View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

On Fast Code Completion using Type Inhabitation

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 with subtyping. We show that the inhabitation problem in the presence of subtyping remains PSPACE-complete. We introduce a succinct representation for type judgements that merges types into equivalence classes to reduce the search space. We introduce a proof rule on this succinct representation of types and show that it is sound and complete for inhabitation. We implemented the resulting algorithm and deployed it as a plugin for the Eclipse IDE for Scala.

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 [23] 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 productively use 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 set of values and their types), find an expression e of this type T, i.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 is decidable and PSPACE-complete. If we add finite intersection types to the language, the type inhabitation becomes an EXPSPACE-complete problem [17]. In this paper, we pursue a sweet spot between these: we consider ground types with function type constructor and a subtyping relation. We develop an algorithm that is complete in the lambda calculus sense, so it is able to synthesis not only function applications, but also lambda abstractions. From the theoretical side, we show that the problem remains PSPACE complete. We present our result in a *succinct ground 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. We also show how to use weights to guide the search, building on our previous experience [6, 7, 16]. We present an implementation within the Eclipse IDE for Scala. Our experience show 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 synthesis 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) 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, InSynth2 within a Scala IDE. We used a corpus of open-source Scala projects as well as the standard Scala library to collect the usage statistics for the initial weights of declarations. We ran InSynth2 on more than 60 examples from the Web, written to illustrate API usage, as well as examples from larger projects. The results show that in over 70% of examples the expected snippet appears among the first five solutions. Moreover, in over 40% of examples, the expected snippet appears first in the list.

To estimate the interactive nature of InSynth2, we measured the time needed to synthesize the expected snippet as a function of a number of visible declarations. We found that a sufficient number of snippets can be typically generated in half a second. This suggests that InSynth2 can efficiently and effectively help the user in software development. Furthermore, we evaluated a number of techniques deployed in our final tool, and found that all of them are important for obtaining good results.

1.1 Contributions

- Following the agenda set in [6, 7, 16] we propose a new code generation feature for IDEs, and show that it closely corresponds to a weighted version of the type inhabitation problem.
- As a sweet spot in the space of type systems suitable for generating code fragments, we identify the lambda calculus with ground types and a subtyping relation structurally extended to function types. Inspired with the properties of conjunction in intuitionistic logic, we introduce succinct representation of types in this system and presented a complete proof rule for solving type inhabitation. Using this representation, we show that the inhabitation problem is PSPACE complete, demonstrating that there is fundemantally no extra price to be paid for supporting subtyping.
- We design and implement an efficient goal-directed search algorithm that solves the type inhabitation problem by operating on the succinct representation of types. Furthermore, we introduce a best-first search goal-directed version of the algorithm that uses weights to guide the search and prioritize the synthesized expressions. To compute the appropriate weights, we developed a Scala compiler plugin that performed off-line analysis on a number of Scala open source projects to compute the frequency of method uses. We also propose an effective policy of giving higher priority to declarations closer to the cursor.
- We implemented all proposed techniques in the InSynth2 tool, including the type inhabitation algorithm working on succinct type representation and exploring the search space guided by the computed weights, the encoding of Scala declarations into our representation, and generation of valid Scala expressions. We integrated these techniques with the Eclipse IDE for Scala.
- We evaluate InSynth2 on a number of examples and larger projects. The evaluation shows that InSynth2 in many cases synthesizes the expected solutions and ranks them reasonably high in the list of offered choices. Our tool and evaluation results are available from
  `http://lara.epfl.ch/w/insynth`

2. Motivating Examples

Here we illustrate the functionality of InSynth2 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 from Scala IDE for Eclipse project `http://scala-ide.org/`. 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 InSynth2 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 following code in the editor:

```
import java.io._
...
def main() = {
  var body = "email.txt"
  var sig = "signature.txt"
  val all:SequenceInputStream = ■
}
```

If we invoke InSynth2 at the program point indicated by ■, in a fraction of a second it displays the following ranked list of five expressions:

```
1. new SequenceInputStream(new FileInputStream(sig),
                           new FileInputStream(sig))
2. new SequenceInputStream(new FileInputStream(body),
                           new FileInputStream(sig))
3. new SequenceInputStream(new FileInputStream(sig),
                           new FileInputStream(body))
4. new SequenceInputStream(new FileInputStream(body),
                           new FileInputStream(body))
5. new SequenceInputStream(new FileInputStream(sig),
                           System.in)
```

Seeing the list, the developer can decide that e.g. the second item in the list matches his intention, and select it to be inserted into the editor buffer. This example illustrates that InSynth2 only needs the current program context, and does not require additional information from the user. InSynth2 is able to use both imported values (such as the constructors in this example) and locally declared ones (such as body and sig). InSynth2 supports methods with multiple arguments and synthesizes expressions for each argument.

In this particular example, InSynth2 loads over 3000 initial declarations from the context, and finds the expected solution in less than 250 milliseconds, as shown in Table 2, benchmark 7.

The effectiveness in the above example is due to several aspects of InSynth2. InSynth2 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. InSynth2 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 a 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.

```
// Scala IDE for Eclipse: org.scala−ide.sdt.core/src/scala/tools/
// eclipse/semantichighlighting/classifier/TypeTreeTraverser.scala
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 InSynth2 at the place ■, the system suggests several suggestions, and the one ranked first turns out to be exactly the one expected, namely

```
new FilterTypeTreeTraverser(var1 => p(var1))
```

The constructor FilterTypeTreeTraverser is an higher order function that takes another function, in this case p. In this example, In-

Synth2 loads over 4000 initial declarations and finds the snippets in less than 300 milliseconds.

Drawing Layout. Consider next the problem 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. InSynth2 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. Evaluation of the Effectiveness of InSynth2

We implemented InSynth2 and evaluated it on over 66 examples. This section evaluates the effectiveness of InSynth2, showing that the techniques we developed and implemented result in a useful tool, appropriate for interactive use within an integrated development environment (concretely, the Eclipse IDE for Scala).

3.1 Creating Benchmarks

There is no standarized set of benchmarks for the problem that we examine, so we constructed our own benchmark suite. We collected benchmarks primarily from `http://www.java2s.com/`. These examples illustrate correct usage of particular API functions and (possibly generic) classes. We manually translated the examples from Java into equivalent Scala code. The original code imports only the classes used in the example. We therefore generalize the import declaration to include more definitions and thereby make the synthesis problem more difficult.

Our idea of measuring tool effectiveness is to estimate its ability to reconstruct a missing expression from a program. We therefore chose a declaration that is used to initialize a variable in an example code. This initialization may be written in several steps, spanning several lines. We identify one or all expressions that contribute to this initialization, save them as the expected result, and delete them from the program. The resulting benchmark is a partial program, much like a program sketch [20]. We measure whether a tool can reconstruct the expression equal to the one removed modulo literal constants (integers, strings, and booleans). Our benchmark suite is available from the InSynth2 web site.

When we invoke InSynth2, it returns N recommended expressions. We call a run successful if the expression that was removed from the example code appears among these N expressions. Usually we run InSynth2 with $N = 5$ and using a time limit of 0.5 seconds for the core quantitative type inhabitation engine; the table (and our experience) shows that the overall response time remains below one second. By using a time limit, we aim to evaluate the usability of InSynth2 in an interactive environment.

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 1. Scala open source project used for the corpus extraction.

3.2 Corpus for Computing Symbol Usage Frequencies

Our algorithm searches for type bindings that can be derived from an initial environment and that minimize a weight function. To compute these initial weights we use the technique from Section 7.3. This technique requires, among others, an initial assignment of weights to variables names. To compute this initial assignment of weights to names, we mine usage frequency information from 18 Scala open source projects. Table 1 lists these open source projects. Among others we analyze the Scala compiler, which is written in the Scala language itself. In addition to the projects listed in the table we analyze the Scala standard library, which mainly consists of wrappers around Java API calls. We extract usage information only about Java and Scala APIs, but not declarations specific to the projects themselves. Overall we extracted 7516 symbol declarations and identified a total of 90422 uses of these symbols. The maximal number of occurrences of a single symbol is 5162 (for the symbol `&&`), whereas 98% of symbols have less than 100 uses in the entire corpus.

3.3 Platform for Experiments

We ran all experiments on an Intel(R) Core(TM) i7 CPU 2.67 GHz with 4 GB RAM machine. InSynth2 is currently implemented sequentially and does not make use of multiple CPU cores. The operating system was Windows 7(TM), Scala version is 2.8, and Java(TM) Virtual Machine is version 1.6.0_22.

3.4 Measuring Overall Effectiveness

We ran InSynth2 in its optimal configuration to recover 66 removed expressions from benchmarks (see Table 2 for a subset of our results). The results show that the desired expression appears in the top 10 snippets (suggested expressions) in 48 benchmarks (73%). It appears as the top snippet (with rank 1) in 32 benchmarks (48%). Note that our corpus (Section 3.2) is disjoint (and somewhat different in nature) from the examples on which we performed the evaluation. The results suggest that InSynth2 can synthesize expected expressions in useful pieces of software.

Table 2 presents the results, in more details, on 10 benchmarks out of 66 benchmarks that we examined. The length column represents the number of declarations in the expected expression. The “Initial” column is the number of initial type declarations that InSynth2 extracts at a given program point and gives to the search procedure. Time includes declaration loading, encoding and weight assignment time, as well as the time within the prover (which was set to 0.5 seconds). InSynth2 was able to synthesize expected ex-

Benchmarks		Length	# Initial	Solution Rank	Time [ms]
1	BufferedReaderInputStreamReader	2	3328	1	370
2	BufferedReaderReaderin	4	4011	6	681
3	DataInputStreamFileInputStreamfileInputStream	2	3328	1	259
4	FileReaderFilefile	2	3329	2	241
5	PipedReaderPipedWritersrc	2	3328	2	238
6	ServerSocketintport	2	4011	1	319
7	SequenceInputStreamInputStreams1InputStreams2	5	3329	2	246
8	TimerintvalueActionListeneract	3	6794	1	575
9	TransferHandlerStringproperty	2	8764	1	742
10	URLStringspecthrowsMalformedURLException	3	4010	1	410

Table 2. Measuring Overall Effectiveness.

pressions in all these benchmarks. We therefore measured the times for InSynth2 to reach the expression that was expected; we found that this time ranges from below 6 to 48 millisecond.

In summary, the expected snippets appear among the top 10 solutions in many examples in short period of time (less than a second).

4. Type Inhabitation Problem for Succinct Ground Types

To answer whether there is a code snippet of the given type, we recall a related problem, the so called *type inhabitation problem*. In this section we first establish a connection between the type inhabitation problem and the problem of finding code snippets. Furthermore, we review relevant definitions and theorems from type theory, and describe an algorithm for checking whether a type is inhabited. Our algorithm is constructive: if the given type is inhabited, it returns an inhabitant.

Let T be a set of types and let E be a set of expressions. A *type environment* Γ is a finite set $\{e_1 : \tau_1, \dots, e_n : \tau_n\}$ containing pairs of the form $e_i : \tau_i$, where x_i is an expression of a τ_i. The pair $e_i : \tau_i$ 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 $\Gamma \vdash e : \tau$?

The type system of Scala [4] is complex, based on F_ω. However, the type inhabitation problem is undecidable already for the Hindley-Milner type system (there are only universal quantifiers on the top level). Therefore, in InSynth2 we focus on the subset of Scala. We investigate the type inhabitation problem for the standard simply typed lambda calculus, defined on the ground types, in the presence of subtyping constraints. We describe in Section 6 how to derive types from Scala.

At a high-level, the algorithm on which our tool is based consists of he following three steps:

1. Parse the program and derive type constraints.
2. Following the standard typing rules (as for example, in the simply typed lambda calculus) derive new type declarations, directing the search towards the inhabitants of the required type.
3. Rank the found type inhabitants, create the code snippets from the found terms and output them to the user.

In the remainder of this section we first describe the standard lambda calculus and afterwords introduce a new succinct representation of the lambda calculus. In order to distinguish the two calculi we use $\vdash_\lambda$ and $\vdash_S$ to denote derivability in the simply typed lambda calculus and in the succinct ground types calculus, respectively. Similarly, we use Γ_λ and Γ_S for type environments. We next review the definition of the simply typed lambda calculus and introduce a new succinct representation and reasoning.

4.1 Simply Typed Lambda Calculus with Subtyping

Let B be a set of basic types. The 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 it is clear from the context we only write τ_λ. With $\tau <: \sigma$ we denote that τ is a subtype of σ. We assume that if there are some subtyping relations, they are initially explicitly defined on T.

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

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

From a given type environment Γ one can derive new type declarations by applying the rules given in Figure 1. This calculus also contains rules for subtyping. Although $<:$ is a reflexive and transitive relation, for the purpose of finding type inhabitants we do not consider the rule for reflexivity. The other axioms, Subsume, Trans and CVarian, are necessary for inferring further type declarations based on the given initial subtyping relations.

$$\textsc{Axiom}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_\lambda x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_\lambda e_1 : \sigma \to \tau \qquad \Gamma \vdash_\lambda e_2 : \sigma}{\Gamma \vdash_\lambda e_1\ e_2 : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma, x:\sigma \vdash_\lambda e:\tau}{\Gamma \vdash_\lambda \lambda x:\sigma.\ e:\sigma \to \tau} \qquad \textsc{Subsume}\ \frac{\Gamma \vdash_\lambda e:\tau \qquad \tau <: \sigma}{\Gamma \vdash_\lambda e:\sigma}$$

$$\textsc{Trans}\ \frac{\tau <: \sigma \qquad \sigma <: \rho}{\tau <: \rho} \qquad \textsc{CVarian}\ \frac{\tau_1 <: \sigma_1 \qquad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

Figure 1. Calculus Rules for the Simply Typed Lambda Calculus with Subtyping

4.2 Succinct Ground Types

Consider the following code

```
val a:Int = 0
def f(i1: Int, i2: Int, i3: Int):String = {...}
```

In the standard lambda calculus this code translates to type environment $\Gamma = \{a : \text{Int}, f : \text{Int} \to \text{Int} \to \text{Int} \to \text{String}\}$. Checking whether there is an inhabitant of type String requires three calls of the App rule. In order to reduce the search space we introduce succinct ground types. This new formalism enables us to find an inhabitant in only one step.

DEFINITION 4.1 (Succinct Ground Types). *Let B_S be a set containing basic types. Ground succinct types τ_s are constructed by the grammar:*

$$\tau_s ::= \{\tau_s, \ldots, \tau_s\} \rightarrow B_S$$

We denote the set of all ground succinct types with $\tau_s(B_s)$, sometimes also only with τ_s.

A type declaration $f : \{t_1, \ldots, t_n\} \rightarrow t$ 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 a type $\emptyset \rightarrow t$: it is a type of a function that takes no arguments and returns a value of type t. Therefore for us types t and $\emptyset \rightarrow t$ are equivalent.

Every type $\tau \in \tau_\lambda(B)$ can be converted into a succinct ground type in $\tau_s(B)$. With v we denote a basic type, $v \in B$. A translation function $Tr : \tau_\lambda(B) \rightarrow \tau_s(B)$ is defined as follows:
$Tr(v) = v \quad Tr(\tau_1 \rightarrow \tau_2) = \{Tr_a(\tau_1)\} \cup S_r(\tau_2) \rightarrow R_r(\tau_2)$

$Tr_a(v) = v \qquad Tr_a(\tau_1 \rightarrow \tau_2) = Tr(\tau_1 \rightarrow \tau_2)$
$Tr_r(v) = \emptyset \rightarrow v \qquad Tr_r(\tau_1 \rightarrow \tau_2) = C(\tau_1 \rightarrow \tau_2)$
Let $Tr_r(\tau) = S \rightarrow \tau_r$. Then:
$S_r(\tau) = S$ and $R_r(\tau) = \tau_r$

In the implementation we made the rule for $Tr(\tau_1 \rightarrow \tau_2)$ more general and it became $Tr(\tau_1 \rightarrow \ldots \rightarrow \tau_{n-1} \rightarrow \tau_n) = \{Tr_a(\tau_1), \ldots, Tr_a(\tau_{n-1})\} \cup S_r(\tau_n) \rightarrow R_r(\tau_n)$. This way we converted the example from the beginning of this section in a single step to the type environment $\Gamma_s = \{a : \text{Int}, f : \{\text{Int}\} \rightarrow \text{String}\}$.

Succinct Ground Terms. With $f\{a_1 : t_1, \ldots, a_n : t_n\} : t$ we denote the application of a function of type $\{t_1, \ldots, t_n\} \rightarrow t$ to values a_i of type t_i. Note that $f\{a_1, \ldots, a_n\}$ is not equivalent to $f(a_1, \ldots, a_n)$, since some of arguments might appear several times. Similarly, the expression $\{\lambda x : t\}.e$ represents an expression $\lambda x_1 : t.\lambda x_n : t.e(x_1, \ldots, x_n)$, where n depends on the original type declaration in the simply typed lambda calculus.

We assume there is a method *Orig* that translates a succinct ground term into a lambda term. Intuitively, we could define *Orig* as:
$Orig(x : v, \Gamma) = x : v, \qquad$ for $v \in B$
$Orig(f\{a_1 : t_1, \ldots, a_n : t_n\} : t, \Gamma) = e : t$
where:
$Orig(a_i : t_i, \Gamma) = td_i$
$f : T \in \Gamma$
$\{td_1, \ldots, td_n, f : T\} \vdash_\lambda e : t$

Similarly, we can define *Orig* function for succinct lambda terms.

It is clear that using the succinct representation to generate only one term might not be very efficient. The succinct representation shows its advantages when we generate more terms simultaneously. In Section 7 we give an overview how our implementation for term reconstruction works, but at this point it is enough to know that a succinct ground term can be converted to a lambda term.

Calculus. Figure 2 describes the calculus for deriving new type declarations for ground succinct types.

The AppAbs rule allows forward and backwards reasoning. In general, in the forward reasoning we generate new type declarations by applying functions to already existing terms. In contrast to the forward reasoning, we apply the backward reasoning when there is a function f that returns the required type. In backward reasoning we try to generate the arguments for f and then apply f to those arguments.

Additionally, the AppAbs rule unites the abstraction and application rule from the standard simply typed lambda calculus. When all $S_i = \emptyset$, then each Γ_{S_i} is also the empty set and term e_i is derived directly from Γ. In such settings the AppAbs rule corresponds to the standard application rule. If any of S_i is a non-empty set, then f takes a lambda term (a function) as an argument. In that case we need to generate a lambda term and to do that we add fresh typed variables of the types appearing in S_i to Γ and we try to construct an expression e_i of type t_i. If we succeed, we construct a lambda term : a function that takes arguments from S_i and returns e_i. Clearly, this function has the required type $S_i \rightarrow t_i$. The operator Λ that appears in the App rule creates an expression of the required type. It can be either an expression of type t_i (is $S_i = \emptyset$), or a lambda term.

Type Inhabitation Problem. For a given type environment Γ and a succinct ground type τ, we address the type inhabitation problem by adding a new type declaration $\mathsf{goal} : \{\tau\} \rightarrow \bot$ and directing the search towards an inhabitant of type $\bot$. Symbols goal and $\bot$ are fresh and previously unused, so an inhabitant of type $\bot$ can only be an expression of the form $\mathsf{goal}\{e\}$, where $e : \tau$.

Although the AppAbs rule cannot generate functions (lambda terms) at the top-level, with such encoding of the type inhabitation problem we can still search for inhabitants of functions, as lambda terms can be generated as arguments for the goal function.

4.3 Subtyping using Coercions

While the simply typed lambda calculus in Figure 1 contained the rules for subtyping, the calculus given in Figure 2 does not contain them. This is because we encode subtyping constraints directly into the type environment.

A powerful method to model subtyping is to use coercion functions [2, 12, 18]. This approach raises non-trivial issues when we perform type checking or type inference, but becomes simple and natural if the types are given and we search for the terms.

On the given set of basic types, we model each subtyping relation $A <: B$ by introducing into the environment a fresh coercion expression $c_{AB} : \{A\} \rightarrow B$. 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, e is simply removed.

4.4 Soundness and Completeness of Succinct Ground Calculus

In this section we show that the succinct ground calculus is sound and complete with respect to the type inhabitation problem. Let $\Gamma_\lambda = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ be a type environment. We first observe that for the purpose of answering the type inhabitation problem we can transform Γ_λ to a type environment Γ'_λ where the following holds: for every two type declarations $x_i : t_i \in \Gamma'_\lambda$ and $x_j : t_j \in \Gamma'_\lambda$ $i \neq j \Rightarrow t_i \neq t_j$. To construct Γ'_λ let us assume that Γ contains $y_1 : t$ and $y_2 : t$. Every occurrence of y_2 in Γ_λ can be replace with y_1 and no term will change its type with this substitution. Thus we can freely eliminate $y_2 : t$ from Γ_λ. Note that for every type holds: it is inhabited in Γ_λ iff it is inhabited in Γ'_λ. From now on we assume that Γ_λ contains only distinct type declarations.

Given a type environment $\Gamma_\lambda = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ and a type τ, we define type environment $\Gamma_{TIP}(\Gamma_\lambda, \tau)$ as follows:

1. We translate Γ_λ to a succinct type environment, by translating every type:
$$\Gamma_1 = \{x_1 : Tr(\tau_1), \ldots, x_n : Tr(\tau_n)\}$$
2. For every subtyping relation defined on the basic types, we add a coercion function:
$$\Gamma_2 = \cup\{c_{\tau_1,\tau_2} : Tr(\tau_1 \rightarrow \tau_2) \mid \tau_1 <: \tau_2\}$$
3. For type τ we add a type declaration
$$\Gamma_3 = \{\mathsf{goal} : \{Tr(\tau)\} \rightarrow \bot\}$$

$$\text{Ax}\ \frac{x : t \in \Gamma}{\Gamma \vdash_s x : t}$$

$$\text{AppAbs}\ \frac{\Gamma \vdash_s f : \{S_1 \to t_1, \ldots, S_n \to t_n\} \to t \qquad \Gamma \cup \Gamma_{S_1} \vdash_s e_1 : t_1 \qquad \ldots \qquad \Gamma \cup \Gamma_{S_n} \vdash_s e_n : t_n}{\Gamma \vdash_s f\{\Lambda(\Gamma_{S_1},\, e_1) : S_1 \to t_1, \ldots, \Lambda(\Gamma_{S_n},\, e_n) : S_n \to t_n\} : t}$$

$$\Gamma_S = \bigcup_{t_i \in S} \{x_i : t_i \mid x_i \text{ fresh }\} \qquad \Lambda(\emptyset, e) = e \qquad \Lambda(\{x : t\} \cup \Gamma_1, e) = \{\lambda x : t\}.\Lambda(\Gamma_1, e)$$

Figure 2. Calculus for deriving new type declaration for succinct ground types

4. Finally,
$$\Gamma_{TIP}(\Gamma_\lambda, \tau) = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$$

THEOREM 4.2 (Soundness and Completeness). *Given a type environment Γ_λ and a type τ, τ is inhabited (i.e. there is a term e such that $\Gamma_\lambda \vdash_\lambda e : \tau$) iff $\bot$ is inhabited (i.e. there is e' such that $\Gamma_{TIP}(\Gamma_\lambda, \tau) \vdash_s e' : \bot$).*

Proof By induction on the structure of a term derivation - we show that each rule in one calculus can be simulated in the other calculus. As all the simulation steps are similar, we use the Abs rule as an illustration; the other simulations are proved the same way. Let $\Gamma'_\lambda = \Gamma_\lambda \cup \{x : \sigma\}$. The Abs rule says that if we can conclude from Γ'_λ that τ is inhabited, then from Γ_λ follows that $\sigma \to \tau$ is inhabited. To prove that the same holds in the succinct ground types calculus, let us assume that from Γ'_λ follows that τ is inhabited. By the induction hypothesis, we know that $\bot$ is inhabited and it can be derived from$\Gamma_{TIP}(\Gamma'_\lambda, \tau)$. Now we need to prove that there is a term e such that $\Gamma_{TIP}(\Gamma_\lambda, \sigma \to \tau) \vdash_s e : \bot$. We prove that by induction on the structure of τ.

Let τ be a basic type v. Then $Tr(\tau) = Tr(v) = v$. Based on the fact that from $\Gamma_{TIP}(\Gamma'_\lambda, \tau)$ we can show that $\bot$ is inhabited, and on the fact that goal and $\bot$ are previously unused, we conclude that there must be a term e_1 such that $\Gamma_{TIP}(\Gamma'_\lambda, \tau) \vdash_s e_1 : v$, and in that case goal$\{e_1\}$ is an inhabitant of $\bot$. By construction $\Gamma_{TIP}(\Gamma_\lambda, \sigma \to \tau) = (\Gamma_{TIP}(\Gamma'_\lambda, \tau) \setminus \{\text{goal} : \{Tr(\tau)\} \to \bot\} \setminus \{x : Tr(\sigma)\}) \cup \{\text{goal}_1 : Tr(\sigma \to \tau) \to \bot\}$. Using this set equality and the fact that there exists a derivation tree for $e_1 : v$, we can show that $\Gamma_{TIP}(\Gamma_\lambda, \sigma \to \tau) \vdash_s \text{goal}_1\{\Lambda(\{x : Tr(\sigma)\}, e_1) : \sigma \to v\} : \bot$.

When τ is a complex type, the proof is almost identical, only, in addition to $Tr(\sigma)$, we need to consider $S_r(\tau)$. By applying the demonstrated techniques we can show that each rule of the simply typed lambda calculus can be simulated in the succinct ground types calculus. The proofs for the subtyping rules in Section 4.5.

To prove that the converse holds as well, we again use the same methods. However, for the complete proof we also need the *Orig* function which takes a succinct ground expression and returns a lambda expression. We prove that depending on the emptiness of the S_i set, the rule AppAbs is either simulated by the App rule or by the Abs rule. Nevertheless, the proof is again inductive, and applies the same reasoning as before (inductive case splitting based on the complexity of types), so we omit it.

4.5 Complexity of the Type Inhabitation Problem for Ground Types with Subtyping

The type inhabitation problem for the simply typed lambda calculus is decidable, although of a high complexity (PSPACE-complete [21]). As a direct consequence of Theorem 4.2 we conclude:

THEOREM 4.3. *The type inhabitation problem for the succinct ground types is PSPACE-complete.*

We are interested in whether the complexity of the type inhabitation problem will increase if we add the rules for subtyping. The original proof by Statman [21] and a later, more constructive, proof by Urzyczyn [25] did not consider subtyping constraints.

The type constraints are defined only on the set of the basic types. However, at first sight it may appear that the subtyping constraints can result in instantiating exponentially many additional constraints, because of the transitivity and the CVarian rule. To address this problem we observe that each of the three rules for subtyping given in Figure 1 can be encoded into the calculus for succinct ground types as a single rule:

- Subsume: Let $\Gamma_1 = \Gamma_{TIP}(\Gamma_\lambda, \tau)$. There is an inhabitant of $\bot$ of the form goal$\{e\} : \bot$. We create Γ_2 which is the same as Γ_1 only $\{\text{goal} : \{Tr(\tau)\} \to \bot\}$ is replaced with $\{\text{goal} : \{Tr(\sigma)\} \to \bot\}$. We can easily construct an inhabitant of $\bot$ in Γ_2 using term e
- Trans: If there is an inhabitant of type $Tr(\tau \to \sigma)$ (coercion function c_1) and an inhabitant of type $Tr(\sigma \to \rho)$ (coercion function c_2), by induction on the structure of τ (basic type or a composed type) we can show that then there is also an inhabitant of type $Tr(\tau \to \rho)$ – it is a composition of two coercion function c_1 and c_2.
- CVarian: For clarity of exposition, let us assume that τ, σ and ρ are basic types. The premises of the CVarian rule are then encoded as: c_1 is an inhabitant of type $\{\tau_1\} \to \sigma_1$ and c_2 is an inhabitant of type $\{\sigma_2\} \to \tau_2$. We need to show that there is an inhabitant of type $\{\{\sigma_1\} \to \sigma_2, \tau_1\} \to \tau_2$. Applying the AppAbs rule, we show that it is an inhabited type, and generate the following witness term: $\{\lambda f : \{\sigma_1\} \to \sigma_2\}.\{\lambda x : \tau_1\}.c_2\{f\{c_1\{x\}\}\}$. When the types τ, σ and ρ are composed types, the same reasoning is applicable, only we need to additionally include the Tr function.

The above reasoning shows that we do not need to additionally instantiate new type constraints: it is enough to encode those that are already present on the basic types. The other properties (like subsumption, transitivity, and covariance and contravariance) simply follow from the calculus rules. This observation is crucial in establishing the complexity result for the type inhabitation problem in the presence of subtyping constraints.

THEOREM 4.4. *Given a type environment with type constraints defined on the set of basic types, checking whether some type is inhabited is an PSPACE-complete problem in both, the simply typed lambda calculus and in the succinct ground types calculus.*

Proof We follow the constructive proof given in [25] to establish the PSPACE-completeness of the type inhabitation problem. In that proof Urzyczyn did not consider the subtype constraints. However, we show that adding them does not increase the complexity of the problem. The full proof is available in [25]. In this paper we provide the highlight of the proof and show how it has to be modified so that it can also support the subtyping constraints.

The lower bound of the problem (PSPACE-hardness) is shown by taking a classical PSPACE-complete problem, the satisfiability problem of second-order propositional formulas (QBF), and encoding it into into the type inhabitation problem. Note that this encoding does not in any way effect the subtyping constraints.

To prove that the type inhabitation problem is in PSPACE, Urzyczyn described a deterministic polynomial space algorithm that determines whether a given type is inhabited. The algorithm is similar to the algorithm given in Figure 3. Urzyczyn's algorithm is a recursive algorithm, with the depth of recursion bounded by n^2, where n is the cardinality of the set of all types appearing in Γ.

As we showed earlier, formalizing coercion functions on the basic types only is sufficient to answer the type inhabitation problem, we do not need to instantiate any additional axioms. Therefore, if subtyping constraints are present in the specification and we encode them by using coercion functions, we can still apply the same algorithm to check if a type is inhabited.

The original algorithm did not change, and the only thing that changed is the depth of recursion – now it is bounded by the $(c+n)^2$, where c is the number of coercion functions defined on the basic types. This change does not effect the complexity of the algorithm, i.e. it remains in polynomial space.

4.6 Basic Algorithm Used in InSynth2

Finally, in Figure 3 we present a high-level overview of the algorithm used in the implementation of InSynth2. The algorithm corresponds to a repeated execution of the AppAbs command.

```
INPUT:    type environment $\Gamma_s$ and type $\tau$
OUTPUT:   expression $e$ such that $\Gamma_s \vdash_s e : \tau$,
          UNDEF if $e$ does not exist

TIP($\Gamma$, $\tau$) =
switch ($\tau$)
  case $S \to \tau_1$: // $S \neq \emptyset$
    val e = TIP($\Gamma \cup \Gamma_S, \tau_1$)
    if e == UNDEF return e
      else return $\Lambda(\Gamma_S, e)$
  case $\emptyset \to \tau_1$:
    val R = $\{f : \{A_1, \ldots, A_n\} \to \tau_1 \mid f : \{A_1, \ldots, A_n\} \to \tau_1 \in \Gamma\}$
    if R == $\emptyset$ return UNDEF
    run in parallel forall $r \in R$
      // with weights this line is
      // "run in parallel [sorted by weights] forall $r \in R$"
    let $r \in R := g : \{B_1, \ldots, B_m\} \to \tau_1$
    if m == 0 return g
    foreach $B_i$ do
      val $e_i$ = TIP($\Gamma \setminus \{r\}, B_i$)
      if ($\forall i.e_i \neq$ UNDEF) return g $\{e_1, \ldots, e_m\}$
                                    else return UNDEF
```

Figure 3. Constructive algorithm for checking whether a type τ is inhabited, i.e. whether there is an expression e such that $\Gamma_s \vdash_s e : \tau$

In the description of the algorithm we use the same notation as in the succinct ground calculus rules. With Γ_S we denote the type environment extended with fresh new variables of the types appearing in S, and $\Lambda(\Gamma_S, e)$ constructs a succinct lambda term, as defined in Figure 2.

We already mention that the most important role in ranking the snippets plays the weight function. The above algorithm can be easily extended so that it reasons about weighted terms. In the next section we show how to do it.

5. 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 the word "best". For this purpose to every term we assign a weight. We decided that, similarly as in resolution-based theorem proving, a lower weight will indicate the higher relevance of the term. Having 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 term variable and to each type variable a non-negative number. As the weight plays the crucial role in directing the search for inhabitants, it is important to assign the right weights. In Section 7.3 we describe how do we compute the weight function and which techniques and heuristics are we using. In general, two main sources of determining the value of the weight of a symbols, are:

1. a proximity to the point at which InSynth2 is invoked. We assume that the user would more likely prefer to get a code snippet composed from values and methods defined nearer to the program point. Thus we assign the lower weight to the functions which are declared closer to the program point where InSynth2 is invoked.
2. a frequency with which the symbol appears in the training data corpus, which is described in Section 3.2.

But for now, we assume that there is an initial, pre-computed non-negative weight function assigned to every symbol.

Given the weight function w defined on the term and type variables, the weights of terms and composed types are then defined as:

$w(f\{e_1, \ldots, e_n\}) = w(f) + w(e_1) + \ldots + w(e_n)$
$w(f\{\lambda x : \tau\}.e) = w(\lambda) + w(x) + w(\tau)$
$w(S \to \tau) = w(S) + w(\to) + w(\tau)$
$w(\emptyset) = 0$
$w(\{\tau_1, \ldots, \tau_n\}) = w(\tau_1) + \ldots + w(\tau_n)$

In summary, to compute the weight of non-symbols we sum the weights of all the symbols appearing in the term or he type. All the introduced term variables have the same weight.

The weight of an entire type declaration is defined as $w(e : \tau) = w(e) + w(\tau)$.

The quantitative type inhabitation problem can be seen as an extension of the type inhabitation problem and in Section 7 we describe how is the underlying algorithm of InSynth2 guided by the weights of the terms.

6. From Scala Programs to Simply Typed Lambda Calculus

In this section we describe the translation of the Scala declarations with ground and function types into simply typed Lambda calculus variables. First we define the subset of the Scala types we translate and later we define the translation function and finally we give a table with the examples of the translation.

DEFINITION 6.1 (Scala Ground Types). *Let C_{scala} be a fixed finite set. For every $c \in C_{scala}$, with $c/_n$ we denote the arity of the element. The elements of arity 0 are called constants. The set of all Scala ground types T_{scala} is defined by the grammar:*

$$T_{scala} ::= T_{scalaFun} \mid T_{scalaSimple}$$

$$T_{scalaFun} ::= (T_{scala} \times \ldots \times T_{scala}) \to T_{scala}$$

$$T_{scalaSimple} ::= C_{scala}(T_{scala}, \ldots, T_{scala})$$

For instance, we translate a Scala type declaration $a : Int$ with the simple type $Int \in T_{scalaSimple}$ or $g(x_1 : Int, x_2 : Boolean) : String$ with the function type $(Int \times Boolean) \rightarrow String \in T_{scalaFun}$.

In Figure 4 we define the translation procedure Tr that takes a Scala type $\tau_{scala} \in T_{scala}$ and translate it to $\tau_\lambda \in T_\lambda$.

```
$Tr(\tau_{scala}) =$
switch $(\tau_{scala})$
case $(\tau_1 \times \cdots \times \tau_n) \rightarrow \tau_r$:
  $(Tr(\tau_1) \rightarrow \cdots \rightarrow Tr(\tau_n) \rightarrow Tr(\tau_r))$
case $c(\tau_1, \ldots, \tau_n)$:
  $c(Tr(\tau_1), \ldots, Tr(\tau_n))$
end switch
```

Figure 4. The type translation algorithm.

We translate each Scala type declaration $a : \tau$ where $\tau \in T_{scala}$ to the simply typed Lambda calculus variable $a : Tr(\tau)$. For every Scala subtype relation (e.g. **class** τ_1 **extends** τ_2) over simple types, we introduce subtyping relation in the simply typed lambda calculus (e.g. $\tau_1 <: \tau_2$).

We illustrate the translation function on examples in Table 3.

7. Extensions of the Basic Algorithm

We next present the extensions of the algorithm to make it useful as an interactive tool for synthesizing expression suggests from which the developer can select a suitable expression. Such an algorithm needs to 1) generate multiple solutions (and not just a single one), and 2) rank these solutions to maximize the chances that the suggestions will be relevant for the developer. We therefore first present the extension with generation a representation of all solutions, and then add the complexity of weights to arrive at our final algorithm.

7.1 Generating a Representation of All Solutions

To provide an intuition for our final algorithm, Figure 5 presents first a simpler extension: an algorithm that generates a representation of all solutions. Before we start the algorithm, we group all declarations by type and choose the one representative declaration and put it into the environment. The algorithm terminates because every used declaration is further excluded from the environment in the next step of the recursion. Furthermore, the set of new synthesized declarations whose types differ is finite. The algorithm effectively constructs a graph, whose unrolling could generate infinitely many solutions. For example, if we run the algorithm with $\Gamma = \{f : \tau_1 \rightarrow \tau_1, g : \tau_2 \rightarrow \tau_1, x : \tau_2, y : \tau_2\}$ and $\tau = \tau_1$, the algorithm will generate solutions

$$g(x), g(y), f(g(x)), f(g(y))$$

This representation identifies the relevant components that generate all of the infinitely many solutions

$$\{f^k(g(x)) \mid k \geq 0\} \cup \{f^k(g(y)) \mid k \geq 0\}$$

We will next present a method to extract a desired subset of these solutions according to given weights.

7.2 Generating Best N Solutions

Our final algorithm, presented in Figure 6, presents the search for best N solutions according to a given ranking function. The first phase of the algorithm computes the representation of all solutions, similarly to the previous algorithm in Figure 5; the result is stored in the variable graph. The function reconstruction then performs unfolding of the graph greedily, following partial expressions of minimal weight. Because of the properties of the weight function, this method detects the case when a complete expression has a weight smaller than the weight of all other partial expressions, and adds such complete expression to the resulting list.

```
INPUT:    type environment $\Gamma$ and type $\tau$
OUTPUT:   set of complex expressions with type $\tau$

TIP−ALL($\Gamma$, $\tau$) =
switch ($\tau$)
case $\emptyset \rightarrow \tau 1$:
  val R = { f:T | f:T $\in \Gamma$, T = $\{A_1, \ldots, A_n\} \rightarrow \tau 1$}
  if R == $\emptyset$ return $\emptyset$
  var RES = $\emptyset$
  foreach $r \in R$ do
    switch ($r$)
    case g : $\emptyset \rightarrow \tau 2$
      RES = RES $\cup$ {$g$}
    case g: $S \rightarrow \tau 2$
      foreach $B_i \in S$ do
        $e_i$ = TIP−ALL($\Gamma - \{r\}$, $B_i$)
      if ($\forall i.e_i \neq \emptyset$) RES = RES $\cup$ { g{$e_1, \ldots, e_m$} }
  return RES
case $S \rightarrow \tau 1$:
  val e = TIP−ALL($\Gamma \cup \Gamma_S$, $\tau 1$)
  if e == $\emptyset$ return $\emptyset$
    else return { Lambda($\Gamma_S$, e) }
```

Figure 5. An algorithm for generating a representation of all expressions that have a given type τ given the environment Γ

7.3 Weights for the Initial Type Environment

Previous subsection shows a search algorithm that is guided by a weight function. We next present the weight assignment strategy for declared symbols and their types as implemented in InSynth2. 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 most interesting aspect of the weight assignment is the assignment of weights to names of variables. Note that this assignment differentiates between different symbols of the same type and therefore would be neglected in a system that was only focusing on checking whether a type is inhabited, as opposed to finding actual inhabitants. The first factor in the weight of a name is the proximity of the declaration to the point where InSynth2 is invoked. We take the proximity into account by assigning weights as shown in Table 4. We assign the least weight to local symbols declared in the same method. We assign the weight at the next level to symbols defined in a class where a query is initiated. We assign an even higher weight to symbols in the same package. For an imported symbol x, we determine its weight using the formula in Table 4. Here $f(x)$ is the number of occurrences of x in the corpus, computed by examining syntax trees in a corpus of code (see Section 3.2 for the characteristic of the corpus we used for our experiments). We assign the highest weight to an inheritance conversion function that witnesses the subtyping relation.

Nature of Declaration or Literal	Weight
Local	5
Class	10
Package	15
Literal	400
Imported	$215 + \frac{785}{1+f(x)}$
Inheritance function	4000

Table 4. Weights for Names Appearing in Declarations

Scala declaration	Simply typed Lambda calculus declaration
$a : Int$	$a : Int$
$b : Map[String, Int]$	$b : Map(String, Int)$
$f : (x_1 : Int, x_2 : Boolean) : String$	$f : Int \to Boolean \to String$
$g : (x_1 : (Int \Rightarrow Char), x_2 : Long) : Boolean$	$g : (Int \to Char) \to Long \to Boolean$
class A **extends** B	$A <: B$

Table 3. Examples of the Scala to the simply typed Lambda calculus type declaration translation.

The weight of an expression, $w(e)$, is the sum of weights of all symbols that occur in the expression.

We assign weights to type variables as well. A type is represented by a term that may contain $\perp$ and $\to$ symbols, type constructors and primitive types. Their weights are given in Table 5. Additionally, the term may contains variables, to which we assign the weight of 2. A weight of a type, $w(\tau)$, is the sum of weights of all symbols that occur in the type term.

Initial Type & Constructors	Weight
$\perp$	1
$\to$	1
Primitive type	2
Type constructor	2

Table 5. Weights for Simple Types and Constructors

8. Related Work

We started this line of work with the first version of our tool, InSynth [6, 7, 16]. In the demo tool we used a theorem prover for classical logic for synthesis. Based on an extensive evaluation and various implementation improvements, we concluded that the code completion problem is more related to the type inhabitation problem. Furthermore, we now also provide a method to mine initial weights of declarations, which was very important for obtaining useful results.

Several tools including Prospector [13], XSnippet [19], Strathcona [9], PARSEWeb [24] and SNIFF [3] 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 InSynth2 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 InSynth2.

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 InSynth2 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 [19] 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 InSynth2, those examples can not be fitted into the code without additional interventions.

PARSEWeb [24] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and frequency. In InSynth2 the length of a returned snipped also plays an important role in ranking the snippetsm but InSynth2also has an additional component by taking into account also the proximity of derived snippets and the point where InSynth2 was invoked.

The main idea behind the SNIFF [3] 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. InSynth2 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 [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 [22] and our system is that Djinn generates a Haskell expression of a

```
Global variables :
  unpropagated = set of new requests that are not propagated
  propagated = set of propagated requests
  processed = set of answered request
  graph = the map that keeps answer
  N = the max number of solutions

TIP−Rank−ALL($\Gamma$, $\tau_{desired}$, N) =
  query = $\{\tau_{desired}\} \rightarrow \bot$
  goal = $\bot$
  var ws = { query }
  unpropagated = { Request($\bar{\Gamma}_{init}$, goal, query) }
  propagated = $\emptyset$
  processed = $\emptyset$
  graph = $\emptyset$
  while (ws $\neq \emptyset$) do
    val decl = best(ws)
    ws = ws − {decl}
    val decls' = propagateRequests(decl)
    ws = ws $\cup$ decls'
    processRequests(decl)
  end
  reconstruction (N, graph)

propagateRequests(decl) =
  var set = $\emptyset$
  for all request $\in$ { r | r $\in$ unpropagated, r == Request(_,_,decl)}
  switch (request)
    case Request($\bar{\Gamma}$, sender, receiver )
        switch (receiver )
      case r: S $\rightarrow \tau$
            foreach paramType $\in$ S
        switch (paramType)
          case S1 $\rightarrow \tau 1$
            $\bar{\Gamma}' = \bar{\Gamma} \cup \bar{\Gamma}_{S1}$
            val decls' = { f: T | f:T $\in \bar{\Gamma}'$
                      , T = S2 $\rightarrow \tau 1$}
            forall decl' $\in$ decls'
              val request' = Request($\bar{\Gamma}'$, receiver, decl')
            if (request' $\notin$ propagated)
              unpropagated = unpropagated $\cup$ request'
              set = set $\cup$ decls'
  unpropagated = unpropagated − { request }
  propagated = propagated $\cup$ { request }
  return set

processRequests(decl) =
  var decls' = $\emptyset$
  foreach request $\in$ {r | r $\in$ propagated,
                          r $\notin$ processed, r == Request(_,_,decl)}
  switch (request)
    case Request($\bar{\Gamma}$, sender, receiver )
        switch (receiver )
          case r: S $\rightarrow \tau$
                map = allParamsAnswered($\bar{\Gamma}$, S)
        if (map $\neq$ null)
          update(Key($\bar{\Gamma}$, $\tau$), Node(r, map))
          processed = processed $\cup$ request
          decls' = decls' $\cup$ { sender }
  foreach decl' $\in$ decls'
    processRequests(decl')
```

Figure 6. Constructive algorithm guided by weights for generating a ranked list of best N expressions of a given type τ

```
Definitions:
  best(WS) − Returns the declaration with the smallest weight.

  map.contains(key) − Returns 'true' if the map contains a
  mapping for the specified key.

  map.get(key) − Returns the value to which the specified key
  is mapped.

  map.replace(key, value) − Replaces any existing mapping for
  the specified key with the mapping that associates the key
  with the specified value.

  map.put(key, value) − Associates the specified value
  with the specified key in this map.

//auxiliary functions
update(key, node) =
  if (graph.contains(key))
    graph.replace(key, (graph.get(key) $\cup$ { node }))
  else
    graph.put(key, { node })

allParamsAnswered($\bar{\Gamma}$, paramTypes) =
    var map = $\emptyset$
    foreach(S $\rightarrow \tau$) $\in$ paramTypes
      val key = Key($\bar{\Gamma}$, $\tau$)
      if (graph.contains(key))
         graph.put(map,((S $\rightarrow \tau$), graph.get(key)))
      else
       return null
  return map
```

Figure 7. Definitions and auxiliary functions for the algorithm guided by weights

given type, but unlike our system it does not use weights to guide the algorithm and rank solutions.

Recently we have witnessed an increased 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, we work with higher order functions, and the returned code snippets can be arbitrarily nested and complex (no bound on the number and depth of arguments).

As having a witness term that a type is inhabited is a vital ingredient of our tool, one could think of InSynth2 as a prover for intuitionistic logic. One of the most effective modern intuitionistic theorem provers, Imogen [14], can reason about very expressive non-classical logic (such as linear logics). In contrast, InSynth2's prover is used for reasoning about the fragment of intuitionistic logic which is complete. InSynth2 provides additional functionality such as generating multiple solutions and ranking them.

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 is solvable [26]. The InSynth2 tool does not enumerate them but instead it returns several best ranked proofs.

9. Conclusions

We have presented the notion of quantitative type inhabitation, which searches for expressions of a given type in a type environment while minimizing a metric on the type binding. We implemented an algorithm supporting parametric types and subtyping and deployed it as a tool for suggesting expressions within an IDE for Scala. The synthesized expressions can combine all declared values, fields, and methods that are in the scope at the current program point, so the problem is closely related to the problem of type inhabitation in type systems. Among the key results is a weight-driven version of the theorem proving algorithm, which uses proximity to the declaration point as well as weights mined from a corpus to prioritize among the declarations to consider and sort the solutions. We have deployed the algorithm in an IDE for Scala. Our evaluation on synthesis problems constructed from Java 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] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda - a functional language with dependent types. In *TPHOLs*, pages 73–78, 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] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. FASE '09, pages 385–400, 2009.

[4] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. In *Proceedings of the 31st international conference on Mathematical Foundations of Computer Science*, MFCS'06, pages 1–23. Springer-Verlag, 2006. ISBN 3-540-37791-3, 978-3-540-37791-7.

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

[6] T. Gvero, V. Kuncak, and R. Piskac. Code completion using quantitative type inhabitation. Technical Report EPFL-REPORT-170040, EPFL, July 2011. `http://infoscience.epfl.ch/record/170040`.

[7] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[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*, pages 230–244, 2009.

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

[16] R. Piskac. Decision procedures for software verification and code synthesis. In *Eight-Minute Presentations at POPL'11, ACM SIGPLAN Symp. Principles of Programming Languages*, January 2011.

[17] J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, pages 169–183, 2011.

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

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

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

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

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

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

[24] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. ASE '07, pages 204–213, 2007.

[25] P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In P. de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 373–389. 1997.

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