InSynth tool proof translation and (*Scala*) code generation *

and integration into the *Eclipse* IDE

Ivan Kuraj Tihomer Gvero Viktor Kuncak

École Polytechnique Fédérale de Lausanne
{first name}.{last name}@epfl.ch

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, a useful technique can be to synthesize and suggest valid expressions of a given type at a given program point to developer, whenever he asks for it.

InSynth is a tool for interactive synthesis of code snippets. The code synthesis approach behind *InSynth* is based on 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.

This paper focuses on the "back-end" module that has the role of code generation and code snippet output, its interaction with the type inhabitation problem solver module and integration of *InSynth* into the *Eclipse* IDE [1].

The overall experience with *InSynth* in *Eclipse* indicates that this approach to synthesizing and suggesting code fragments goes beyond currently available techniques and is a useful functionality of software development environments. The positive feedback from *Eclipse Scala* IDE plugin, a popular development environment for *Scala*, community, promises us that such a feature will bring huge advantage to *Eclipse Scala* IDE over other IDEs with respect to *Scala* development.

General Terms Languages, Verification

Keywords Type Inhabitation, Program Synthesis, Typing assist, *Scala*, *Eclipse*

1. Introduction

Libraries are one of the biggest assets for todays software developers, enabling developers to build on the shoulders of their predecessors. Useful libraries often evolve into complex application programming interfaces (*API*s) 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 [6]. Even if that is not the case, in many popular languages, there exists code constructs of non-negligible size that represent specific combination of API calls and are used very frequently. Such constructs tend to take developer's precious time to write them and even introduce chances to write them incorrectly.

Existing Integrated Development Environments (*IDE*s) 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. These efforts usually rely on simple syntactical API searches and do not consider the semantics of offered code snippets at the given program point. Such functionality suggests an interesting general direction of improving modern IDEs: introduce the ability to search and synthesize type-correct code fragments and offer them as suggestions to the developer (as presented in [6]).

In general, existing tools use forward-directed completion in which developer provides a starting value (e.g. a part of an identifier) and the tool bases its search on such starting input. The *InSynth* tool uses an observation that developers can productively use backward-directed completion in which, when identifying a computation step, the developer has the type of a desired code construct in mind. [2] Therefore *InSynth* does not require the developer to indicate starting fragments of code explicitly. Instead, *InSynth* uses an ambitious approach that considers all values in scope as the candidate leaf values from which expressions can be synthesized. Forward search typed values are not necessary and may be used to direct the synthesis.

This paper describes the contributions of the InSynth tool which at the focuses on the Scala language [9] and integrates into the Eclipse IDE [4] as a plugin.

1.1 Motivating example

In the next example we illustrate the functionality of the InSynth tool integrated in Eclipse as a plugin. Figure 1 depicts a portion of an visible editor in Eclipse IDE and the code suggestions returned by the InSynth tool and how they are shown to the developer.

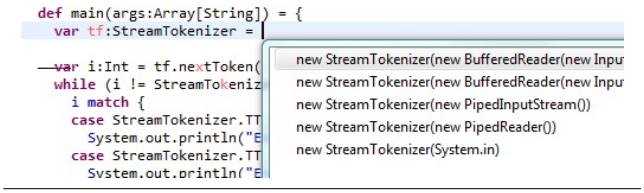


Figure 1. InSynth integration in Eclipse

The example shows a situation in which the developer has already written some Scala declarations and invokes the Eclipse typing assist functionality which in turn calls the InSynth code synthesis while having the typing pointer at a place for definition of the value which is declared to be of type *StreamTokenizer*. In such a scenario, InSynth tool is called with the given program context, needed type and visible declarations as the input. After some specific predefined time working in the background, the developer is provided with a some predefined number of code snippets which were found and ranked the best by InSynth in the given time. The

* The work in this paper is focused on an extension of previous version of the InSynth tool developed at EPFL LARA laboratory

[1] this work contributes to the InSynth tool idea presented in [6]

[2] Note that this idea can be extended to work without providing the starting type, with languages which provide type inference

developer can choose to insert one of such provided code snippets and dramatically reduce the time needed to define the declared *StreamTokenizer* value.

1.2 Paper outline

The introduction that sets up the context and a motivating example for the work in this paper for are already given in the Introduction section (Section 1). We then describe the design of the InSynth tool in terms of modules that InSynth consists of and the phases that are carried out when InSynth tool is invoked by the IDE in Section 2. The same section presents design decisions and rationales, algorithms used and implementation aspects involved in InSynth. In Section 3 a brief evaluation of the InSynth synthesis process and features is given. Finally the paper is concluded in Section 4 together with mentioning related work to InSynth and also some ideas for the future work.

2. Design

In this section we will describe the design of the InSynth tool in high-level of details, introduce the module that uses theorem proving techniques to search for valid code expressions, describe the concepts used in the code generation process and its implementation and justify adopted design decisions along the way.

2.1 High-level overview

InSynth can be thought of as being partitioned into two main modules: 1) proof resolution module and 2) code generation module. The overview of InSynth design is given in Figure 2.

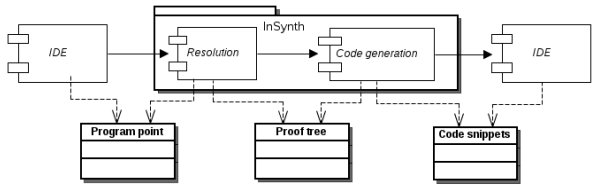


Figure 2. InSynth modules

The first, resolution module gets as input program context and a given type to synthesize (which are passed to it from the IDE) and its task is to search for all possible combinations of code constructs so that the resulting expressions are typable in the given program point to the desired type. This search uses somewhat simplified representation of program context so that it makes utilization of certain theorem proving techniques feasible and efficient. The output of the proof resolution phase represents proofs that involve simplified language constructs and witness that code expressions of the given type can be constructed. Such proofs are passed to the code generation module which uses additional information about the program to extrapolate code from them and synthesize code snippets, which when inserted at the desired location in the program evaluate to the given type and the overall program compiles, and feeds them back to the IDE so that they can finally be given as suggestions to the developer.

2.2 Resolution

The resolution phase has the task to search for every possible construction of an expression which has the given type. It considers all values in the given scope as the candidate leaf values for expression synthesis. Considering such a general scenario leads us directly to the type inhabitation problem [6, 14].

Definiton 1. *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 the type T, i.e. find an expression such that $\Gamma \vdash e : T$.*

More specifically, the goal of the resolution phase is to solve the type inhabitation problem - for a input type T, to use theorem proving techniques to search for proofs of construction of all valid expressions e of type T. These proofs are then forwarded to the code generations phase which uses them to produce synthesises syntactically correct code snippets.

InSynth gets the necessary inputs from the IDE and the resolution phase computes Γ by looking at all program declarations and visible API from the position of the cursor in the editor, looks up T by examining the declared type appearing left of the cursors in the editor and finally solves the type inhabitation problem.

The structure of the phase done by the proof resolution module is depicted in Figure 3.

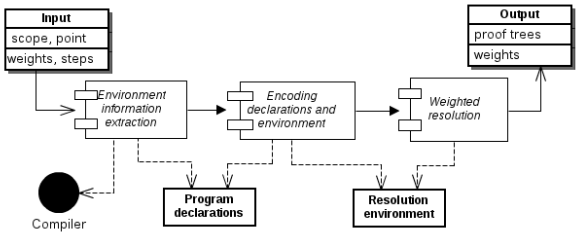


Figure 3. InSynth modules

2.2.1 Environment information extraction

The first step is the environment information extraction which has the task to extract all usable information about the program. In the case of Scala programs, the Scala compiler is consulted [3] to obtain all local and imported declarations visible at the given program point, which is determined by the typing cursor in the active editor. The environment extraction automatically assigns weights to all visible declarations according to the distance between the given declaration and the program pointer computed with respect to lines of code and package hierarchy. The rationale is that the more recent declarations are most likely to be used in the needed expression therefore they should be ranked according to the distance from the given program point.

The number of steps represents a constraint imposed on InSynth in terms of processing time, in order to achieve better responsiveness and ensure a timely termination and suggestions output even in cases of long computations.

2.2.2 Encoding the environment

The next step of the resolution phase is to encode the environment in the representation which is suitable for the type inhabitation solver to reason about and search for feasible expressions which can be derived from it.

Table 1 shows some examples of how are Scala declarations (and their types) transformed to appropriate representation terms. The simplified representation of Scala declarations and their types corresponds to the Definition 4.1 of *ground types* presented in [6]. The recursive definition of *ground types* denotes a set of types which include all constants (which correspond to Scala primitive types) and instantiations of constructors (which correspond to Scala generic types) with ground types. This means that in our resolution process we only consider the instantiated type constructions (e.g.

[3] in the Eclipse IDE integration implementation this is the Scala presentation compiler

a valid ground type could correspond to Scala types *Int, String, List[String], Map[Int, List[String]]* but not to *Map[X, Y]* where *X, Y* are polymorphic type variables).

Table 1. Correspondence between Scala declarations and proof representation terms

Scala declaration	Proof representation term
`val I:Int`	I: $\{\} \rightarrow Int$
`def fun(g: Int, f: Int=>Boolean): String`	fun: $\{Int, \{Int\} \rightarrow Boolean\} \rightarrow String$
`class A{ def m(): String }`	m: $\{A\} \rightarrow String$
`class A{ val s: String }`	s: $\{A\} \rightarrow String$
`def fun(i: Int, c:Char, j: Int): Char`	fun: $(Int, Char) \rightarrow Char$

The environment representation is based on the notion of a type function $f : \{X_1, X_2, \ldots\} \rightarrow Y$ which encodes the information that an expression of type Y can be constructed if a declaration f is used together with expressions of types that belong to the set $X = \{X_1, X_2, \ldots\}$. More specifically, if expressions $e_i : X_i$ such that $X_i \in X$ where $i = 1..|X|$ are available then we can construct an expression $e : Y$ by using those expressions e_i with the declaration f in combination.

Note that this step produces a representation that sacrifices some information about the language constructs (such as parameter arity, parameter ordering, curring, receiver objects, constructors...) in order to enable more convenient and efficient reasoning with the inhabitation problem solving process (and to make it actually feasible) but also includes corresponding program declarations in order to make the results that use such representation eligible for code reconstruction.

2.2.3 Weighted resolution

In order to search for code snippets of the given type, the resolution phase uses theorem proving techniques and reasoning about a similar problem that can be found in type theory, the so-called *type inhabitation problem*, as is defined in Definition 1. 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 [14].

After establishing the connection between program declarations and simplified representation suitable for resolution (or *ground type* terms) in the previous step which encodes the environment (Section 2.2.2), a calculus for reasoning about the type inhabitation problem is constructed by using only one resolution rule (in [6] it is denoted by *the ground applicative calculus* and it uses multiple rules instead).

APPABS

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

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

Figure 4. The calculus rule

The rule for the calculus construction is depicted in Figure 4. In combination with the *ground type* representation the APPABS rule subsumes both the application and abstraction semantics of the lambda calculus (and thus also function composition) in the resolution process. In order to define starting condition for the resolution process, the *query ground type* is added as $q : T \rightarrow \perp$ where T represents the required type of synthesized expressions and $\perp$ represents a fresh type (not usable with the rule for construction of new terms) which is used to denote the end of some expression construction (since q, can be instantiated only if an expression of type T itself has been constructed).

According to the defined calculus the last step of the resolution phase performs the weighted resolution (weights obtained from the information extraction are used to direct the search to more optimal results [6]) and produces proof trees which are then used as inputs to the code generation phase.

2.3 Code generation

In this section we will describe the code generation module which has the task of extrapolating and synthesis of code snippets from the proof trees obtained as a result of the resolution phase.

The overview of the code generation phase is given in Figure 5.

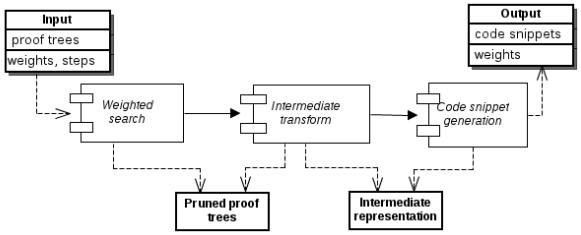


Figure 5. InSynth code generation phase

The code generation phase starts when the resolution phase finished (although its design allows starting the code generation phase as soon as possible and running it in parallel with the resolution phase while receiving partial proof tree updates[4]) and works on the proof tree representation. The input to the code generation phase besides the constructed proof trees includes the maximal amount of time for the computation (a constraint put on the InSynth tool for responsiveness) and the number of code snippets that should be generated and fed back to the developer.

The first step of the code generation phase is to extract a proof subtree (and transform it into very similar pruned tree representation) which is guaranteed to hold enough information for generation of sufficient number of code snippets with the lowest weight. The second step takes such pruned proof trees, consults embedded information about the program environment and constructs an intermediate representation tree which holds enough information about the structure of code and program declarations used in the synthesis. The third step takes the intermediate representation tree and applies transformations which generate a set of Scala code snippets and reports them back to the Eclipse IDE and the developer.

2.3.1 InSynth proof trees

InSynth proof trees are the output of the resolution phase and represent "proofs" of expressions construction, in the sense that expressions of the required type can indeed by constructed out of leafs as declarations from the given program environment. With respect to the encoding (described in 2.2.2), such proofs only bear information that valid expressions can be constructed with combining some specific ground types together with nested sub-expressions. Such combination is derived by using the rule presented in Figure 4 and uses simplified ground type reasoning - therefore it does not directly encode how should the synthesized code snippets actually look like when given as suggestions to the developer. InSynth proof trees include the information about correspondence between used ground types and program declarations in order to allow syntactically synthesis of valid code snippets. In addition to including information about program declarations, since the resolution is driven

[4] the functionality is considered to be implemented in future work

by weights of such declarations, proof trees also include information about the weights of encoded ground types.

An example that depicts the outline of the structure of InSynth proof trees which are inputs to the code generation phase is depicted in Figure 6.

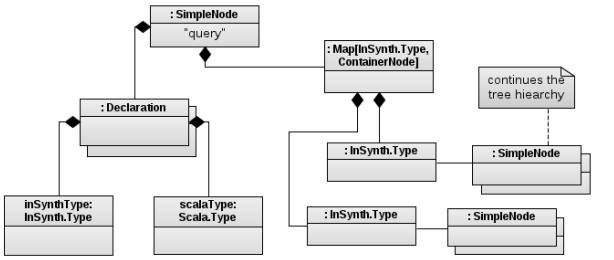


Figure 6. Object diagram with an InSynth proof tree example

The proof tree consists of nodes (implemented as *SimpleNode* classes) that carry the information about which ground types (given by *Declaration* class) can be combined with expressions of types defined by the parameters map. The parameters map represents information on how to construct sub-expressions of a given node and maps a type (*InSynth.Type*) to a set of subtrees (contained in the *ContainerNode* object). Each *Declaration* consists of the simplified ground type *InSynth.Type* which can be used in the combination its corresponding program declaration *Scala.Type* which stores the language-specific information that are needed for the actual code generation.

Note that according to the resolution phase (which uses the APPABS rule) and simplified representation of *ground types*, each node in the proof tree provides information how to synthesize expression of the "rightmost" primitive type, T_1 in the rule, while having variables of types in S_1 added to the typing context (e.g. for synthesis of Scala type *Int =>Char =>String* the proof tree would provide information on synthesis of expressions of type *String* while having variables of types *Int, Char* added to the typing context).

2.3.2 Weighted search

Weighted search is the first step in the code generation process and its main goal is to extract only the needed number N of the most optimal combinations in terms of associated weights. The result of this step is a proof tree that contains only a subset of nodes from the original, input proof tree such that the belonging nodes are sufficient in constructing (at least) N code snippets with the lowest weight that need to be suggested to the developer[5]. The rationale behind this step is that the resolution step may, due to combinatorial explosion, output complex proof trees with a large number of nodes and the code generation phase can benefit from its pruning to achieve better responsiveness of the typing assist. After this step, the unnecessary nodes are removed from the proof tree and the output entails only the necessary information for construction of most optimal solutions based on the assigned weights of program declarations.

The algorithm that accomplishes the weighted search is based on the uniform-cost search which is a search algorithm used for traversing weighted tree structures [11]. The search begins at the root node and continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

[5] more specifically the output of this step uses a slight variation of the proof tree representation and due to the nature of the simplified representation, is guaranteed to hold information to generate N or more code snippets

The main difference is that in our case there is no single global goal state. When a subtree is fully explored its cumulative weight serves as a weighted goal for constructing an expression corresponding to that subtree. When the root of the tree is fully explored we are sure that at least one expression can be combined in order to synthesize the goal (query) type. Number of expressions a subtree can combine is equal to the product of number of possible declarations and each of its explored parameter subtrees.

The weighted search algorithm is depicted in Algorithm 1.

Algorithm 1 Weighted search

```
Require: root node r
enqueue r
while queue is not empty do
    dequeue node n
    if n is not pruned then
        update weights of nodes up the tree and prune nodes
        according to their updated weights if needed
        if n is a leaf node then
            mark the subtree as explored and propagate to r
            if number of combinations at r is ≥ N then
                enable pruning of nodes
            end if
        end if
        for all children c of n do
            if c is not visited then
                enqueue c
            end if
        end for
    end if
end while
return r
```

The algorithm gradually explored nodes according to their weights (set of visited nodes is maintained since in general there can be cycles in the proof tree), prunes the subtrees that ought to construct expressions of larger weight and finishes when the resulting tree contains nodes for construction of at least N expressions. Note that the algorithm does not stop until it exhausts the priority queue of unexplored nodes and only examines nodes that can explore optimal subtrees (with total weight less than previously explored subtrees).

The design of the algorithm allows implementation of incremental updates of proof trees from the resolution phase (which would add unexplored nodes to the priority queue) and it allows its implementation to support some more sophisticated policies for nodes weight calculation (instead of summing the weights of subtree node it can search for package declarations path matches, specific subtree pattern matches, etc.).

2.3.3 Intermediate transformation

In the second step of the code generation phase, an intermediate transform is performed on the input (pruned) proof tree.

The goal of this step is to produce intermediate representation trees which combine program declarations and combinations encoded in proof trees and contain enough information about the structure of the code to be synthesized. The resulting trees in the intermediate representation are very similar to terms from the λ-calculus and thus precisely encode information about abstractions and applications to be used in the synthesized code.[6]

[6] In spite of containing more precise information about the code to be synthesized, the intermediate representation offers an abstract way of encoding the code structure so that potentially other programming languages can be supported for code snippet generation

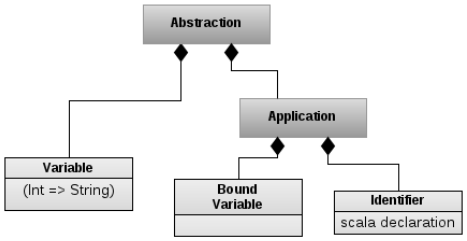


Figure 7. Intermediate representation tree example

An example of a tree in intermediate representation is given in Figure 7. The example encodes a single expression in which an identifier from the program context is applied to the function term declared in the top abstraction. The intermediate representation trees contain abstraction nodes which correspond to abstraction terms in the λ-calculus but which can bind multiple terms (in order to correspond syntactically more powerful abstraction counterparts in Scala), application nodes which represent an application in the λ-calculus (again, the distinction is that they can include multiple parameters), identifier nodes that represent program declarations and bound variables that represent usage of terms declared by some abstraction at higher levels in the tree. Important property of the intermediate representation is that each node in the place of its sub-terms can contain multiple nodes (a set of nodes) in order to allow encoding that multiple expressions of the same type can be used whenever an expression of that type is required.

We will now describe the algorithm for transforming InSynth proof trees to intermediate representation trees.

Algorithm 2 Transformation procedure

Require: InSynth proof tree rooted at r
1: $\{r$ is the query node which has type $(T \rightarrow \bot)\}$
2: **return** Transform($\emptyset$, r, $\bot$)

The entry point to the transformation is depicted in Algorithm 2. Its input is the (pruned) proof tree, more specifically its root r which contains the initial *query type* $T \rightarrow \bot$, where T is the type of expressions we want to synthesize. The algorithm initializes an empty typing context and calls the recursive *Transform* procedure given in Algorithm 3 on the root node with an empty context and with $\bot$ as the goal type. Note that if an expression of type $\bot$ is derived then its immediate sub-expressions will be of the required type T.

Algorithm 3 Transform

Require: context Γ, current node n, goal type t
1: {context Γ is the current typing context}
2: **if** t is of the form $(X \Rightarrow Y)$ **then**
3: **for all** type X_i in X according to real type **do**
4: let be fresh variable x_i of type X_i
5: **end for**
6: let a be an abstraction that bounds all variables in X $\{a = (\lambda x_1 : X_1.(\lambda x_2 : X_2. \ldots (\lambda x_n : X_n.$ "_"$)))\}$
7: **for all** t' in Transform declarations($\Gamma \cup (\bigcup_i x_i : X_i)$, n, Y) **do**
8: **return** a ["_" $\rightarrow t'$]
9: {plug t' into the abstraction a in place of "_"}
10: **end for**
11: **else**
12: **return** Transform declarations(Γ, n, t)
13: **end if**

The recursive Transform, depicted in Algorithm 3 expects as inputs the current node n in the proof tree, current typing context Γ and a goal type t, to which expressions constructed from subtree n should typecheck to.

It checks the form of t (at the first call, t is equal to the *query type*):

- if t is not a function type - according to the rule APPABS by which the resolution step is derived, t represents T_1 and $S1 = \{\}$ - the transformation proceeds immediately recursively to get sub-expressions of type T_1
- if t is a function type $X \rightarrow Y$ - according to the rule APPABS, t represents $S_1 \rightarrow T_1$ - an abstraction terms(s) are formed in order to introduce variables of types found in X and the transformation proceeds recursively to get sub-expressions of type T_1 under the updated context

The transformation of type T_1 in both cases is achieved with a helper procedure given in Algorithm 4, which scans the available declarations in the current node and current context in order to transform sub-expressions.

Algorithm 4 Transform declarations

Require: context Γ, current node n, goal type t
1: search for declarations that can return Y type in declarations contained in the node n and in Γ
2: denote the result set of declarations as D
3: **for all** declarations d from D **do**
4: **if** d has no parameters **then**
5: **return** identifier or bound variable node for d
6: **else**
7: **for all** parameters p_i of type t_i in d **do**
8: **for all** child nodes n' that are contained in container node $parameters(t_i)$ **do**
9: let S_i be the result of Transform(Γ, n', Y) {set of nodes that represent sub-expressions of type t_p}
10: **end for**
11: **end for**
12: **return** application node (d S_1 S_2 $\ldots S_n$) {if d has n parameters}
13: **end if**
14: **end for**

The *Transform declarations* procedure scans declarations at the given node and the context to find suitable declarations that can transform to a given goal type. Such declarations may be returned directly as identifier or bound variable nodes (do not require application to them) or as applications of recursively transformed parameters of appropriate type. Note that intermediate representation allows set of nodes to be included for each application and abstraction nodes thus enables representing multiple expressions with one tree.

2.3.4 Code snippet generation

The code snippet generation represents the final step in the code generation phase and it takes intermediate representation as input and produces code snippets in a target language, namely Scala, and ranks them according to their weight.

This step is based on a tree traversal transformation algorithm which traverses the intermediate representation tree and produces a set of code snippets. Since the intermediate representation tree encodes the program structure and also allows multiple sub-trees in its abstraction and application nodes, the code snippet generation step has to consider and collect every possible expression that is included in the intermediate representation tree (and returns only the specified number of them, ranking them by weight).

Although the intermediate representation precisely encodes the structure of code snippets to be generated in terms of λ-calculus terms, the code snippet generation step has to consult the information provided by the program declarations in order to be able to generate syntactically correct code and also to be able to simplify the resulting code as much as possible. These language-specific transformations include usage of correct syntactical constructs (e.g. if the application represents a construction of an object, keyword *new* has to be outputed), usage of syntactic sugar instances (e.g. omitting dot, parentheses and *apply* in certain method calls) and general simplification steps (e.g. omitting of explicitly given types to expressions).

Resulting code snippets are encoded as a set of Scala pretty print documents (*Scala.text.Document* objects) which are then transformed by custom-indentation defined properties to strings and reported back to the IDE (the output of such objects can be then processed with Scala format and refactoring libraries in order to have visually better suggestions reported to the developer, as described in architecture section in [5]).

3. Evaluation

In this section we will describe important design decisions, properties and functionalities of the code generation phase of InSynth. Afterwards, we will demonstrate the code generation process with an example of code synthesis for a particular input.

3.1 Properties of the code generation

The most important property of the code synthesis is to generate valid code snippets. Generated code snippets are valid if under an assumption that proof trees obtained from the resolution phase encode valid expressions (and include correct program declarations), the code generation phase synthesises code snippets that, when inserted at the given program point, typecheck to the given type and the overall program compiles successfully.

3.1.1 Validity of the code generation phase

Since the code generation phase consists of three steps, we will enumerate their validity properties individually:

- weighted search - This step performs pruning of initial proof trees in a way that guarantees at least (needed) N expressions to be combined. The pruned proof tree represents a subset of nodes from the original proof tree in such a way that the structure of that subset is preserved, thus the set of expressions encoded in the pruned proof tree must be a subset of set of expressions encoded in the original proof tree.
- intermediate transform - The correctness of this step directly depends on the correctness of the APPABS resolution rule and the correspondence between encoded ground types and program declarations. The intermediate transform directly follows the APPABS rule by transforming, for each provided declaration its parameters according to the enriched context in the case of parameter type being a function or the original context otherwise. Since the transformation is done according to actual program declarations (types), the intermediate transform trees encodes the correct code structure.
- code snippet generation - This step relies on properties of the target programming language (Scala) and the transformation is defined by programming language syntactic rules and syntactic sugar instances.

3.1.2 Completeness

The second important property of the code synthesis is to be complete, in the sense that the code generation phase should generate all expressions that can be found by the resolution phase (i.e. expressed in the target language) and encoded as "proofs" in the input InSynth proof trees.

An interesting treatment of the InSynth code synthesis process is the generation of combinators from the SKI combinatory logic [7]. Combinatory logic may be viewed as a subset of lambda calculus, the theories are largely the same, becoming equivalent in the presence of the rule of extensionality. Extensional equality captures the mathematical notion of the equality of functions: two functions are equal if they always produce the same results for the same arguments [13]. The SKI combinatory logic contains the same expressive power as lambda calculus and the logic is variable free, i.e. the abstractions are not part of the logic. Combinators from the SKI combinatory logic can be composed to produce combinators that are extensionally equal to any lambda term, and therefore, by Church's thesis, to any computable function whatsoever. The process of obtaining an expression in combinatory logic from any given λ-calculus term can be done with the *abstraction elimination* procedure [13]. In the next section we will see that InSynth is capable of synthesis of combinators from the SKI combinatory logic, when the appropriate input type is given.

In our case the extensional equality of two expressions translates to equivalence of behavior of the two expressions under certain evaluation (reduction [10]) rules. Although, due to a specific treatment of declarations (which correspond to application terms as defined in λ-calculus) InSynth synthesis process is not capable of producing every possible code snippet that could be otherwise typed by the developer, it can produce a behaviorally (extensionally) equivalent expression, under the reduction rules of the Scala language. The specific treatment of declarations limit the expressiveness of the application term to an identifier or a declared term - this means that the expression $(\lambda x : Int.x)\ 5$ (where 5 represents an appropriate λ-calculus encoding of constant 5) which corresponds to Scala code snippet *((x:Int) => x)(5)* (which typechecks successfully to *Int*) cannot be produced by InSynth, but its behavioral equivalent $(\lambda x : Int.x)[x \rightarrow 5]$ or in Scala, just the literal *5*, can.

We can conclude that InSynth is indeed capable of producing behaviorally equivalent code to every possible Scala expression, thus every reasonable and useful snippet of code. [8]

Although the code generation phase should be as expressive as the resolution phase resulting proof trees can be (since the code generation phase consults only proof trees for the code extrapolation and synthesis) the intermediate representation adopted provides the same expression power as λ-calculus (which is Turing-complete [9, 10, 13]) and thus is even more expressible than the proof tree encodings.

3.2 Demonstration example

We will give an example of InSynth tool code generation process[9], show its intermediate outputs and final results, for the case of the S combinator synthesis, somewhat a combinator from the SKI logic [13]. The S combinator is defined as $(S\ x\ y\ z) = (x\ z\ (y\ z))$ and by its nature it does not require any predefined program declarations, i.e. it can be synthesized with the empty program environment.

[7] many examples in the literature refer to the combinatory logic as SKI, in spite of the fact that combinators S and K provide completeness of the theory, while I can be expressed as I = S K K

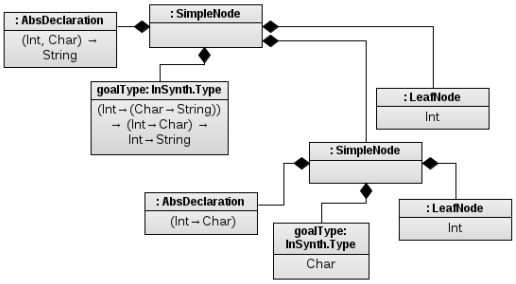
[8] One interesting remark to notice that under some rather strange (and unpractical) reduction rules (like e.g. full beta-reduction [10]) $(\lambda x : T.y)z$ may not be behaviorally equivalent to $y[x \rightarrow z]$ in all contexts

[9] such an example seems the most appropriate since the full integration of the code generation module was not yet possible at the time of writing this paper

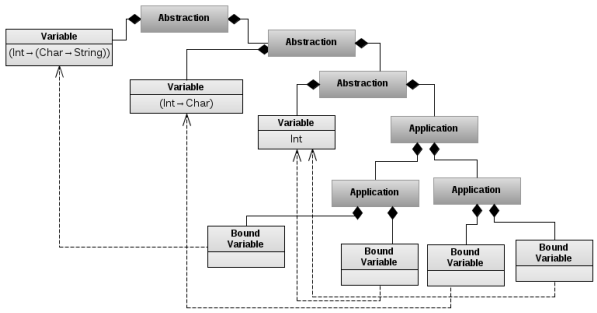
The Scala declaration with type of the S combinator (instantiated appropriately with Scala primitive types *Int, Char, String* to be able to be represented as a *ground type*) can be given as

```
val S:(Int=>(Char=>String))=>(Int=>Char)=>Int=>String =
```

The resulting proof tree as the output of the resolution phase is given in the following figure and it represents nested applications of terms introduced in the context (by the APPABS rule).



In this particular example the weighted search phase does not affect the tree since there is only one valid expression to be synthesized. The intermediate transformation phase produces a tree that in this case completely corresponds to the λ-calculus encoding of the S combinator.



The last step produces of the generation phase produces the scala code which successfully typechecks when inserted as the definition of the declared value *S*.

```
val S:(Int=>(Char=>String))=>(Int=>Char)=>Int=>String =
{
 (var_1: ((Int) => ((Char) => String)))
 =>
 {
  {
   (var_2: (Int) => Char)
   =>
   { { (var_3: Int) => { var_1 (var_3) (var_2 (var_3)) } } }
  }
 }
}
```

4. Conclusions

We have described the InSynth tool, mainly its code generation process and its integration with the Eclipse IDE. Code generation module is incorporated into the InSynth tool and extends the resolution module which searches for expressions of a given type in a type environment while minimizing a metric on the type binding in order to extrapolate and synthesize the most optimal meaningful code snippets and reports them to the developer. We implemented the module in three passes and while its implementation offers presented functionality, its design allows convenient extension to support more advanced functionalities. Among the key features of the code generation module is the process which takes proof trees from the resolution phase as the input, extracts only necessary information according to specified weight functions and produces code snippets which are valid and appropriately formatted.

4.1 Related Work

There has been work on tools that provide similar functionalities such as InSynth. There are tools for Haskell that do an API search. The Hoogle [3] and Hayoo [2] search engines perform search but rely on user inputs and search only for function names inside APIs. Djinn [1] employs a similar idea as InSynth but does not consider weights for its searches. Some tools like Prospector [8] and XSnippet [12] base their search only on a corpus of collected code rather then searching for possible solutions to be synthesized.

The tool demo on the InSynth tool [7] suggested the use of theorem prover for classical logic for synthesis. In contrast, the current view of our problem is more related to the type inhabitation problem, and intuitionistic logic. Furthermore, in our approach a method to derive initial weights of declarations is provided, which can prove very important for obtaining useful results.

4.2 Future Work

The future work that will happen in a near future would definitely include work on full integration with the Eclipse IDE. Besides generating expression only when the developer invokes the typing assist after declaring a value with a specified type, InSynth could be used to synthesize code expressions in places such as method call parameters and places where the Scala type inferencer can determine the needed expression type. Some more advanced features that could be built into the Eclipse IDE would be to include incremental resolution in which user inputs can aid the resolution process and make InSynth much more interactive.

Since InSynth currently supports only type inhabitation problem solving for fully instantiated *ground terms*, one idea would be to extend InSynth to cope with polymorphic types, bounded quantification and reasoning about type variance. Very attractive idea is to introduce a corpus of source code (projects written in Scala) and to mine the corpus in order to obtain weighting function for terms and even specific patterns of terms and thus synthesize more useful code snippets.

An implementation of certain semantic checks on the synthesized code would greatly contribute to the practical value of InSynth since by the current process, semantically useless code snippets can be generated (e.g. new *List[Int]().head* can be generated as a code snippet of type *Int*).

References

[1] The djinn theorem prover. URL `http://www.augustsson.net/Darcs/Djinn/`.

[2] Hayoo! api search. URL `http://holumbus.fh-wedel.de/hayoo/`.

[3] Hoogle api search. URL `http://www.haskell.org/hoogle/`.

[4] The eclipse foundation. the eclipse project. URL `http://www.eclipse.org/`.

[5] Scala ide team. scala ide developer documentation. URL `http://scala-ide.org/docs/dev/`.

[6] T. Gvero, V. Kuncak, and R. Piskac. Code Completion using Quantitative Type Inhabitation. Technical report, 2011.

[7] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *In CAV, pages 418423*, 2011.

[8] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005.

ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065018. URL http://doi.acm.org/10.1145/1065010.1065018.

[9] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition edition, Nov. 2008. ISBN 0981531601.

[10] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition, Dec. 2009. ISBN 0136042597.

[12] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. *SIGPLAN Not.*, 41(10):413–430, Oct. 2006. ISSN 0362-1340. doi: 10.1145/1167515.1167508. URL http://doi.acm.org/10.1145/1167515.1167508.

[13] J. Tromp. Binary Lambda Calculus and Combinatory Logic. Technical report, 2006.

[14] P. Urzyczyn. Inhabitation in typed lambda-calculi (a syntactic approach). In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, TLCA '97, pages 373–389, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-62688-3. URL http://dl.acm.org/citation.cfm?id=645893.671612.