Comfusy: A Tool for Complete Functional Synthesis
Tool Presentation

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter*

`firstname.lastname@epfl.ch`
Swiss Federal Institute of Technology (EPFL), Switzerland

Abstract. Synthesis of program fragments from specifications can make programs easier to write and easier to reason about. We present Comfusy, a tool that extends the compiler for the general-purpose programming language Scala with (non-reactive) functional synthesis over unbounded domains. Comfusy accepts expressions with input and output variables specifying relations on integers and sets. Comfusy symbolically computes the precise domain for the given relation and generates the function from inputs to outputs. The outputs are guaranteed to satisfy the relation whenever the inputs belong to the relation domain. The core of our synthesis algorithm is an extension of quantifier elimination that generates programs to compute witnesses for eliminated variables. We present examples that demonstrate software synthesis using Comfusy and illustrate how synthesis simplifies software development.

1 Introduction

Synthesis is among the most ambitious techniques for building correct computer systems [4]. Recently, we have seen advances of synthesis for finite-state reactive systems [6, 1]. In this paper, we describe a step in another direction: synthesis for *infinite-state* non-reactive software systems [2]. Our goal is to gradually introduce synthesis into software development by supporting new programming language constructs that leverage synthesis in delimited portions of the program. Specifically, we introduce a programming language construct, choose. The choose construct accepts a parameterized predicate P. It synthesizes a function that maps the parameters to output values satisfying P. We restrict the language of predicates to a decidable logic, and provide a complete synthesis procedure: whenever a value satisfying the predicate exists, the synthesized function will compute one such value.

We continue by illustrating our system through examples. We then define our synthesis problem more precisely and describe our implementation.[1]

* The author list has been sorted according to the alphabetical order; this should not be used to determine the extent of authors' contributions. Ruzica Piskac was supported in part by the SNF Grant SCOPES IZ73Z0_127979. Philippe Suter was supported by the SNF Grant 200021_120433.

[1] For further details, see [2] and http://lara.epfl.ch/dokuwiki/comfusy.

2 Examples

Linear arithmetic. As a first example, consider the problem of decomposing a number of seconds into hours, minutes and the leftover seconds. We can specify this problem as follows:

```
val (hours, minutes, seconds) = choose((h: Int, m: Int, s: Int) ⇒ (
    h ∗ 3600 + m ∗ 60 + s == totsec && 0 ≤ m && m < 60 && 0 ≤ s && s < 60))
```

On this example, Comfusy generates the following code:[2]

```
val (hours, minutes, seconds) = {
  val loc1 = totsec div 3600
  val num2 = totsec + ((−3600) ∗ loc1)
  val loc2 = min(num2 div 60, 59)
  val loc3 = totsec + ((−3600) ∗ loc1) + (−60 ∗ loc2)
  (loc1, loc2, loc3)
}
```

Arithmetic pattern matching. We also found synthesis for linear arithmetic to be useful for extending pattern-matching in a way that is similar to, but goes beyond Haskell $(n+k)$-patterns. The following code implements the fast exponentiation algorithm:

```
def pow(base : Int, p : Int) = {
  def fp(m : Int, b : Int, i : Int) = i match {
    case 0 ⇒ m
    case 2∗j ⇒ fp(m, b∗b, j)
    case 2∗j+1 ⇒ fp(m∗b, b∗b, j)
  }
  fp(1,base,p)
}
```

The third pattern, for instance, will match the integer i if there exists an integer j such that i == 2 * j + 1. The pattern also works as a binder, and the value computed for j is thus available on the right hand side. Comfusy checks that the match expression is exhaustive and that no pattern is subsumed by the previous ones, and emits a warning if it can find a value matched by no pattern or if a pattern in unreachable.

Parametrized linear arithmetic. The previous two examples are in standard linear arithmetic. Comfusy can also handle constraints expressed in *parametrized* linear arithmetic, that is, constraints that are not linear at compile-time but become linear at run-time, when some of the values are known. For example, the following code computes, if it exists, the integer ratio between two numbers a and b:

```
val ratio = choose((r: Int) ⇒ a == r ∗ b || b == r ∗ a)
```

[2] The div operator computes the floored integer division. For example -1 div $2 = -1$.

Although the term r $*$ b, for instance, is not linear at compile-time, the value of b is known at run-time at the point where the value of r needs to be computed. The synthesized code thus needs to handle all possible values of the parameters a and b.

Set constraints. Finally, Comfusy can be used to synthesize code handling sets. Consider the following example:

```
val (a1,a2) = choose((a1:Set[O],a2:Set[O]) ⇒
      a1 ++ a2 == s && a1 ** a2 == Set.empty
  && a1.size − a2.size ≤ 1 && a2.size − a1.size ≤ 1)
```

Here, ++ and ** denote set union and intersection respectively. The generated code constructs two sets a1 and a2 such that they form a partition of the existing set s, with the additional constraint that the sizes of a and b should not differ by more that 1. Note that requiring that their sizes be identical would result in an unsatisfiable set of constraints whenever the size of s is odd.

3 Definition and Algorithm for Synthesis in Comfusy

Definitions. Let $\mathsf{FV}(q)$ denotes the set of free variables in a formula or term q. If $\boldsymbol{x} = (x_1, \ldots, x_n)$ then $\boldsymbol{x}_s$ denotes the set of variables $\{x_1, \ldots, x_n\}$. If q is a term or formula, $\boldsymbol{x} = (x_1, \ldots, x_n)$ a vector of variables and $\boldsymbol{t} = (t_1, \ldots, t_n)$ a vector of terms, then $q[\boldsymbol{x} := \boldsymbol{t}]$ denotes the term resulting from substituting in q free variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, respectively.

Definition 1 (Synthesis Procedure). *A synthesis procedure takes as input a formula F and a vector of variables $\boldsymbol{x}$ and outputs a pair of*

1. *a precondition formula* pre *with* $\mathsf{FV}(\mathsf{pre}) \subseteq \mathsf{FV}(F) \setminus \boldsymbol{x}_s$
2. *a tuple of terms* $\boldsymbol{\Psi}$ *with* $\mathsf{FV}(\boldsymbol{\Psi}) \subseteq \mathsf{FV}(F) \setminus \boldsymbol{x}_s$

such that the following two implications are valid:

$$\exists \boldsymbol{x}.F \rightarrow \mathsf{pre}$$
$$\mathsf{pre} \rightarrow F[\boldsymbol{x} := \boldsymbol{\Psi}]$$

Algorithms. Our core specification language is quantifier-free Boolean Algebra with Presburger Arithmetic (BAPA) [3].[3] Our procedure for integer linear arithmetic synthesis is related to the Omega-test algorithm [7]. One of the key differences is that our procedure computes witness terms for eliminated variables. Additionally, in the parametrized arithmetic case, some choices in the algorithm need to be delayed until the run-time values are known; the synthesized code must account for these choices by generating different cases for different signs of coefficients and by, e.g., invoking a GCD algorithm in the generated code. The algorithm for constraints on sets is based on a witness-generating version of [3].

[3] We currently do not support quantifiers in the specification predicates. Quantifiers do not increase the set of definable relations, because BAPA has quantifier elimination [3]. We could support quantifiers by running the quantifier elimination algorithm first, then invoking our synthesis procedure.

4 Implementation

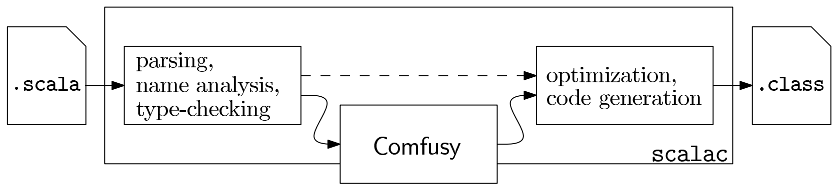

Fig. 1. Interaction of Comfusy with `scalac`, the Scala compiler. Comfusy takes as an input the abstract syntax tree of a Scala program and rewrites calls to choose to syntax trees representing the synthesized function.

We have implemented Comfusy as a plugin for the Scala compiler (scalac), adding a phase to the standard compilation process (see Figure 1). During this phase, our plugin extracts calls to the choose function and arithmetic patterns and replaces them by code that computes the appropriate values. The input and output of Comfusy are thus abstract syntax trees in the internal format of scalac. The compiler then proceeds as usual, so all further optimizations are applied to the synthesized code as well. Comfusy supports synthesis for predicates expressed in integer linear arithmetic, parametrized linear arithmetic, and set algebra with size constraints, as well as linear arithmetic patterns. Comfusy can also check whether the synthesis predicates are always satisfiable (for all possible run-time values of the program variables) or whether they describe unique solutions, and emit compile-time warnings with counter-examples when necessary. We use an off-the-shelf decision procedure for these checks [5]. In our experience, the execution time of the synthesized code is similar to equivalent hand-written code. We also found the compile-time overhead to be negligible.

References

1. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Computer Aided Verification (CAV). LNCS, vol. 4590 (2007)
2. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: ACM Conf. Programming Language Design and Implementation (PLDI) (2010)
3. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. Journal of Automated Reasoning 36(3), 213–239 (2006)
4. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Communications of the ACM 14(3), 151–165 (1971)
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963 (2008)
6. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: ACM Symp. Principles of Programming Languages (POPL) (1989)
7. Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM 35(8), 102–114 (1992)