Code Completion using Quantitative Type Inhabitation

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. The technique generates expressions by taking into account 1) polymorphic type constraints of the values in scope, 2) the API usage patterns in a corpus of code, and 3) any available test cases. It supports polymorphic type declarations and can synthesize expressions containing methods with any number of arguments and any depth. Our synthesis approach is based on a quantitative generalization of the type inhabitation problem with weighted type assignments. Weights indicate preferences to certain type bindings; they guide the search and enable the ranking of solutions. We present a new polynomial-time algorithm for a restricted version of quantitative type inhabitation, as well as a complete semi-decision procedure for the general case of generic types. We identify a simple method to handle subtyping by introducing coercion functions and then erasing them in the final expressions. We have implemented our technique and evaluated it on over 100 examples taken from the Web. The system was remarkably effective in reinventing the erased expressions from the (previously unprocessed) code and ranking these expressions among the top suggestions for the developer. Our overall experience indicates that this approach to synthesizing and suggesting code fragments goes beyond currently available techniques and is a useful functionality of software development environments.

1. Introduction

Libraries are one of the biggest assets for today's software developers, enabling developers to build on the shoulders of their predecessors. 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 [20] and IntelliJ [9] 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 an interesting general direction of improving modern IDEs: introduce the ability to synthesize 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 explicitly. Instead, we devise a more ambitious approach that considers all values in scope as the candidate leaf values of expressions to be synthesized.

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 (more flexible solutions are also possible based on type inference). 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 basic version of the type inhabitation problem is well understood (although not sufficient by itself to solve the practical problem that we consider). In the absence of parametric polymorphism, the problem can be seen as the type inhabitation in the simply typed lambda calculus, which is decidable and PSPACE-complete. If we add finite intersection types to the language, the type inhabitation becomes an EXPSPACE-complete problem [14].

The synthesized code is extrapolated 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.

Going beyond the past work that inspired us [11], we aim to go further and also support parametric types (generics). Parametric types are widely used in modern libraries, including the Scala library that is of immediate interest to us. In the presence of parametric polymorphism, the type inhabitation problem becomes undecidable. The corresponding theorem proving problem is checking type inhabitation for the Hindley-Milner type system, without nested type quantifiers. The first component of our solution is a search algorithm that solves a quantitative version of the type inhabitation problem, i.e. finds the most relevant type inhabitant. The implementation uses techniques from resolution-based theorem proving, including unification and data structures for avoiding redundant proof steps. Despite the undecidability of the problem, our is fairly successful in finding a proof for the code synthesis problems that we encounter. In fact, the main problem is not the difficulty of finding one solution, but rather the fact that there are so many solutions, among which we need to select those that are most likely to be helpful.

Another contribution of our work comes from combining this 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.

Our proof rules naturally combine weights of premises to determine the weight of the conclusion, and ensure that very long proofs result in terms with a very large weight. In addition, the symbols in current clauses (derived type) contribute to the weight. The use of proof length in that weight ensures a form of relative completeness: weight prioritization does not prevent the tool from finding proofs that an exhaustive application of proof rules would find. The particular weights of symbols, however, are an important factor for the quality of generated results. To estimate these initial weights 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, QTI within a Scala IDE. We used a corpus of 18 open source Scala projects as well as the standard Scala library to collect the usage statistics for the initial weights of declarations. We run QTI on more than 100 examples from the Web written to illustrate Java API usage (we translated these examples from Java to Scala). 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 at the first place in the list. To estimate the interactive nature of QTI, 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 QTI 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

- We propose a new code generation feature for IDEs, and show that it closely corresponds to the type inhabitation problem.
- We produce an entire list of code snippets, ranked according to weights. We extend the rule system with rules that manipulate weighted formulas.
- We use off-line analysis on a number of Scala open source projects to compute the initial weights used in synthesis. We also propose a simple but effective policy of giving higher priority to declarations closer to the cursor.
- We observe that the quality of results can be improved by using a basic continuous-testing mechanism, which runs the synthesized code snippets and removes those that crash or fail user-supplied test cases.
- We implemented all proposed techniques in the QTI tool, including a resolution-style technique, with unification and support for weights and multiple proof results, encoding of Scala declarations, the mining of the usage of declarations in a corpus, and test-based filtering. We integrated these techniques with the Ensime IDE for Scala and the underlying Scala presentation compiler.
- We evaluate QTI on a number of examples from the Web meant to illustrate API usage. The evaluation shows that QTI in many cases synthesizes expected solutions and ranks them reasonably high in the list of offered choices. We evaluate the impact of individual techniques that we employ and show that they are often necessary to achieve sufficiently high quality of results. For further evaluation, QTI is publicly available for download from our web site (the link is provided to the PC chair as the non-anonymous material).

Paper outline. We start by presenting the examples to provide the reader with an idea of how QTI works. We believe that QTI provides a new level of functionality and performance compared to all existing solutions. We therefore start with our experimental results that provide evidence of the effectiveness of QTI. We then present algorithms for quantitative type inhabitation, first for the ground case, then for the case of generics. We then describe implementation aspects of QTI and present further experimental results that show why the particular choice of techniques in QTI is important for its effectiveness. We then discuss related work and conclude.

2. Example

We next illustrate the functionality of QTI through several examples, primarily code from the online repository of Java API examples `http://www.java2s.com/`, as well as an example with generalized algebraic data types.

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

If we invoke QTI 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 QTI only needs the current program context, and does not require additional information from the user. QTI is able to use both imported values (such as the constructors in this example) and locally declared ones (such as body and sig). QTI supports methods with multiple arguments and synthesizes expressions for each argument.

Another feature of QTI is filtering out code fragments that fail to pass given test cases. Suppose that the developer provides a simple test-case that invokes the test method. Then QTI runs the test case and removes the code suggestions that do not terminate successfully in a short amount of time. This removes the fifth option above, because it blocks on the input. As a result, the options change, and the new fifth suggestion becomes:

```
new SequenceInputStream(new FileInputStream(sig),
    new SequenceInputStream(new FileInputStream(sig),
                            new FileInputStream(sig)))
```

Completing Expression in a Type-Safe Evaluator. As our next example consider the following simple type-safe evaluator, distributed as an example with Scala. The example shows how QTI integrates into a Scala IDE and supports several Scala features. The evaluator function eval returns the values of an algebraic data type that encode expressions. Different sub-expressions can have either integer or boolean type, and the declarations encode this precisely thanks to the ability to extend an instance of a supertype and recover this information during pattern matching.

```
abstract class Term[T]
case class Lit(x: Int) extends Term[Int]
case class Succ(t: Term[Int]) extends Term[Int]
case class IsZero(t: Term[Int]) extends Term[Boolean]
case class If[T](c: Term[Boolean],
                 t1: Term[T], t2: Term[T]) extends Term[T]
def eval[T](t: Term[T]): T = t match {
  case Lit(n) ⇒ n
  case Succ(u) ⇒ eval(u) + 1
  case IsZero(u) ⇒ eval(u) == 0
  case If(c, u1, u2) ⇒ eval(if (■) u1 else u2)
}
```

The code has an erased argument of the **if** expression and this is where the developer has positioned the editor cursor ■. At this point, the developer invokes QTI to obtain suggestions for possible snippets. As the first two choices, QTI offers a selection window with the following expressions:

```
1. eval[Boolean](c)
2. false
```

The first expression turns out to be appropriate to insert at the cursor, and completes the type-safe evaluator. Overall, QTI often finds an expected completion among the top few choices.

List operations. Consider next the problem of finding an iterator in a generic ArrayList[E] class. The following code is used to demonstrate how to use iterators.

```
//java2s.com/Code/JavaAPI/java.util/ArrayListiterator.htm
class MainClass {
  def main(args:Array[String]) {
    var al:ArrayList[String] = new ArrayList[String]()
    al.add("A")
    al.add("B")
    var itr:Iterator[String] = ■
    while (itr.hasNext()) {
      val element = itr.next() //itereates over elements in "al"
    }
  }
}
```

Note that handling this example requires simultaneous support for parametric polymorphism and for subtyping, because the type declarations are given by the following code.

```
class ArrayList[T] extends AbstractList[T] with List[T]
   with RandomAccess with Cloneable with Serializable { ... }
abstract class AbstractList[E] extends AbstractCollection[E]
  with List[E] {
  ...
  def iterator():Iterator[E] = {...}
}
```

The Scala compiler has access to the information about generics from Java libraries. QTI supports both generics and subtyping and in 502 milliseconds returns a number of solutions among which the first one is the desired expression al.iterator(). While doing so, it examines 1524 declarations, as shown in Table 2, benchmark 13.

Stream Tokenizer. Consider the task of generating a stream tokenizer. This example examplifies some of the typical complexity of the Java API for I/O.

```
//java2s.com/Code/JavaAPI/java.io/newStreamTokenizerReaderr.htm
import java.io._
class Main {
  def main(args:Array[String]) = {
    var tf:StreamTokenizer = ■
    ...
  }
}
```

The system suggests several suggestions, and the one ranked number three turns out to be exactly the one expected in the example from the Web, namely

```
new StreamTokenizer(new BufferedReader(
   new InputStreamReader(System.in)))
```

The effectiveness in the above examples is due to several aspects of QTI. In this particular example, given a budget of 0.5 seconds, QTI explores over 7000 intermediate expressions, and finds over 380 examples of the expected StreamTokenizer type, as shown in Table 2, benchmark 1. QTI 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. QTI learns weights from a corpus of declarations, assigning lower weight (and thus favoring) declarations appearing more frequently.

3. Evaluation of the Effectiveness of QTI

We implemented QTI and evaluated it on over 100 examples. This section evaluates the effectiveness of QTI, showing that the techniques we developed and implemented result in a useful tool, appropriate for interactive use within an integrated development environment (concretely, the Ensime tool running inside emacs). Later, in section 8.3, we will present further experimental analysis to measure the importance of individual techniques deployed within QTI.

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. In some cases, 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 [17]. 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 QTI web site.

When we invoke QTI, it returns five recommended expressions. We call a run successful if the expression that was removed from the example code appears among these five expressions. We run QTI using a time limit of 0.5 seconds for the core quantiative type inhabitation engine; the table (and our experience) shows that the overall response time remains below 0.7 seconds. By using a time limit, we aim to evaluate the usability of QTI in an interactive environment.

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 8.2. 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

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.

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. QTI 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 QTI in its optimal configuration to recover 100 removed expressions from benchmarks (see Table 2 for a subset of our results). The results show that the desired expression appears in the top 5 snippets (suggested expressions) in 76 benchmarks (76%). It appears as the top snippet (with rank 1) in 53 benchmarks (53%). In several benchmarks we generalized the import statements, by importing the entire API packages instead of methods from a single class. As expected, due to a larger number of initial declarations the percentage of recovered snippets dropped. We were able to recover the expected expression in 48 benchmarks (48%); the expected expression was ranked as number one in 38 benchmarks (38%). Note that our corpus (Section 3.2) is disjoint (and somewhat different in nature) from the examples on which we performed the evaluation.

Additionally, we found 25 examples where it was natural to remove multiple lines of code at once. (In the remaining examples either the initialization was done in one line, or the intermediate variables were used in multiple later locations, so completing a single declaration would never produce valid code.) We asked QTI to recover a single expression that subsumes all these lines, constructing also sub-expressions that were explicitly assigned to intermediate local variables in the original code. QTI managed to suggest the expected snippet (within top 5 choices) in 13 out of 25 benchmarks (52%). The results suggest that QTI can synthesize expected expressions in useful pieces of software.

Table 2 presents the results, in more details, on 26 benchmarks out of 100 benchmarks that we examined. of the more interesting examples, on which QTI performs well. The length column represents the number of declarations in the expected expression. The "Initial" column is the number of initial type declarations that QTI extracts at a given program point and gives to the search procedure. The "Derived" column is the number of intermediate expressions generated during the search. The "Expression" column is the number of expressions of the desired type that the prover found within its time limit. Time includes declaration loading, encoding and weight assignment time, as well as the time within the prover (which was set to 0.5 seconds). QTI was able to synthesize expected expressions in all these benchmarks. We therefore measured the times for QTI to reach the expression that was expected; we found that this time ranges from below 1 to 219 millisecond.

Note that the numbers of the initial declarations range from tens to a little over thousand, whereas the number of derived bindings ranges up to over ten thousand, all generated within half of a second. As expected, a larger set of initial declarations generally leads to a larger set of clauses derived, but the exact number depends on the specific types involved.

The last 14 benchmarks illustrate the performance of QTI in the case of generic types. The number of expressions of derived type in this case is small compared to the generalized benchmarks, so the rank of the desired expression is high.

In summary, the expected snippets appear among the top 5 solutions in many examples, despite the fact that our algorithm often generates hundreds of expressions of the desired type.

4. Inhabitation for Ground Types

For the main question of finding a code snippet for the given type, there is a similar problem in type theory, so-called the *type inhabitation problem*. In this section we recall some basic definitions and facts from type theory and we establish a connection between the type inhabitation problem and the problem of finding code snippets.

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

An expression $\Gamma \vdash e : \tau$ denotes that from an environment Γ we can derive a type assumption $e : \tau$ by applying rules of some calculus. The type inhabitation problem for the given calculus is stated as: given a type τ and a type environment Γ, does there exist an expression e such that $\Gamma \vdash e : \tau$.

DEFINITION 4.1 (Ground Types). *Let C be a fixed finite set. For every $c \in C$, with $c/_n$ we denote the arity of the element. The elements of arity 0 are called constants. The set of all ground types T_g is defined by the grammar:*

$$T_g ::= C(T_g, \ldots, T_g) \mid T_g \to T_g$$

To establish a connection between T_g and the Scala types, one could consider the set C as a set containing the Scala primitive types (such as `Int` or `String`) and type constructors (such as `List`$/_1$, `Map`$/_2$).

Let S be a set containing function symbols. The set of all ground terms E_g is formed inductively from S as follows: all constants of S are ground terms. If $t_1, \ldots, t_n$ are ground terms and $f/_n \in S$, then $f(t_1, \ldots, t_n)$ is a ground term.

Figure 1 lists the rules of a calculus for the ground types. We call this calculus *the ground applicative calculus*. It supports the application of a function to a term, and the function composition. Those two rules have a natural interpretation in a programming language. Through the application we construct a snippet, where a method is applied on its argument, while the composition represents a combination of several methods.

Benchmarks (ground, then generic)		Length	# Initial	# Derived	# Expressions	Solution Rank	Time [ms]
1	StreamTokenizerReaderr	4	555	7416	387	3	586
2	BufferedReaderInputStreamReader	3	554	7699	402	1	564
3	BufferedReaderReaderin	4	82	2192	372	1	530
4	ByteArrayInputStreambytebufintoffsetintlength	4	63	4533	225	3	537
5	CharArrayReadercharbuf	3	78	1515	300	1	546
6	DataInputStreamFileInputStreamfileInputStream	3	554	7180	341	3	573
7	FileReaderFilefile	3	555	6028	283	2	572
8	PipedReaderPipedWritersrc	2	554	7196	338	3	579
9	ServerSocketintport	2	1038	10051	254	2	622
10	TimerintvalueActionListeneract	3	63	8362	1	1	538
11	TransferHandlerStringproperty	2	659	5537	230	1	629
12	URLStringspecthrowsMalformedURLException	3	1038	8830	208	1	620
13	ArrayListiterator	2	79	1524	10	1	502
14	ArrayListtoArray	2	76	1587	2	2	655
15	HashMapentrySet	2	68	15822	2	1	571
16	HashMapvalues	2	152	3632	1	1	584
17	HashSetiterator	2	104	2501	8	1	570
18	Hashtableelements	2	64	2849	2	2	527
19	HashtableentrySet	2	65	6171	2	1	541
20	HashtablekeySet	2	65	2298	1	1	556
21	Hashtablekeys	2	58	2392	2	1	529
22	PriorityQueuepoll	2	77	2267	179	1	549
23	TreeMapentrySet	2	88	3227	2	1	601
24	TreeMapvalues	2	87	798	1	1	547
25	Vectorelements	2	88	2392	5	1	531
26	VectortoArray	2	87	2074	2	2	526

Table 2. Measuring Overall Effectiveness.

$$\textsc{Axiom}\ \frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau}\qquad \textsc{App}\ \frac{\Gamma\vdash f:\tau_1\to\tau_2\qquad \Gamma\vdash x:\tau_1}{\Gamma\vdash f(x):\tau_2}$$

$$\textsc{Compose}\ \frac{\Gamma\vdash f_1:\tau_0\to\tau_1\qquad \dots\qquad \Gamma\vdash f_n:\tau_{n-1}\to\tau_n}{\Gamma\vdash f_n\circ\ldots\circ f_1:\tau_0\to\tau_n}$$

Figure 1. Calculus for the Ground Types

4.1 Type Inhabitation in the Ground Applicative Calculus

The problem of type inhabitation is widely studied for various calculi. However, very often this problem is undecidable. The ground applicative calculus can be seen as a sub-calculus of the simply typed lambda calculus, which additionally contains the lambda abstraction. In the simply typed lambda calculus the type inhabitation problem is decidable, but very hard. By reduction to the canonical quantified Boolean formula (QBF) problem, it was shown in [18] that the problem is PSPACE-complete. In this section we show that if the lambda abstraction is disabled, the type inhabitation problem can be solved much faster.

THEOREM 4.2. *The type inhabitation problem in the ground applicative calculus can be solved in polynomial time.*

Proof Let Γ be a type environment $\Gamma = \{e_1 : \tau_1, \ldots, e_n : \tau_n\}$, with $e_i \in E_g$ and $\tau_i \in T_g$. Let τ_0 be a type for which we ask if there is an expression e_0 such that $\Gamma \vdash e_0 : \tau_0$. We encode the query as the type assumption $\mathsf{goal} : \tau_0 \to \bot$, where $\bot$ is a designated symbol, previously unused. The goal of an algorithm is to derive a type assumption $e : \bot$. The expression e can only be of the form $\mathsf{goal}(x)$ and the term x has the desired type τ_0.

Let $\mathsf{TParts}(\Gamma)$ denote the set of all types appearing in Γ, together with τ_0. In addition, if the type is not a constant, then $\mathsf{TParts}(\Gamma)$ also contains all its subterms of the terms representing types. The set $\mathsf{TParts}(\Gamma)$ is clearly finite and polynomial in the size of Γ.

We consider a type derivation sequence of type assumptions that starts with an enumeration of Γ and continues with the application of inference rules until reaching a type judgment of the form $e_0 : \tau_0$. We show that if there is a term e_0 such that $\Gamma \vdash e_0 : \tau_0$, then it can be derived in polynomial time. For this purpose we can assume that each step produces a term that is non-redundant, that is, it is subsequently used in the derivation (otherwise we could eliminate it).

We first assume that there is no COMPOSE rule, so we only apply the APP rule. In that case each derived term has a type from $\mathsf{TParts}(\Gamma)$. The set of derived types does not change if we always adopt the following principle: never use in premises elements $t : \tau$ of a type derivation sequence if there is a term $t' : \tau$ with the same type appearing earlier in the sequence. If we adopt this policy, the number of newly introduced elements is bounded by $|\mathsf{TParts}(\Gamma)|^2$. Therefore, the process terminates. The resulting sequence also gives a representation of the (possibly infinite) set of terms that have given type. The infinite sets of solutions appear precisely from derivations that use a term of some type to derive a new term of the same type. However, the policy described ensures that such loops are detected and not followed.

We next assume that we can also use the composition rule. This problem does not reduce to the case of application because viewing $\circ$ as a higher-order function would require assigning it a polymorphic type. Nonetheless, we show that we also obtain a polynomial bound.

First we observe that, if the COMPOSE rule is used to obtain a term of the form $(f_1 \circ \ldots \circ f_n)(x)$ then it was not necessary for producing a new type assumption: we can instead directly use APP alone to construct $f_1(\ldots f_n(x)\ldots)$. We next use the fact that the COMPOSE rule already accounts for any number of function symbols, so it is not necessary to use the result of a compose rule again. From those observations we conclude that COMPOSE always produces either an argument of APP or the required type τ_0. In the second case, by using the APP rule, we derive an inhabitant of the $\bot$ type, i.e. that COMPOSE again produced an argument of APP.

We can therefore replace COMPOSE rule with the following APPCOMPOSE rule, in a process similar to completion in term rewriting. This results in the following system, which again has the crucial property that its result is always an element of $\mathsf{TParts}(\Gamma)$:

$$\textsc{App}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f(x) : \tau_2}$$

$$\textsc{AppCompose}\ \frac{\begin{array}{c}\Gamma \vdash c : (\tau \to \tau_n) \to \sigma \\ \Gamma \vdash f_1 : \tau \to \tau_1 \qquad \dots \qquad \Gamma \vdash f_n : \tau_{n-1} \to \tau_n\end{array}}{\Gamma \vdash c(f_n \circ \ldots \circ f_1) : \sigma}$$

Therefore, application of such rules also finishes in at most $|\mathsf{TParts}(\Gamma)|^2$ steps. This completes the proof that type inhabitation problem where we restrict terms to be obtained from application and function composition is polynomial.

5. Quantitative Applicative Ground Inhabitation

Given a type environment Γ and a type τ_0, we encode the type inhabitation problem by adding to Γ the type assumption goal : $\tau_0 \to \bot$ and then directing the search towards the inhabitants of the type $\bot$. However, there might be many terms belonging to a given type, and the question of finding the best term naturally arises. We address this problem by assigning a weight to every expression. Similar to resolution-based theorem proving, a lower weight indicates the higher relevance of the term.

We compute the weights of types and expressions under the assumption that there is an initial, pre-computed non-negative weight assigned to every symbol.

DEFINITION 5.1. *Let w be a weight function defined on the type and expression symbols. First, a weight of an expression e, $w(e)$, is the sum of weights of all symbols that occur in the expression. Second, a weight of a type τ, $w(\tau)$, is the sum of wights of all symbols that occur in the type term. Finally, a weight of a type assumption $e : \tau$ is $w(e : \tau) = w(e) + w(\tau)$.*

A more detailed description of how the initial weight function is derived is given in Section 8.2.

In this section we further extend the type inhabitation problem with the additional requirement to find an expression of the minimal weight.

THEOREM 5.2. *For a type environment Γ and a type τ_0 in the ground applicative calculus it is possible to find in polynomial time an expression e of type τ_0, such that the weight of e is smaller or equal to the weight of all other expressions of the type τ_0.*

Proof The proof extends the proof of Theorem 4.2. It builds a sequence of type assumptions that can be derived from Γ. To every element τ of $\mathsf{TParts}(\Gamma)$ we assign a pair (n, t) where n is the minimum weight of all terms of type τ, which are currently in the sequence, and t is an expression such that $w(t) = n$. Initially, we assign $(\infty, -)$ to all elements of $\mathsf{TParts}(\Gamma)$. As before, we construct a sequence of type assumptions. With every type assumption $e : \tau$ added to the sequence, we recalculate the annotation of τ. If its current minimum weight is strictly greater than $w(e : \tau)$, then the new annotation becomes $(w(e), e)$. In the sequence we also replace every occurrence of the expression e' of the type τ by e. We can do such a replacement safely, since e' does not appear in the derivation of e (otherwise it would not hold $w(e') > w(e)$). We continue with the enumeration of the derived typed assumptions as in the proof of Theorem 4.2, using the same restrictive principle about the type assumptions that can participate in the derivations. Applying the same arguments we prove that it is possible to find a term of the minimum weight in polynomial time.

$$\textsc{App}\ \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1'}{\Gamma \vdash f(x) : \tau_2'} \quad \begin{array}{c}\sigma = \mathsf{mgu}(\tau_1, \tau_1') \\ \tau_2' = \sigma(\tau_2)\end{array}$$

$$\textsc{Compose}\ \frac{\begin{array}{c}\Gamma \vdash f : \tau_1 \to \tau_2 \\ \Gamma \vdash g : \tau_0 \to \tau_1'\end{array}}{\Gamma \vdash (f \circ g) : \tau_0' \to \tau_2'} \quad \begin{array}{c}\sigma = \mathsf{mgu}(\tau_1, \tau_1') \\ \tau_0' = \sigma(\tau_0) \\ \tau_2' = \sigma(\tau_2)\end{array}$$

Figure 2. Rules for Generic Types used by Our Algorithm

```
INPUT: $\Gamma_0$ - environment at program point
INPUT: $\tau_G$ - desired type
OUTPUT: res - set of resulting expressions $e$ with $\Gamma_0 \vdash e{:}\tau_G$
Definitions:
  $w(e{:}\tau) := w(e) + w(\tau)$
  $\mathsf{bestT}(\tau, \Gamma) := \{(e{:}\tau) \in \Gamma \mid (\forall (e'{:}\tau) \in \Gamma.\ w(e) \le w(e'))\}$
  $\mathsf{bestT}(e'{:}\tau', \Gamma) := \mathsf{bestT}(\tau', \Gamma)$
  $w(\mathsf{bestT}(b, \Gamma)) = w(b)$, if $\exists b \in \mathsf{bestT}(b, \Gamma)$, $+\infty$ otherwise
  $\mathsf{best}(\mathsf{q}) := \{b \in \mathsf{q} \mid \forall b' \in \Gamma.\ w(b) \le w(b')\}$
  $\mathsf{cmpt}(\tau_1, \tau_2)$ := an mgu in APP or COMPOSE of Figure 2 exists

Code:
  $\Gamma = \Gamma_0 \cup \Gamma_{\mathsf{Comb}} \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}$
  $\mathsf{q} = \Gamma$
  $\mathsf{res} = \emptyset$
  while $\neg\mathsf{timeout} \land \mathsf{q} \neq \emptyset$ do
    let $(e_1{:}\tau_1) \in \mathsf{best}(\mathsf{q})$
    $\mathsf{q} = \mathsf{q} \setminus \{(e_1{:}\tau_1)\}$
    for all $(e_2{:}\tau_2) \in \{(e_2{:}\tau_2) \in \mathsf{bestT}(\tau_2, \Gamma) \mid \mathsf{cmpt}(\tau_1, \tau_2)\}$ do
      $\mathsf{derived} = \mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2) \cup \mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$
      $\mathsf{res} = \mathsf{res} \cup \{e' \mid (e : \bot_{fresh}) \in \mathsf{derived}, e[\mathsf{G} := \mathsf{I}] \overset{\mathsf{I}(t) \to t}{\rightsquigarrow^*} e'\}$
      $\mathsf{q} = \mathsf{q} \cup \{b \in \mathsf{derived} \mid w(b) < w(\mathsf{bestT}(b, \Gamma))\}$
      $\Gamma = \Gamma \cup \mathsf{derived}$
    end for
  end while
```

Figure 3. The Search Algorithm for Quantiative Inhabitation for Generic Types

6. Quantitative Inhabitation for Generics

This section presents our algorithm for type inhabitation in the presence of generic (parametric) types as in the Hindley-Milner type system, without nested type quantifiers. We represent type variables implicitly (as in resolution for logics with variables). Figure 2 shows the rules for application, as well as the rule for composition (which we introduce to improve performance by making derivations shorter). We assume the axiom rule that $\Gamma \vdash (x{:}\tau)$ for $(x{:}\tau) \in \Gamma$. We will ensure that combinators, that can be used to simulate the effects of lambda abstraction in a calculus with application, belong to the initial environment, with their polymorphic types, and denote their set by Γ_{Comb}. For example, we can use

$$\{\mathsf{K}{:}\alpha \to \beta \to \alpha, \mathsf{S}{:}(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\}$$

This makes the application rule complete for the purpose of finding a term of a given type, thanks to the translation from lambda calculus to combinatory logic. We therefore omit the lambda abstraction rule. This approach is also used in [14], but for a non-generic type system with intersection types.

Description of the algorithm. Figure 3 shows the algorithm that systematically applies rules in Figure 2, while avoiding cycles due to repeated types whose terms have non-minimal weights. The algorithm maintains two sets of bindings (pairs of expressions and their types): Γ, which holds all initial and derived bindings, and q, which is a work list containing the bindings that still need to be processed. At the start of algorithm, Γ contains the initial declarations, as well as the combinators and the goal encoded as $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$ where τ_G is the type for which the user wishes to generate expressions. The work list initially contains all these declarations as well. The algorithm accumulates the expressions of the desired type in the set res. The main loop of the algorithm runs until the timeout is reached or the work list q becomes empty.

The body of the main loop of the algorithm selects a minimal (given by $\mathsf{best}(\_)$) binding $(e_1{:}\tau_1)$ from the work list q and attempts to combine it with all other bindings in Γ for which the types τ_1 and τ_2 can be unified to participate in one of the inference rules (we denote this condition using the $\mathsf{cmpt}(\tau_1, \tau_2)$ relation). Note, however, that there is no point in combining $(e_1{:}\tau_1)$ with a $(e_2{:}\tau_2)$ if there is another $(e_2'{:}\tau_2)$, with the same τ_2 but with a strictly smaller $w(e_2')$. Therefore, the algorithm restricts the choice of $(e_2{:}\tau_2)$ to those where $w(e_2)$ is minimal for a given τ_2. We formalize this using the function $\mathsf{bestT}(\tau_2, \Gamma)$ that finds a set of such bindings with minimal e_2. We also extend the function to accept a candidate e_2' (which is ignored in looking up the minimal e_2). Moreover, we define $w(\mathsf{bestT}(\tau_2, \Gamma))$ to denote the value of this minimum (if it exists).

The sets $\mathsf{App}(e_1{:}\tau_1, e_2{:}\tau_2)$ and $\mathsf{Comp}(e_1{:}\tau_1, e_2{:}\tau_2)$ are results of applying the rules from Figure 2. If no rule can be applied the result is the empty set. We use derived to denote the set of results of applying the inference rules to selected bindings. These results may need to be processed further and therefore the algorithm may need them into q. However, it avoids doing this if the derived binding has a type that already exists in Γ and the newly derived expression does not have a strictly smaller weight. This reduces the amount of search that the algorithm needs to perform.

Because of the declaration $(\mathsf{G}{:}\tau_G \to \bot_{fresh})$, the algorithm detects expressions of type τ_G using the expressions e of fresh type $\bot_{fresh}$. To obtain the expression of the desired type, we replace in e every occurrence of G with the identity combinator I. This is justified because $\bot_{fresh}$ is a fresh constant, so replacing it with τ_G in a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \bot_{fresh})\}(e{:}\bot_{fresh})$ yields a derivation of $\Gamma \cup \{(\mathsf{G}{:}\tau_G \to \tau_G)\} \vdash (e{:}\tau_G)$, in which we can use I instead of G. The algorithm also simplifies the accumulated expressions by reducing I where possible. In the presence of higher-order functions I may still remain in the expressions, which is not a problem because it is deducible from any complete set of combinators.

Finally, under the assumption that a linear weight function is given, and the weight of each expression symbol is strictly positive, it is straightforward to see that the algorithm finds the derivations for all types that can be obtained using the rules from Figure 2. Indeed, the weight of an expression strictly increases during the derivation, so an algorithm, if it runs long enough, reaches arbitrarily long value as the minimum of the work list. This shows that the algorithm is complete.

7. Subtyping using Coercions

A powerful method to model subtyping is to use coercion functions [2, 10, 15]. This approach raises non-trivial issues when we perform type checking or type inference, but becomes simple and natural if the types are given but we search for the terms.

Simple conversions. In the absence of variant constructors and type bounds, we can model the subtyping relation $A <: B$ by the existence of a coercion expression $c{:}A \to B$. For example, if a class $A[\vec{T}]$ with type parameters $\vec{T}$ extends or mixes-in another class $B[\vec{\tau}(\vec{T})]$, we introduce into the environment a conversion function $c{:}A[\vec{T}] \to B[\vec{\tau}(\vec{T})]$. Note that the composition of coercion functions immediately accounts for the transitivity of the subtyping relation.

Example revisited. Let us now reconsider the ArrayList[T] example from Section 2. Because ArrayList[T] extends AbstractList[T], and AbstractList[E] extends AbstractCollection[E] we generate two coercion functions:

c1 : ArrayList[α] $\to$ AbstractList[α]
c2 : AbstractList[β] $\to$ AbstractCollection[β]

There is a declared member of AbstractList[E] with the following type (including the received object):

iterator : AbstractList[γ] $\to$ Iterator[γ]

The local variable declaration in the main method of the example yields the binding

al : ArrayList[String]

The goal in the example is to find an expression of type Iterator[String], which yields the declaration

τ_G : Iterator[String] $\to \bot_{fresh}$

Using rules in Figure 2, the algorithm in Figure 3 unifies the type variables and ground type of String and derives in Γ the type binding

τ_G(iterator(c1(al))) : $\bot_{fresh}$

This produces I(iterator(c1(al))) in the res variable of the algorithm in Figure 3, and then reduces to iterator(c1(al)). Finally, we erase all conversion functions and obtain iterator(al), which is displayed to the user as the Scala code al.iterator().

The subtyping constructor. In the presence of variant and contravariant subtyping rules as well as for bounded type parameters, we need more control over the subtyping relation, so we introduce a binary subtype constructor Sub[A,B] to represent A <: B. The translation of function signatures then corresponds to dictionary translation for type classes [13, 22]. The subtyping is applied using a polymorphic constant

$$\mathsf{subApp} : \mathsf{Sub}[\alpha, \beta] \to \alpha \to \beta$$

A covariant subtyping of a List constructor can then be expressed through a constant

$$\mathsf{listCo} : \mathsf{Sub}[\alpha, \beta] \to \mathsf{Sub}[\mathsf{List}[\alpha], \mathsf{List}[\beta]]$$

We witness the reflexivity of subtyping through a constant of type Sub[α, α] and express transitivity also explicitly through a polymorphic constant. We encode bounded quantification in function signatures by adding extra dictionary-like parameters that express type bounds.

8. Implementation

QTI extends Ensime, an emacs plugin that serves as a development environment for Scala. Ensime accesses the Scala presentation compiler to extract abstract syntax trees (ASTs) of Scala programs. We use the ASTs both for offline analysis to collect data from open source projects and for the online analysis, to extract the type declarations visible at the current program point. Ensime also enables us to present ranked snippets to a user. QTI presents them in a drop-down list, where the user has the option to choose the one that she prefers. After the user makes her choice, QTI inserts the chosen snippet at the cursor position.

We produce snippets while supporting following Scala features: higher order functions, sub-typing, monomorphic and polymorphic types. In order to generate those snippets, our on-line analysis has four key components:

- A loader that extracts a desired type and visible declarations, assigns them weights and encodes them into formulas.

- A search algorithm that takes formulas and finds a proof(s), using the weight driven search algorithm.
- A reconstructor that reconstructs Scala code snippet(s) from proofs.
- A test filter that filters snippet candidates.

If a user initiates a query at the place of a local variable initializer of a method m, QTI starts by collecting all local variables visible from this point. It adds m's parameters if they exist. Next, it collects all declared methods, fields and class parameters in the class C that contains the method m, including those present in super-classes. It also includes the declaration of m itself, supporting expressions with recursive calls. The query to QTI can be also initiated at a place of a field initializer. If the query is initiated for field f in class C, QTI collects all declarations in C except f itself. (The reason is that f is not visible for the initializer.) This also applies if m or f were defined in a Scala object (a singleton class). Next, QTI collects all public methods and fields from the package where C is defined. Finally, it collects all public methods and fields from imported classes and objects. We also allow a user to initiate a query at a place where an **if** condition appears, knowing that the desired type is `Boolean`. For types that are used often, such as `Int`, `Boolean` and `String` we introduce custom declarations for sample constants of this type. The user can, of course, modify the particular values generated subsequently in the code. The declarations associated with constants have large weight, so they appear if no declaration that has these types is found or if such declaration also has a vary large weight.

8.1 Test Case Filtering

We allow the user to define a set of tests that check the correctness of the code with a snippet inserted. The correctness can be checked by defining the postcondition using Scala ensuring method or assert statements. Thus, after our search algorithm returns candidate snippets, we insert them one by one in the user code and run the tests. If at least one test fails we discard the candidate. At the end, the user obtains a list of snippets that pass all tests. In order to implement the test engine we use the Scala interactive interpreter and modify it to detect uncaught expressions, blocked and infinite executions. The blocked and infinite executions are prevented by setting a time limit to the interpreter. We should remark that using tests to filter snippets is mainly useful when modifying an existing code with a test suite. Moreover, in order to test the code we require that it is compilable after a candite snippet was filled in.

Our QTI implementation is publicly available at the link provided to the PC chair.

8.2 Weights for the Initial Type Environment

We next present in detail the weight assignment strategy for declared symbols and their types as implemented in QTI. 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 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 QTI is invoked. We take the proximity into account by assigning weights as shown in Table 3. 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 3. 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 3. Weights for Names Appearing in Declarations

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

We also assign weights to types. A type is represented by a term that may contain $\bot$ and $\rightarrow$ symbols, type constructors and primitive types. Their weights are given in Table 4. 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 wights of all symbols that occur in the type term.

Initial Type & Constructors	Weight
$\bot$	1
$\rightarrow$	1
Primitive type	2
Type constructor	2

Table 4. Weights for Simple Types and Constructors

Assignment of an entire type binding is then $w(e{:}\tau) = w(e) + w(\tau)$.

8.3 Evaluating the Impact of Search Techniques

Section 3 presented experiments that illustrate the overall effectiveness of QTI. To demonstrate the importance of the particular combination of techniques we employed, in QTI, we next present further experiments that measure the impact of each individual techniques. We specifically measure: backward search (function composition with a type of the form $\tau \rightarrow \bot_{fresh}$), using weights assigned according to declaration proximity, and using weights computed from a corpus of Scala code (see 8.2).

Table 5 presents the results and the effects of the individual techniques using the same set of benchmarks as in Table 2. As in the previous experiments, we measure for each example whether the expected expression is recovered. The table also records the number of expressions that our prover generates in a given benchmark (column # Generated Snippets). First, we ran QTI where all techniques (weights and backward search) were turned off (column W.+BS OFF). QTI could not generate 15 snippets (denoted with "FAILED"), and 1 had a rank greater than 5 (denoted with ">5"). If the rank is greater than 5 it is also possible that the expected snippet is not generate. In the remaining 10 benchmarks the sinppet had mostly the top rank.

Next, we turned on backward search, i.e., turned only weights off (column W. OFF). Here, QTI was able to recover 18 snippets, in 9 benchmarks the snippet was not in the top 5, and in one it did not generated the snippet. Also, the number of generated snippets increased, and in many benchmarks it is over few hundreds.

Benchmark (ground, then generic)		Solution Rank				# Generated Snippets			
		W.+BS. OFF	W. OFF	C.W. OFF	All	W.+BW. OFF	W. OFF	C.W. OFF	All
1	StreamTokenizerReaderr	FAILED	>5	>5	3	0	524	454	387
2	BufferedReaderInputStreamReader	FAILED	>5	>5	1	0	342	287	402
3	BufferedReaderReaderin	FAILED	2	2	1	0	862	764	372
4	ByteArrayInputStreambytebufintoffsetintlength	FAILED	>5	3	3	2	2	77	225
5	CharArrayReadercharbuf	FAILED	1	1	1	0	665	352	300
6	DataInputStreamFileInputStreamfileInputStream	FAILED	>5	>5	3	0	266	438	341
7	FileReaderFilefile	FAILED	>5	>5	2	0	430	420	283
8	PipedReaderPipedWritersrc	FAILED	3	3	3	1	580	311	338
9	ServerSocketintport	FAILED	2	2	2	1	906	138	254
10	TimerintvalueActionListeneract	FAILED	FAILED	1	1	0	0	1	1
11	TransferHandlerStringproperty	>5	>5	1	1	10	740	552	230
12	URLStringspecthrowsMalformedURLException	FAILED	>5	1	1	0	933	117	208
13	ArrayListiterator	FAILED	2	1	1	0	387	12	10
14	ArrayListtoArray	2	2	2	2	3	2	2	2
15	HashMapentrySet	FAILED	1	1	1	0	658	4	2
16	HashMapvalues	1	1	1	1	1	1	1	1
17	HashSetiterator	1	1	1	1	2	530	8	8
18	Hashtableelements	2	2	2	2	2	2	2	2
19	HashtableentrySet	FAILED	1	1	1	0	675	5	2
20	HashtablekeySet	1	1	1	1	1	1	1	1
21	Hashtablekeys	1	1	1	1	2	2	2	2
22	PriorityQueuepoll	2	1	1	1	4	557	220	179
23	TreeMapentrySet	FAILED	1	1	1	0	2	2	2
24	TreeMapvalues	1	1	1	1	1	1	1	1
25	Vectorelements	1	1	1	1	4	610	15	5
26	VectortoArray	2	2	2	2	5	2	10	2

Table 5. Impact of Individual Techniques in QTI Implementation.

In the next experiment we turned on weights based on declaration proximity (column C.W. OFF). The results show that QTI can recover 22 snippets, which is more than before. We further observe that rank of the benchmark 13 improves. The overall improvement is due to the fact that the deleted expressions often use local declarations and those defined close by. We also observe that this is common for many of the 100 benchmarks in our evaluation.

In the last experiment we turned on all techniques (column All). QTI was able to recover all 26 expected snippets, generating fewer solutions per benchmark. This shows that weights based on the corpus play an important role in guiding our algorithm towards snippets of higher quality.

In summary, our experiments suggest that each of the techniques we incorporated into QTI is important for obtaining high-quality results.

9. Related Work

Several tools including Prospector [11], XSnippet [16], Strathcona [7], PARSEWeb [21] and SNIFF [3] that generate or search for relevant code examples have been proposed. In contrast to all these tools we support polymorphic types and expressions with higher order functions. Additionally, we synthesize snippets using all visible methods in a context, where they 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 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. Unfortunately, the authors did not report exact running times for the tools. We next provide more detailed descriptions for some of the tools, and we compare their functionality to QTI.

Prospector [11] 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 QTI this is done automatically. Prospector uses a corpus for down-casting, where we use it to guide the search and rank the solutions. Moreover Prospector has no knowledge what methods are used more frequently. Unfortunately, we could not compare our implementation with Prospector, because it was not publicly available.

XSnippet [16] 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, context-sensitive and context-independent heuristics. In order to narrow the search the tool uses parental structure of the class where the query is initiated to compare it with the parents of the class 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 [7] 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 QTI those examples can not be fitted into the code without additional interventions.

PARSEWeb [21] uses the Google code search engine to get relevant code examples. The solutions are ranked by length and a frequency. QTI has an additional component by taking into account also the proximity of derived snippets and the point where QTI was invoked.

The main idea behind the SNIFF [3] tool is to use a 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. QTI is based on a logical formalism, so it can overcome the gap between programming languages and the natural language.

Agda [1] is a dependently typed programming language and proof assistant. Using Agdas 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 [8] 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 [6] search engine does not use types for searching functions: its search is based on function names. The main difference between Djinn [19] 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.

The tool demo on the InSynth tool [5] suggests the use of theorem prover for classical logic for synthesis. In contrast, we view our problem as more related to the type inhabitation problem, and intuitionistic logic. Furthermore, we provide a method to mine initial weights of declarations, which was very important for obtaining useful results. Finally, we provide an experimental evaluation on a substantial number of examples, as well as new theoretical foundations.

One of the most effective modern intuitionistic theorem provers, Imogen [12], can reason about very expressive non-classical logic (such as linear logics). QTI's prover is used for reasoning about the fragment of intuitionistic logic which is complete. QTI 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 [4] 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 [23]. The QTI tool does not enumerate them but instead it returns several best ranked proofs.

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

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

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

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

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

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

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

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

[12] S. McLaughlin and F. Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In *CADE*, pages 230–244, 2009.

[13] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 341–360, 2010.

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

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

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

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

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

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

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

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

[22] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th ACM POPL*, 1989.

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

A. Rule that combines Application and Abstraction

$$\textsc{App}\ \frac{\Gamma \vdash f : (\{S_1 \rightarrow T_1\} \cup S_2) \rightarrow T_2 \qquad \Gamma \cup \Gamma_{S_1} \vdash h : T_1}{\Gamma \vdash f(\lambda x_1 : ts_1 \ldots \lambda x_n : ts_n .\, h) : S_2 \rightarrow T_2}$$

$$\Gamma_{S_1} = \{x_i : ts_i | ts_i \in S_1 \wedge x_i \in Fresh\}$$

$$n = |\Gamma_{S_1}|$$