m : \tau_n\} \vdash_\lambda e : \tau \qquad \tau \in B}{\Gamma_o \vdash_\lambda \lambda x_1 \ldots x_m.e : \tau_1 \to \ldots \to \tau_m \to \tau}$$

Figure 2. Rules for deriving lambda terms in long normal form

the standard lambda calculus: the APP rule requires that only those functions present in the original environment Γ_o can be applied on terms.

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

- $e \equiv \lambda x_1 \ldots x_m.f e_1 ... e_n$, *where* $m, n \geq 0$
- $(f : \rho_1 \to \ldots \to \rho_n \to \tau) \in \Gamma_o$, *where* $\tau \in B$
- $\tau_e \equiv \tau_1 \to \ldots \to \tau_m \to \tau$
- $\Gamma'_o \vdash_\lambda e_i : \rho_i$ *are in long normal form, where* $\Gamma'_o = \Gamma_o \cup \{x_1 : \tau_1, \ldots, x_m : \tau_m\}$

Note that m can be zero. Then, $\tau_e \equiv \tau$ and Definition 3.1 reduces to the App rule. Otherwise, if $M \equiv f e_1 ... e_n$, then $M : \tau$ can be derived by App and $\lambda x_1 \ldots x_m.M : \tau_e$ by Abs rule.

In long normal form a variable f is followed by exactly the same number of sub-terms as the number of arguments indicated by the type of f. As an illustration, consider $f : \tau_1 \to \tau_2 \to \tau_3$ and $x : \tau_1$. There is no derivation resulting in a judgement $\Gamma_o \vdash_\lambda f x : \tau_2 \to \tau_3$ in long normal form, but $\lambda y.f x y : \tau_2 \to \tau_3$ has a long normal form derivation.

When solving the type inhabitation problem it suffices to derive only terms in long normal form, which restricts the search space. This does not affect the completeness of search, because each simply-typed term can be converted to its long normal form [6].

We define the depth $\mathcal{D}$ of a term from a long normal form judgement as follows:
$\mathcal{D}(\lambda x_1 \ldots x_m.a) = 1$
$\mathcal{D}(\lambda x_1 \ldots x_m.f e_1, \ldots, e_n) = \max(\mathcal{D}(e_1), \ldots, \mathcal{D}(e_n)) + 1$,
where a and f belong to V.

3.2 Succinct Types

To make the search more efficient we introduce *succinct types*, which are types modulo isomorphisms of products and currying, that is, according to the Curry-Howard correspondence, modulo commutativity, associativity, and idempotence of the intuitionistic conjunction.

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

$$t_s ::= \{t_s, \ldots, t_s\} \to v, \quad \text{where } v \in B$$

We denote the set of all succinct types with $t_s(B)$, 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. The type $\emptyset \to t$ plays a special role: it 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 this conversion function. Every basic type $v \in B$ becomes an element of the set of basic succinct types, and $\sigma(v) = \emptyset \to v$. We also denote $\emptyset \to v$ only with v. Let A (arguments) and R (return type) be two functions defined on $t_s(B)$ as follows:

$$\begin{array}{lcl} A(\{t_1, \ldots, t_n\} \to v) & = & \{t_1, \ldots, t_n\} \\ R(\{t_1, \ldots, t_n\} \to v) & = & v \end{array}$$

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

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

In particular, for $v \in B$, a type of the form

$$\tau_1 \to \ldots \to \tau_n \to v$$

which often occurs in practice, has the succinct representation

$$\{\sigma(\tau_1), \ldots, \sigma(\tau_n)\} \to v$$

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

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

It follows immediately that the conversion distributes over unions:

$$\sigma(\bigcup_{i \in I} \Gamma_o^i) = \bigcup_{i \in I} \sigma(\Gamma_o^i)$$

To demonstrate the power of the succinct representation, we provide the statistics from the example in Figure 1. In this example, the original type environment with 3356 declarations is reduced to the compact succinct environment with 1783 succinct types, after the σ transformation. This drastically reduces the search space later explored by our main algorithm.

3.3 Succinct Patterns

Succinct patterns have the following structure:

$$\Gamma @ \{t_1, \ldots, t_n\} : t$$

where $t_i \in t_s(B)$, $i = 1..n$, and $t \in B$.

A pattern $\Gamma @ \{t_1, \ldots, t_n\} : t$ indicates that types $t_1, \ldots, t_n$ are inhabited in Γ and an inhabitant of type t can be computed from them also in Γ. They abstractly represent an application term in lambda calculus.

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 ways in which a term can be computed. In the second phase we do a term reconstruction based on the original type declarations (Γ_o) and the set of succinct patterns.

3.4 Succinct Calculus

Figure 3 describes the calculus for succinct types. Note that the patterns are derived only in the APP rule. The rule ABS modifies Γ – it can either reduce Γ or enlarge it, depending on whether we are doing backward or forward reasoning.

$$\textsc{App}\ \frac{\{t_1, \ldots, t_n\} \to t \in \Gamma \qquad \Gamma \vdash_c t_i,\ i = 1..n \qquad t \in B}{\Gamma \vdash_c\ \Gamma @ \{t_1, \ldots, t_n\} : t}$$

$$\textsc{Abs}\ \frac{\Gamma \cup S \vdash_c (\Gamma \cup S) @ \pi : t \qquad t \in B}{\Gamma \vdash_c S \to t}$$

Figure 3. Calculus rules for deriving succinct patterns

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 $\Gamma = \{\emptyset \to \text{Int}, \{\emptyset \to \text{Int}\} \to \text{String}\} = \{\text{Int}, \{\text{Int}\} \to \text{String}\}$. By applying the APP rule on Int, we derive a succinct pattern $\Gamma @ \emptyset : \text{Int}$ that we add to a set of derived patterns. Having a pattern for Int we apply the ABS rule. By setting $S = \emptyset$, we derive $\Gamma \vdash_c \emptyset \to \text{Int}$. Finally, by applying again the APP rule, we directly derive a pattern $\Gamma @ \{\text{Int}\} : \text{String}$, for the String type and store it into the set of derived patterns.

3.5 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. 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 completeness that captures this ability, we introduce two functions, CL and RCN, shown in Figure 4. These functions describe the terms in long normal form of a desired type, up to a given depth d. They refer directly to $\vdash_c$ and are therefore not meant as algorithms, but as a way of expressing the completeness of succinct representation and as specifications for the algorithms we outline in Section 5.

```
fun CL(Γ, S→t) = {(Γ ∪ S)@S1 : t | (S1→t) ∈ (Γ ∪ S),
                                   ∀t' ∈ S1.Γ ∪ S ⊢c t'}
fun Select(Γo, t) := {v:τ | v:τ ∈ Γo and σ(τ) = t}
fun RCN(Γo, τ1→···→τn→v, d) :=
  if (d = 0) return ∅
  else
    S→v := σ(τ1→···→τn→v)
    Γ := σ(Γo)
    Γ'o := Γo ∪ {x1 : τ1, ..., xn : τn} //x1, ..., xn are fresh
    TERMS := ∅
    foreach ((Γ ∪ S)@{t1, ..., tm'} : v) ∈ CL(Γ, S→v)
      foreach (f : τ) ∈ Select(Γ'o, {t1, ..., tm'}→v)
        (ρ1→···→ρm→v) := τ
        if (m=0) TERMS := TERMS ∪ {λx1 ... xn.f}
        else
          foreach i ← [1..m]
            Ti := RCN(Γ'o, ρi, d−1)
          foreach (e1, ..., em) ← (T1 × ··· × Tm)
            TERMS := TERMS ∪ {λx1 ... xn.f e1 ... em}
    return TERMS
```

Figure 4. The function RCN constructs lambda terms in long normal form up to given depth d, invoking the auxiliary functions CL and Select.

The CL function in Figure 4 takes as arguments a succinct type environment Γ and a succinct type $S \to t$. It returns the set of all patterns $(\Gamma \cup S) @ S_1 : t$ that describe the derivation of t. The function RCN uses the initial environment and the desired type to reconstruct lambda terms. Additionally, RCN takes a non-negative integer d to limit the reconstruction to terms with depth smaller or equal to d. It uses type $\tau_1 \to \cdots \to \tau_n \to v$ to extend the environ-

ment and find all patterns that witness inhabitation of v. We extend the environment with fresh variables $x_1 : \tau_1, \ldots, x_n : \tau_n$, and use CL to find the patterns. Further, we find all declarations f with a return type v in the extended environment. If f has a function type $\rho_1 \to \cdots \to \rho_m \to v$, we recursively generate corresponding sub-terms with types $\rho_1, \ldots, \rho_m$. Finally, we use $x_1, \ldots, x_n$, f and sub-terms to construct terms in long normal form.

Given the functions CL and RCN we can formalize the completeness theorem: each judgement in long normal form derived in the standard lambda calculus can also be derived by reconstruction using derivations (patterns) of the succinct calculus.

THEOREM 3.3 (Soundness and Completeness). *Let Γ_o be an original environment, e an lambda expression, $\tau \in \tau_\lambda(B)$ and functions* RCN *and $\mathcal{D}$ defined as above, then:*

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

We provide the proof of Theorem 3.3 in [11].

4. Quantitative Type Inhabitation Problem

When answering the question of 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 adequate meaning of "best". For this purpose we assign a weight to every term. 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.

Nature of Declaration or Literal	Weight
Lambda	1
Local	5
Coercion	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.

Let w be a function that assigns a weight (a non-negative number) to each symbol 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 lowest weight to local symbols declared in the same method. We assign a higher weight to symbols defined in the class where the query is initiated and the highest weight to symbols that are only in the same package.
2. the frequency with which the symbol appears in the training data corpus, as described in Section 7.3. 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.

We also assign a low weight to a conversion function that witnesses the subtyping relation, as explained in Section 6. 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.

The function w also assigns a weight to a term such that the weight of the term $\lambda x_1 \ldots x_m . f e_1 \ldots e_n$ is the sum of the weights of all elements that occur in the expression:

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

We use the weight of succinct types to guide the algorithm in Figure 5. Given Select in Figure 4, the weight of a succinct type t in Γ_o is defined as:

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

5. Synthesis of All Terms in Long Normal Form

In this section we first motivate and introduce the backward search as the core mechanism of the algorithm, then we illustrate the algorithm and optimizations we implement in InSynth.

5.1 Backward Search

If we were to apply the rules in Figure 3 in a forward manner we could have started from any environment in the premise(s). However, there are infinitely many such environments. Moreover, rule Abs states that we can split Γ' in $3^{|\Gamma'|}$ possible ways into two subsets S and Γ, such that $\Gamma' = \Gamma \cup S$. However, only some environments and splittings will lead to the final conclusion $\Gamma_{init} \vdash_c S_{init} \to t_{init}$, where Γ_{init} and $S_{init} \to t_{init}$ are the initial environment and the desired type, respectively. This means we would have many unnecessary guesses and computations, leading to the wrong conclusions.

In contrast, if we use a backward search, then we start from the conclusion $\Gamma_{init} \vdash_c S_{init} \to t_{init}$ in Abs rule, and use a premise to create a hypothesis that a pattern $(\Gamma_{init} \cup S_{init})@\pi : t_{init}$ is derivable. Further, we need to check that it is indeed derivable by applying App rule. Now, unlike in the forward manner, thanks to the conclusion in App, we know the exact environment and the type t of the first premise. Selecting only types in Γ that have return types t introduce constraints on the other premises as well. The entire process is applied recursively until the initial hypothesis is proven or disproved. However, the constraints allow us to narrow the search. This is the main advantage of the backward search. All this suggest that the backward search is more efficient, revealing only the search space reachable from the initial environment and the desired type, unlike the forward search.

To formalize the backward search we reformulate the earlier rules by splitting them into the five new rules shown in Figures 6 and 8. One should read and apply the new rules in the forward manner. In the following subsections we explain those rules in more detail.

5.2 Main Algorithm

In this section we 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 a developer can select a suitable expression. 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.

The algorithm is illustrated in Figure 5. As input Synthesize takes a desired type τ_o, and an environment Γ_o and outputs at most N terms in long normal form with a type τ_o. We first transform Γ_o and τ_o by σ into a succinct environment and a type, respectively. Then we execute the algorithm in three phases. First, Explore takes the succinct type and the environment as input, and returns the

discovered search space reachable from the desired type and the initial environment. Next, GenerateP takes the space as input and outputs a set of patterns. Finally, GenerateT takes patterns, Γ_o, τ_o and the integer N, and produces at most N ranked terms.

The Explore and GenerateP use succinct types to prune the search space in a light way. They leave only a portion where declaration argument-return types conform. This helps GenerateT to perform heavy reconstruct only when needed, returning compilable terms.

```
fun Synthesize(Γo, τo, N):= {
  space := Explore(σ(Γo), σ(τo))
  patterns := GenerateP(space)
  return GenerateT(patterns, Γo, τo, N)
}
```

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

The algorithm Synthesize represents the imperative description of RCN, in Figure 4. It is the RCN version with a bound on the number of terms, N. Moreover, it uses weights to steer the search towards useful terms. We discuss this at the end of the section. The backward search and search driven by weights make the new algorithm effective, practical and interactive. Synthesize produces the same set of solutions as RCN, given the same input, if we remove bounds d and N or gradually increase them. Note that the set of solutions might be infinite.

5.3 Exploration phase

The goal of Explore is to start from the desired succinct type and environment and gradually explore the search space. We split the algorithm into three key steps:

1. **Type reachability.** Given a succinct type t and Γ we want to find all types reachable from t. We specify this request by $t \underset{\Gamma}{\rightsquigarrow}?$. We use the request to trigger the Match rule in Figure 3. By the rule, types in set S are reachable from type t, $A(t) = \emptyset$, in Γ, if $(S{\to}t) \in \Gamma$ holds. We denote this with *reachability* term, $t \underset{\Gamma}{\rightsquigarrow} (S, \Pi)$ (later we explain what the set Π is).
2. **Request propagation.** Once we discover that types S are reachable from t, we want to discover what types are reachable from any type $t' \in S$. Thus, we generate a new request $t' \underset{\Gamma}{\rightsquigarrow}?$. New requests are issued with the Prop rule. In other words, we use the Prop rule to propagate the search.
3. **Environment extension.** However, t can be a function type, i.e., $t \equiv S' \to t'$ and $S' \neq \emptyset$. Thus, we introduce the Strip rule that transforms a request $(S \to t) \underset{\Gamma}{\rightsquigarrow}?$ to a request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$. Now, we can further use the request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$ in the Match rule.

$$\textsc{Match}\ \frac{t \underset{\Gamma}{\rightsquigarrow}? \qquad (S{\to}t) \in \Gamma \qquad A(t) = \emptyset}{t \underset{\Gamma}{\rightsquigarrow} (S, \emptyset)}$$

$$\textsc{Prop}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (S, \emptyset) \qquad t' \in S}{t' \underset{\Gamma}{\rightsquigarrow}?} \qquad \textsc{Strip}\ \frac{(S \to t) \underset{\Gamma}{\rightsquigarrow}?}{t \underset{\Gamma \cup S}{\rightsquigarrow}?}$$

Figure 6. Type reachability rules.

A set of reachability terms keep the information about the explored search space. Thus our goal is to derive all such terms starting from the desired type and the environment. We next give the detailed description of the Explore algorithm.

```
fun Explore(Γ, S→t) := {
  queue := {S→t ⇝Γ ?}
  visited := ∅
  space := ∅
  while(queue ≠ ∅) {
    curr := queue.dequeue
    visited := visited ∪ {curr}
    t' ⇝Γ' ? := Strip(curr)
    found := {Match(t' ⇝Γ' ?, S'→t') | S'→t' ∈ Γ'}
    space := space ∪ found
    newr := {Prop(tf ⇝Γf (Sf,∅), t') | tf ⇝Γf (Sf,∅) ∈ found
                                       and t' ∈ Sf}
    queue := queue ∪ (newr \ visited)
  }
  return space
}
```

Figure 7. The algorithm that explores the search space.

To initiate the Explore algorithm in Figure 7 we create the request $(S{\to}t) \underset{\Gamma}{\rightsquigarrow}?$, where $S{\to}t$ is the desired type, and Γ is the initial type environment. We put the request into a working queue. In the loop we process one request at the time, until queue is empty. First, we use Strip to obtain a new request $t' \underset{\Gamma'}{\rightsquigarrow}?$ with an extended environment Γ'. Here, Strip is the function that implements the corresponding rule, Strip in Figure 3. It takes request $(S{\to}t) \underset{\Gamma}{\rightsquigarrow}?$ and returns new request $t \underset{\Gamma \cup S}{\rightsquigarrow}?$. In the same fashion, we implement the other two functions, Match that returns a reachability term, and Prop that returns a request. Next, by applying Match to $t' \underset{\Gamma'}{\rightsquigarrow}?$ and every type in Γ' we find a set of reachability terms, found. We store the terms in the set space. The space set represents the entire search space discovered from the desired type and the initial environment. Finally, we propagate the search by issuing newr requests using the Prop function onto found. We update queue with these requests. We additionally keep the set of all visited requests, to avoid cycles in the exploration.

5.4 Pattern generation phase

In this phase we use the space explored by the Explore algorithm to create patterns. We start from the reachability terms with inhabited types, and use them to produce patterns and new inhabited types. We repeat the process until no new types can be inhabited.

$$\textsc{Prod}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)}{\Gamma@\Pi : t}$$

$$\textsc{Transfer}\ \frac{t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S' \to t'\}, \Pi) \qquad t' \underset{\Gamma \cup S'}{\rightsquigarrow} (\emptyset, \Pi')}{t \underset{\Gamma}{\rightsquigarrow} (S, \Pi \cup \{S' \to t'\})}$$

Figure 8. Pattern synthesis rules.

Initially, we divide the search space, space, into two groups: 1) leaves that contains reachability terms in the form $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$, i.e., reachability terms with inhabited types, and 2) others that contains the remaining terms. The set Π collects succinct types that have an inhabitant. It is initialized by Match to an empty set. The Transfer rule, in Figure 8, turns $t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S'{\to}t'\}, \Pi)$ into

```
fun GenerateP(space) := {
  patterns := ∅
  visited := ∅
  leaves := {x | x = t ⇝_Γ (∅,∅) and x ∈ space}
  others := space \ leaves
  while (leaves ≠ ∅) {
    t ⇝_Γ (∅,Π) := leaves.dequeue
    visited := visited ∪ {t ⇝_Γ (∅,Π)}
    patterns := patterns ∪ {Prod(t ⇝_Γ (∅,Π))}
    compatible := {x | x = t' ⇝_Γ' (S ∪ {S' → t}, Π')
                       and Γ = Γ' ∪ S' and x ∈ others}
    newt := {Transfer(x, t ⇝_Γ (∅,Π)) | x ∈ compatible}
    newLeaves := {x | x = t' ⇝_Γ' (∅,Π') and x ∈ newt}
    others := (others \ compatible) ∪ (newt \ newLeaves)
    leaves := leaves ∪ (newLeaves \ visited)
  }
  return patterns
}
```

Figure 9. The algorithm that generates patterns.

$t \underset{\Gamma}{\rightsquigarrow} (S, \Pi \cup \{S' \to t'\})$ if $\{S' \to t'\}$ is inhabited. We use Π to produce a pattern by the Prod rule.

In the loop we remove one term $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$ from leaves to generate: 1) a pattern $\Gamma@\Pi : t$ by the Prod function, and 2) the newt reachability terms by the Transfer function. To perform the latter we first calculate the set of compatible terms. Those are the reachability terms in form $t \underset{\Gamma}{\rightsquigarrow} (S \cup \{S' \to t'\}, \Pi)$, such that $\Gamma = \Gamma' \cup S'$ holds. Every term in compatible can be resolved with $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$ by the Transfer rule. The result is the set of newt terms. These reachability terms can be split into two groups. The first group contains terms of the form $t' \underset{\Gamma'}{\rightsquigarrow} (\emptyset, \Pi')$, that we add to leaves. The second group contains the remaining terms and we add them to others. We also keep the set of visited leaves in order to avoid cycles in generation.

5.5 Term generation phase

In Figure 10 we illustrate the algorithm that finds at most N lambda expressions with the smallest weight. First, we introduce the notion of holes to define the partial expressions, and later we describe the algorithm GenerateT.

A typed hole $[\ ]_h : \tau$ is a constant $[\ ]$ with a name h and a type τ. Let V be a set of typed variables, and H a set of typed holes. Partial typed expressions are constructed according to the following syntax:

$$e ::= x \mid [\ ]_h : \tau \mid \lambda x{:}\tau.e \mid e\,e, \quad \text{where } x \in V \text{ and } [\ ]_h : \tau \in H$$

To derive the partial expressions in *long normal form* one can use the same APP and ABS rules in Figure 2, where $e_i, i = [1..m]$ and e are partial types expressions. Moreover, one can substitute all holes in a partial expression and get a new partial or complete expression, without holes. A hole $[\ ]_h : \tau$, in a judgment $\Gamma_o \vdash [\ ]_h : \tau$, can be substituted only with a partial expression $e : \tau$, where $\Gamma_o \vdash e : \tau$.

We next describe the GenerateT algorithm. We start from the desired type and original environment, follow patterns and gradually create and unfold partial expressions. During the process we keep partial expressions in the queue. Once a partial expression becomes complete, we store it in the set of snippets.

We use a priority queue to process partial expressions. The expressions are sorted by the weight in ascending order. We initiate the queue with $[\ ]_x : \tau_{init}$, where τ_{init} is the desired type. In the loop we process one partial expression at a time. The loop terminates either when the queue is empty or we find N expression. First, we remove the highest ranked partial expression, expr_p, from the priority queue. Then, we call the function findFirstHole, that for a given judgment $\Gamma_{init} \vdash \text{expr}_p$ finds (if it exists) a hole $[\ ]_h : \tau_1 \to \cdots \to \tau_n \to v$ and its corresponding environment Γ_o. If expr_p has no holes, it is a complete lambda expression, i.e., a snippet that we will output to a user. Hence, we append it to snippets. If there is a hole in expr_p we build all partial expressions that can substitute the hole. We extend the environment and use patterns with the return type v to find declarations f. If the declaration has function type, we build the expression filling all arguments with fresh holes. (Note that the new holes might be substituted in a later iteration.) For each expression expr_{newp}, we build a substitution that maps a name of the hole to the expressions. We apply the substitution to substitute the hole with the new expression. We use the function w to calculate expression weights and store them in the priority queue. The weight of a hole is equal to zero. We find

```
fun GenerateT(patterns, Γ_init, τ_init, N) := {
  snippets := NIL
  pq := PriorityQueue.empty
  pq.put(0, [ ]_x : τ_init)
  while(pq.size > 0 and |snippets| < N){
    expr_p = pq.dequeue
    findFirstHole(Γ_init,expr_p) match {
      case None ⇒
        snippets.append(expr_p) //appends to the end
      case Some((Γ_o, [ ]_h : τ1→···→τn→v)) ⇒
        S→v := σ(τ1→···→τn→v)
        Γ := σ(Γ_o)
        //x1,...,xn are fresh
        Γ'_o := Γ_o ∪ {x1 : τ1,...,xn : τn}
        foreach ((Γ ∪ S)@S' : v) ∈ patterns //S' is binder
          foreach (f : ρ1→···→ρm→v) ∈ Select(Γ'_o, S'→v)
            expr_newp :=
              sub(expr_p, h ↣ (λx1...xn.f[ ]_r1 : ρ1 ... [ ]_rm : ρm))
            //r1,...,rm are fresh names
            pq.put(w(expr_newp), expr_newp)
    }
  }
  return snippets
}

fun findFirstHole(Γ_o, exp):= exp match {
   case [ ]_x : τ ⇒ Some((Γ_o, [ ]_x : τ))
   case λx1...xn.fe1...em ⇒
     Γ'_o := Γ_o ∪ {x1 : τ1,...,xn : τn} //x1,...,xn are fresh
     for(i ∈ [1..m])
       findFirstHole(Γ'_o, e_i) match {
         case Some(hole) ⇒ return Some(hole)
         case Node ⇒
       }
     None
}

fun sub(expr1, y ↣ expr2):= expr1 match {
  case [ ]_x : τ ⇒ if (x = y) expr2 else expr1
  case λx1...xn.fe1...em ⇒
    λx1...xn.f sub(e1, y ↣ exp2) ... sub(em, y ↣ exp2)
}
```

Figure 10. A function that constructs the best N lambda terms in long normal form.

the partial expressions that replace the hole using patterns. First we calculate a succinct type $S \to v$ and environment Γ. We expand the environment to $\Gamma \cup S$ and use it with type v to find all pat-

terns with form $(\Gamma \cup S)@S' : v$ in the pattern set. When we find all such sets S' we use them to select all type variables in Γ'_o whose type maps to a succinct type $S' \to v$. Once we have such a variable we use it to create the most general partial expression $\lambda x_1 \ldots x_n.f[\ ]_{r_1} : \rho_1 \ldots [\ ]_{r_m} : \rho_m$. Such an expression has holes at the places of f's arguments. In this way we gradually unfold a partial expression until it becomes complete.

5.6 Responsiveness

We use first two phases to synthesize patterns starting from the desired type and the initial environment. We referred to those phases as a $prover$. To be interactive we allow a user to specify a time limit for the prover. Due to time bound, we decide to interleave the two phases, such that whenever Explore discovers a new leaf, it immediately triggers GenerateP. Every time GenerateP is called it uses all discovered reachability terms to generate as many new patterns as possible. Moreover, to generate the best solutions, within a given time, we use a priority queue in Explore instead of the regular queue. Requests in the priority queue are sorted by weights. A weight of a request $t \underset{\Gamma}{\rightsquigarrow} ?$ is equal to a weight of type t in the initial environment. Additionally, we allow a user to specify a time limit for GenerateT.

5.7 Optimizations

To efficiently find the compatible set in GenerateP, we create a backward map that maps a term to its predecessor terms. Last reachability term_f that initiated creation of term, through propagation, is the predecessor of term. We build the map in Explore, that records all predecessors of a given term, by storing an entry (term, predecessors). By using the map, compatible becomes the predecessors set of $t \underset{\Gamma}{\rightsquigarrow} (\emptyset, \Pi)$. This way we do not preform expensive calculation of compatible. However, whenever a new term x is generate by Transfer(y, z) in GenerateP, we need to update the map by substituting every occurrence of y with x in the map. To speed up this process, for every term in the map we keep the list of entries where the term occurs.

6. Subtyping using Coercion Functions

We use a simple method of coercion functions [2, 17, 21] to extend our approach to deal with subtyping. We found that this method works well in practice. On the given set of basic types, we model each subtyping relation $v_1 <: v_2$ by introducing into the environment a fresh coercion expression $c_{12} : \{v_1\} \to v_2$. If there is an expression $e : \tau$, and e was generated using the coercion functions, then while translating e into 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 $\rho_1 \to \rho_2 <: \tau_1 \to \tau_2$). We proved that even with those new rules the complexity of the problem does not change and the type inhabitation remains a PSPACE-complete problem. If subtyping constraints are present, then the coercion functions are used in the construction of succinct patterns. However, in the RCN function the coercion functions are omitted when deriving new lambda terms.

7. Evaluation of the Effectiveness of InSynth

This section discusses our implementation, a set of benchmarks we used to evaluate InSynth, and the experimental results.

7.1 Implementation in Eclipse

We implemented InSynth as an Eclipse plugin that extends the code completion feature. It enables developers to accomplish a complex action with only a few keystrokes: declare a 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 (usually 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 snippets are displayed as proposals in the content assist proposal list, in the order corresponding to the weighted ranking. InSynth supports invocation at any location immediately following declaration of 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.

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 goal expressions that needed to be synthesized (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 [23]. We measure whether InSynth can reconstruct an expression equal to the one removed, modulo literal constants (of integer, string, or 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 typed terms that can be derived from the initial environment and that minimize the weight function. To compute initial declaration weights we follow the steps presented in Section 4. The key step is to derive declarations frequencies. Hence, we collected a code corpus which dictates those initial weights. The corpus contains code statistics from 18 Java and Scala open-source projects. Table 3 lists those projects together with their description.

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 3, 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. Overall, we extracted 7516 declarations and identified a total of 90422 uses of these declarations. 98% of declarations have less than 100

	Benchmarks	Size	#Initial	No weights		No corpus		All				Provers	
				Rank	Total	Rank	Total	Rank	Prove	Recon	Total	Imogen	fCube
1	AWTPermissionStringname	2/2	5615	>10	5157	1	101	**1**	8	125	133	127	20123
2	BufferedInputStreamFileInputStream	3/2	3364	>10	2235	1	45	**1**	7	46	53	44	5827
3	BufferedOutputStream	3/2	3367	>10	2009	1	18	**1**	7	11	19	44	5781
4	BufferedReaderFileReaderfileReader	4/2	3364	>10	2276	2	69	**1**	7	43	50	44	0176
5	BufferedReaderInputStreamReader	4/2	3364	>10	2481	2	66	**1**	7	42	49	44	0175
6	BufferedReaderReaderin	5/4	4094	>10	5185	>10	4760	**6**	7	237	244	61	0228
7	ByteArrayInputStreambytebuf	4/4	3366	>10	5146	**3**	94	>10	4	18	22	44	5836
8	ByteArrayOutputStreamintsize	2/2	3363	>10	2583	2	51	**2**	8	63	70	44	5204
9	DatagramSocket	1/1	3246	>10	5024	1	74	**1**	7	80	88	38	5555
10	DataInputStreamFileInput	3/2	3364	>10	2643	1	20	**1**	6	46	52	44	5791
11	DataOutputStreamFileOutput	3/2	3364	>10	5189	1	29	**1**	7	38	45	44	5839
12	DefaultBoundedRangeModel	1/1	6673	>10	3353	1	220	**1**	10	257	266	193	36337
13	DisplayModeintwidthintheightintbit	2/2	4999	>10	6116	1	136	**1**	6	147	154	99	10525
14	FileInputStreamFileDescriptorfdObj	2/2	3366	>10	3882	3	24	**2**	6	17	23	44	3929
15	FileInputStreamStringname	2/2	3363	>10	2870	1	125	**1**	9	100	109	44	4425
16	FileOutputStreamFilefile	2/2	3364	>10	4878	1	86	**1**	8	51	60	44	4415
17	FileReaderFilefile	2/2	3365	>10	3484	2	37	**2**	7	13	20	44	4495
18	FileStringname	2/2	3363	>10	3697	1	169	**1**	7	155	163	44	5859
19	FileWriterFilefile	2/2	3366	>10	4255	1	40	**1**	8	28	36	45	4515
20	FileWriterLPT1	2/2	3363	6	3884	1	139	**1**	7	89	96	44	4461
21	GridBagConstraints	1/1	8402	>10	3419	1	3241	**1**	19	323	342	290	0121
22	GridBagLayout	1/1	8401	>10	2	1	1	**1**	0	1	1	290	56553
23	GroupLayoutContainerhost	4/2	6436	>10	4055	1	24	**1**	10	26	36	190	29794
24	ImageIconStringfilename	2/2	8277	>10	3625	2	495	**1**	13	154	167	300	50576
25	InputStreamReaderInputStreamin	3/3	3363	>10	3558	8	90	**4**	7	177	184	44	4507
26	JButtonStringtext	2/2	6434	>10	3289	2	117	**1**	9	85	95	184	27828
27	JCheckBoxStringtext	2/2	8401	>10	3738	3	134	**2**	18	50	68	188	4946
28	JformattedTextFieldAbstractFormatter	3/2	10700	>10	3087	**2**	2048	4	21	101	122	520	99238
29	JFormattedTextFieldFormatterformatter	2/2	9783	>10	3404	2	67	**2**	15	85	100	419	74713
30	JTableObjectnameObjectdata	3/3	8280	>10	3676	2	109	**2**	13	129	142	300	46738
31	JTextAreaStringtext	2/2	6433	>10	2012	**2**	232	>10	9	293	302	183	29601
32	JToggleButtonStringtext	2/2	8277	>10	3171	2	177	**2**	12	123	135	299	5231
33	JTree	1/1	8278	2	3534	1	3162	**1**	16	2022	2039	298	52417
34	JViewport	1/1	8282	8	5017	**1**	20	8	12	7	19	298	22946
35	JWindow	1/1	6434	3	4274	1	296	**1**	10	425	434	194	2862
36	LineNumberReaderReaderin	5/4	3363	>10	2315	>10	3770	**9**	6	233	239	44	5876
37	ObjectInputStreamInputStreamin	3/2	3367	>10	3093	1	20	**1**	6	29	35	44	5849
38	ObjectOutputStreamOutputStreamout	3/2	3364	>10	4883	1	31	**1**	7	47	54	44	5438
39	PipedReaderPipedWritersrc	2/2	3364	>10	2762	2	54	**2**	8	60	68	44	262
40	PipedWriter	1/1	3359	>10	4801	1	107	**1**	6	133	139	44	5432
41	Pointintxinty	3/1	4997	>10	2068	5	133	**2**	6	96	103	101	8573
42	PrintStreamOutputStreamout	3/2	3365	>10	2100	6	16	**1**	7	20	27	44	5841
43	PrintWriterBufferedWriter	4/3	3365	>10	2521	4	135	**4**	8	36	44	44	448
44	SequenceInputStreamInputStreams	5/3	3365	>10	4777	2	35	**2**	8	20	28	44	5862
45	ServerSocketintport	2/2	4094	>10	2285	2	28	**1**	6	57	63	61	11123
46	StreamTokenizerFileReaderfileReader	3/2	3365	>10	2012	1	34	**1**	8	57	65	44	5782
47	StringReaderStrings	2/2	3363	>10	2006	1	35	**1**	6	37	43	45	5746
48	TimerintvalueActionListeneract	3/3	6665	>10	2051	1	123	**1**	10	189	199	186	34841
49	TransferHandlerStringproperty	2/2	8648	>10	3911	1	27	**1**	14	17	31	319	67997
50	URLStringspecthrows	3/3	4093	>10	3302	6	124	**1**	8	175	183	60	11197

Table 2. 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 use of input statistics, and with weights and input statistics (with the distribution of execution time between the engine and reconstruction parts). The last two columns show execution time for checking provability using the Imogen and fCube provers.

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, we invoked InSynth at the place where the goal expression was missing. We parametrized InSynth with N=10 and used a time limit of 0.5s seconds for $prover$ (Section 5.6) and 7s for the reconstruction. By using a time limit, our goal was to evaluate the usability of InSynth in an interactive environment (which IDEs usually are).

We ran InSynth on the set of 50 benchmarks. Results are shown in Table 2. The Size column represents the size of the goal expression in terms of number of declarations in its structure. It is illustrated in the form c/nc where c is the size with coercion functions and nc is the size without. Note that when c>nc holds, InSynth needs to deal with subtyping to synthesize the goal expression. The #Initial column represents the size of the initial environment, i.e. the number of initial type declarations that InSynth extracts at a given program point. The following columns are partitioned into three groups, one for each variant of the synthesis algorithm: 1) the

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 3. Scala open-source projects used for the corpus extraction.

algorithm without weights (the No weights column), 2) the algorithm with weights derived without the corpus (the No corpus column) and 3) finally, the full algorithm, with weights derived using the corpus (the All column).

In all groups, Rank represents the rank of the goal expression in the resulting list, and Total represents the total execution time of synthesis. The distribution of the execution time between $prover$ and the reconstruction is shown in columns Prove and Recon, respectively. The last column group gives execution times of two state-of-the-art intuitionistic theorem provers (`Imogen` [19] and `fCube` [9]) employed for checking provability of inhabitation problems for the benchmarks.

Table 2 shows the differences in both effectiveness and execution time between the variants of the algorithm.

First, 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 weight function to guide the search, InSynth is driven into a wrong direction toward less important solutions, whose ranks are as low as the actual solutions.

Second, we can see that adding weights to terms helps the search drastically and the algorithm without corpus fails to find the goal expression in only 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.

Third, we show the times for `Imogen` and `fCube` provers on the same set of benchmarks. We can see that our $prover$ is up to 2 orders of magnitude faster than `Imogen` and up to 4 orders than `fCube`. Note that reconstruction of terms in `Imogen` was limited to 10 seconds and `Imogen` failed to reconstruct a proof within that time limit in all cases.

In the case of the full algorithm, the results show that the desired expressions appear in the top 10 suggested snippets in 48 benchmarks (96%). They appear as the top snippet (with 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) from the one used for benchmarks. This suggests that even a knowledge corpus derived from unrelated code increases the effectiveness of the synthesis process; a specialized corpus would probably further increase the quality of results.

In summary, the expected snippets were found among the top 10 solutions in many benchmarks. Weights play an important role in finding and ranking those snippets high in a short period of time (on average around just 145ms). Finally, our $prover$ outperforms two state of the art provers `Imogen` and `fCube`. These results suggest that InSynth is effective in quickly finding (i.e. synthesizing) desired expressions at various places in source code.

8. Related Work

Several tools including Prospector [18], XSnippet [22], Strathcona [13], PARSEWeb [27] 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 initiates multiple queries. In contrast, we generate full expressions using just a single query. 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 [26] and it implements the same behaviour. More advanced solutions appeared recently, such as [3], that proposes declarations, and the Eclipse code recommenders [8], that suggests declarations and code templates. Both systems use API declaration call statistics from the existing code examples in order to offer suggestions to the developer with appropriate statistical confidence value. InSynth is fundamentally different from these approaches (it even subsumes them) and can synthesize even code fragments that never previously occurred in code.

Prospector [18] 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 method calls that starts at $type_{in}$ and ends at $type_{out}$. Prospector ranks solutions by the length, preferring shorter solutions. In contrast, 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 [22] 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 [13], 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 [27] 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 [1]. 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 [14] 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 [12] search engine does not use types for searching functions: its search is based on function names. The main difference between Djinn [25] 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 [20]. The tool [20] generates 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 [29], 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.

Having a witness term that a type is inhabited is a vital ingredient of our tool, so one could think of InSynth as a prover for propositional intuitionistic logic (with substantial additional functionality). Among recent modern provers are Imogen [19] and fCube [9]. 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. This is not entirely surprising because these tools are not necessarily optimized for the task that we aim to solve (looking for shallow proofs from many assumptions), 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 when used to merely 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. This additional functionality of our algorithm is essential for the intended application: using type inhabitation as a generalization of code completion.

For a given type InSynth produces a finite representation of all the type inhabitants. In general, if an expression is an inhabitant of the given type, there is a derivation that proves that fact. Using Curry-Howard isomorphism for each proof derivation there is a lambda term representing it. The problem of enumerating all the proofs for a given formula is an important research topic, since it can be also used to answer other problems like provability or definability. We keep the system of patterns to represent all the type inhabitants, achieving this way finite representation of a possibly infinite set of the proofs. In [7] the authors used a semi-grammatically description of all proof-terms for minimal predicate logic and a positive sequent calculus. The use of grammars is an alternative to our use of graphs as the representation for all solutions; we therefore expect that grammars could similarly be used as the starting point for a practical system such as ours.

9. Conclusions

We have presented the design and implementation of a code completion inspired by complete implementation of type inhabitation for the 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 representation of all solutions (a set of pattens), 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.

Our experience suggests that the idea of computing type inhabitats using succinct types and weights is useful by itself. Moreover, our subsequent exploration suggests that these techniques can also serve as the initial phase of semantic-based synthesis [16]. The idea is to generate a stream of type-correct solutions and then filter it to contain only expressions that meet given specifications, such as postconditions (or, in the special case, input/output examples).

Note that the approach based on the techniques we presented can also generate programs with various control patterns, because conditionals, loops, and recursion schemas can themselves be viewed as higher-order functions. Although we believe the current

results to be a good starting point for such tasks, further techniques may be needed to control larger search spaces for more complex code correctness criteria and larger expected code sizes.

Acknowledgments

We thank the anonymous reviewers of PLDI 2013 for useful feedback. We are grateful to Iulian Dragos and the Scala IDE team for the collaboration on integrating InSynth into Scala IDE. We thank Martin Odersky, Aleksandar Prokopec, and Sean McLaughlin for useful discussions. Tihomir Gvero is supported by the European Research Council (ERC) Project "Implicit Programming", `http://lara.epfl.ch/w/impro`. Ivan Kuraj was supported by a Google Summer of Code project.

References

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

[2] 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.

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

[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] G. Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, II:1009–1062, 2001.

[7] G. Dowek and Y. Jiang. Enumerating proofs of positive formulae. *Comput. J.*, 52(7):799–807, Oct. 2009. ISSN 0010-4620.

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

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

[10] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets (tool demonstration). In *23rd Int. Conf. Computer Aided Verification, July 14-20*, 2011.

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

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

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

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

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

[16] I. Kuraj. Interactive code generation. Master's thesis, EPFL, February 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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