From Decision Procedures to Synthesis Procedures

(Invited Paper)

Ruzica Piskac
Yale University
Email: ruzica.piskac@yale.edu

***Abstract*—Software synthesis is a technique for automatically generating code from a given specification. The goal of software synthesis is to make software development easier while increasing both the productivity of the programmer and the correctness of the produced code. In this paper we present an approach to synthesis that relies on the use of automated reasoning and decision procedures. First we describe how to generalize decision procedures into predictable and complete synthesis procedures. Here completeness means that the procedure is guaranteed to find code that satisfies the given specification. We illustrate the process of turning a decision procedure into a synthesis procedure using linear integer arithmetic as an example.**

However, writing a complete specification can be a tedious task, sometimes even harder than writing the code itself. To overcome this problem, ideally the user could provide a few input-output examples, and then the code should be automatically derived. We outline how to broaden usability and applications of current software synthesis techniques. We conclude with an outlook on possible future research directions and applications of synthesis procedures.

ACKNOWLEDGMENTS

This presentation is based on [16], [15], [25], [10], and the author thanks to all her co-authors and acknowledges their contributions. This paper contains some excerpts from the original papers.

I. INTRODUCTION

Synthesis of software from specifications, discussed already in [20], [19], [7], promises to make programmers more productive. Despite substantial recent progress [9], [27], [26], [32], [29], synthesis is limited to small pieces of code. We expect that this will continue to be the case for some time in the future, for two reasons: 1) synthesis is algorithmically a difficult problem, and 2) synthesis requires detailed specifications, which for large programs become difficult to write.

We therefore expect that practical applications of synthesis lie either in its integration into the compilers of general-purpose programming languages, or synthesis tools will need to be able to derive programs from incomplete specification, such as type constraints or input-output examples.

We first tackle the problem of integrating compilers and synthesis. To make this integration feasible, our goal is to identify well-defined classes of expressions and synthesis algorithms guaranteed to succeed for these classes of expressions, just like a compilation attempt succeeds for any well-formed program. Our starting point for such synthesis algorithms are *decision procedures*.

A decision procedure for satisfiability of a class of formulas accepts a formula in its class and checks whether the formula has a solution. On top of this basic functionality, many decision procedure implementations provide the additional feature of generating a satisfying assignment (a model) whenever the given formula is satisfiable. Such a model-generation functionality has many uses, including better error reporting in verification [21] and test-case generation [1]. An important insight is that model generation facility of decision procedures could also be used as an advanced computation mechanism. Given a set of values for some of the variables, a constraint solver can at run-time find the values of the remaining variables such that a given constraint holds. Two recent examples of integrating such a mechanism into a programming language are the quotations of the $F\#$ language [30] and a Scala library [14], both interfacing to the Z3 satisfiability modulo theories (SMT) solver [5]. Such mechanisms promise to bring the algorithmic improvements of SMT solvers to declarative paradigms such as Constraint Logic Programming [13]. However, they involve a possibly unpredictable search at run-time, and require the deployment of the entire decision procedure as a component of the run-time system.

This paper describes an approach to synthesis which provides the benefits of the declarative approach in a more controlled way: we aim to run a decision procedure at *compile time* and use it to generate code. The generated code then computes the desired values of variables at run-time. Such code is thus specific to the desired constraint, and can be more efficient. It does not require the decision procedure to be present at run-time, and gives the developer static feedback by checking the conditions under which the generated solution will exist and be unique. We use the term *synthesis* for our approach because it starts from an implicit specification, and involves compile-time precomputation. Because it computes a function that satisfies a given input/output relation, we call our synthesis *functional*, in contrast to reactive synthesis approaches [23]. Finally, we call our approach *complete* because it is guaranteed to work for all specification expressions from a well-specified class.

We illustrate our approach by describing synthesis algorithms for the domains of linear arithmetic. There are another extensions such as collections of objects [17], [16] and bit-vectors [28]. These synthesis algorithms are implemented and deployed them as a compiler extension of the Scala programming language [22]. We have found that using such

978-1-5090-0461-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SYNASC.2015.78

constraints we were able to express a number of program fragments in a more natural way, stating the invariants that the program should satisfy as opposed to the computation details of establishing these invariants.

However, writing a complete specification can be a tedious task, sometimes even harder than writing the code itself. One of the approaches to overcome this problem is so called *Programming by Example*. Ideally the user could provide a few input-output examples, and then the code should be automatically derived. The Programming by Example (PBE) paradigm [4], [18], [8] is a promising research direction that can enable rich, easy data manipulation even for non-programmers [9]. The success and impact of this line of work is witnessed by the fact that some of this technology ships as part of the popular Flash Fill feature in Microsoft Excel 2013.

We finish the paper with giving an overview of some on-going work and describing possible future research directions and applications of synthesis procedures.

II. Complete Functional Synthesis

A. A Motivating Example

We first illustrate the use of a synthesis procedure for integer linear arithmetic on the following example. Consider the scenario: you are given a number of seconds (stored in the variable `totsec`) and you need to to break it down into hours, minutes, and leftover seconds. We specify this problem as:

```
val (hrs, mns, scs) = choose((h: Int, m: Int, s: Int) ⇒
 h ∗ 3600 + m ∗ 60 + s == totsec &&
 0 ≤ m && m ≤ 60 &&
 0 ≤ s && s ≤ 60)
```

Our synthesize tool called Comfusy [15], succeeds, but in addition to the code, it emits the following warning:

```
Synthesis predicate has multiple solutions
for variable assignment: totsec = 0
  Solution 1: h = 0, m = 0, s = 0
  Solution 2: h = -1, m = 59, s = 60
```

The reason for this warning is that the bounds on `m` and `s` are not strict. After correcting the error in the specification, replacing $m \leq 60$ with $m < 60$ and $s \leq 60$ with $s < 60$, Comfusy emits no warnings and generates code corresponding to the following:

```
val (hrs, mns, scs) = {
  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)
}
```

The absence of warnings guarantees that the solution always exists and that it is unique. Note that, if the developer imposes the constraint

```
val (hrs, mns, scs) = choose((h: Int, m: Int, s: Int) ⇒
 h ∗ 3600 + m ∗ 60 + s == totsec &&
 0 ≤ h < 24 &&
 0 ≤ m && m < 60 &&
 0 ≤ s && s < 60)
```

our system emits the following warning:

```
Synthesis predicate is not satisfiable
for variable assignment: totsec = 86400
```

pointing to the fact that the constraint has no solutions when the `totsec` parameter is too large.

B. Synthesis Procedures

Preliminaries. Each of our algorithms works with a set of formulas, Formulas, build from terms, whose set we denote with Terms. Formulas denote truth values, whereas terms and variables denote values from the domain (e.g. integers). We denote the set of variables by Vars. $\mathsf{FV}(q)$ denotes the set of free variables in a formula or a term q. If $\vec{x} = (x_1, \ldots, x_n)$ then $\vec{x}_s$ denotes the set of variables $\{x_1, \ldots, x_n\}$. If q is a term or formula, $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables and $\vec{t} = (t_1, \ldots, t_n)$ is a vector of terms, then $q[\vec{x} := \vec{t}]$ denotes the result of substituting in q the free variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, respectively. Given a substitution $\sigma : \mathsf{FV}(F) \to \mathsf{Terms}$, we write $F\sigma$ for the result of substituting each $x \in \mathsf{FV}(F)$ with $\sigma(x)$. Formulas are interpreted over elements of a first-order structure $\mathcal{D}$ with a countable domain D. We assume that for each $e \in D$ there exists a ground term c_e whose interpretation in $\mathcal{D}$ is e; let $C = \{c_e \mid e \in D\}$. We further assume that if $F \in \mathsf{Formulas}$ then also $F[x := c_e] \in \mathsf{Formulas}$ (the class of formulas is closed under partial grounding with constants).

The choose programming language construct. We integrate into a programming language a construct of the form

$$\vec{r} = \mathsf{choose}(\vec{x} \;\Rightarrow\; F) \tag{1}$$

Here F is a formula (typically represented as a boolean-valued programming language expressions) and $\vec{x} \;\Rightarrow\; F$ denotes a function from $\vec{x}$ to the value of F. Two kinds of variables can appear within F: output variables $\vec{x}$ and parameters $\vec{a}$. The parameters $\vec{a}$ are program variables that are in scope at the point where choose occurs; their values will be known when the statement is executed. In the motivating example, the parameter was the variable `totsec`). Output variables $\vec{x}$ denote values that need to be computed so that F becomes true, and they will be assigned to $\vec{r}$ as a result of the invocation of choose.

Just like an interpreter can be considered as a baseline implementation for a compiler, deploying a decision procedure at run-time can be considered as a baseline for our approach. Such dynamic invocation approach is flexible and useful. However, there are important performance and predictability advantages of an alternative *compilation* approach. Our goal is therefore to explore a compilation approach where a modified decision procedure is invoked at compile time, converting the formula into a solved form.

Definition II.1 (Synthesis Procedure)**.** *We denote an invocation of a synthesis procedure by* $[\![\vec{x}, F]\!] = (\mathsf{pre}, \vec{\Psi})$*. A synthesis procedure takes as input a formula* F *and a vector of variables* $\vec{x}$ *and outputs a pair of*

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

such that the following two implications are valid:

$$(\exists \vec{x}.F) \;\rightarrow\; \mathsf{pre}$$
$$\mathsf{pre} \;\rightarrow\; F[\vec{x} := \vec{\Psi}]$$

Observation II.2. *Because another implication always holds:*

$$F[\vec{x} := \vec{\Psi}] \rightarrow \exists \vec{x}.F$$

the above definition implies that the three formulas are all equivalent: $(\exists \vec{x}.F), \mathsf{pre}, F[\vec{x} := \vec{\Psi}]$*. Consequently, if we can define a function* witn *where for* $\mathsf{witn}(\vec{x}, F) = \vec{\Psi}$ *we have* $\mathsf{FV}(\vec{\Psi}) \subseteq \mathsf{FV}(F) \setminus \vec{x}_s$ *and* $\exists \vec{x}.F$ *implies* $F[\vec{x} := \vec{\Psi}]$*, then we can define a synthesis procedure by*

$$[\![\vec{x}, F]\!] = (F[\vec{x} := \mathsf{witn}(\vec{x}, F)], \mathsf{witn}(\vec{x}, F))$$

Our tool emits the terms $\vec{\Psi}$ in compiler intermediate representation; the standard compiler then processes them along with the rest of the code. We identify the syntax tree of $\vec{\Psi}$ with its meaning as a function from the parameters $\vec{a}$ to the output variables $\vec{x}$. The overall compile-time processing of the choose statement (1) involves the following:

1) emit a non-feasibility warning if the formula $\neg$pre is satisfiable, reporting the counterexample for which the synthesis problem has no solutions;
2) emit a non-uniqueness warning if the formula

$$F \wedge F[\vec{x} := \vec{y}] \wedge \vec{x} \neq \vec{y}$$

   is satisfiable, reporting the values of all free variables as a counterexample showing that there are at least two solutions;
3) as the compiled code, emit the code that behaves as **assert**(pre); $\vec{r}$ = $\vec{\Psi}$

III. Synthesis for Linear Integer Arithmetic

We now describe a synthesis algorithm for linear integer arithmetic, which performs synthesis for quantifier-free formulas of Presburger arithmetic (integer linear arithmetic). In this theory variables range over integers. Terms are linear expressions of the form $c_0 + c_1 x_1 + \ldots + c_n x_n$, $n \geq 0$, c_i is an integer constant and x_i is an integer variable. Atoms are built using the relations $\geq$, $=$ and $|$. The atom $c|t$ is interpreted as true iff the integer constant c divides term t. We use $a < b$ as a shorthand for $a \leq b \wedge \neg(a = b)$. We describe a synthesis algorithm that works for conjunction of literals. An extension to other Boolean combinations can be done using a general approach described in [16].

Pre-processing. We first apply the following pre-processing steps to eliminate negations and divisibility constraints. We remove negations by transforming a formula into its negation-normal form and translating negative literals into equivalent positive ones: $\neg(t_1 \geq t_2)$ is equivalent to $t_2 \geq t_1 + 1$ and $\neg(t_1 = t_2)$ is equivalent to $(t_1 \geq t_2 + 1) \vee (t_2 \geq t_1 + 1)$. We also normalize equalities into the form $t = 0$ and inequalities into the form $t \geq 0$.

We transform divisibility constraints of a form $c|t$ into equalities by adding a fresh variable q. The value obtained for the fresh variable q is ignored in the final synthesized program:

$$[\![\vec{x}, (c|t) \wedge F]\!] = (\mathsf{pre}, \vec{\Psi}),$$
$$\text{where} \qquad (\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1})) = [\![(\vec{x}, q),\ t = c\,q \wedge F]\!]$$

The negation of divisibility $\neg(c|t)$ can be handled in a similar way by introducing two fresh variables q and r:

$$[\![\vec{x}, \neg(c|t) \wedge F]\!] = (\mathsf{pre}, \vec{\Psi}),$$
$$\text{where}$$
$$F' \;\equiv\; t + r = c\,q \wedge 1 \leq r \leq c - 1 \wedge F$$
$$(\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1}, \Psi_{n+2})) \;=\; [\![(\vec{x}, q, r), F']\!]$$

In the rest of this section we assume the input formula F to have no negation or divisibility constraints (these constructs can, however, appear in the generated code and precondition).

$$[\![\_\ ,\ \_]\!] : \bigcup_n \left(\mathsf{Vars}^n \times \mathsf{Formulas} \rightarrow \mathsf{Formulas} \times \mathsf{Terms}^n\right)$$
$$[\![(\vec{y}, \vec{x}), E \wedge F]\!] = (\mathsf{pre}_Y \wedge \mathsf{pre}, (\vec{\Psi}_{Y0}, \vec{\Psi}_X)),$$
$$\text{where}$$
$$(\mathsf{pre}_Y, \vec{\Psi}_Y, \vec{\lambda}) = \mathsf{eqSyn}(\vec{y}, E)$$
$$F' = \mathsf{simplify}(F[\vec{y} := \vec{\Psi}_Y])$$
$$(\mathsf{pre}, (\vec{\Psi}_\lambda, \vec{\Psi}_X)) = [\![(\vec{\lambda}, \vec{x}), F']\!]$$
$$\vec{\Psi}_{Y0} = \vec{\Psi}_Y[\vec{\lambda} := \vec{\Psi}_\lambda]$$

$$\mathsf{eqSyn}: \bigcup_n \mathsf{Vars}^n \times \mathsf{Formulas} \rightarrow \mathsf{Formulas} \times \mathsf{Terms}^n \times \mathsf{Vars}^{n-1}$$
$$\mathsf{eqSyn}(y_1, \Sigma_{i=1}^m \beta_i b_i + \gamma_1 y_1 = 0) \;=$$
$$(\gamma_1 | (\Sigma_{i=1}^m \beta_i b_i),\ -(\Sigma_{i=1}^m \beta_i b_i)/\gamma_1,\ (\ ))$$
$$\mathsf{eqSyn}(y_1, \ldots, y_n, \Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0) \;=$$
$$\mathsf{eqSyn}(y_1, \ldots, y_n,\ t/d + \Sigma_{j=1}^n (\gamma_j/d) y_j = 0),$$
$$\text{where}$$
$$t = \Sigma_{i=1}^m \beta_i b_i$$
$$d = \gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n)$$
$$d > 1$$
$$\mathsf{eqSyn}(y_1, \ldots, y_n, \Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0) \;= (\mathsf{pre}, \vec{\Psi}, \vec{\lambda}),$$
$$\text{where}$$
$$(\vec{s}_1, \ldots, \vec{s}_{n-1}) = \mathsf{linearSet}(\gamma_1, \ldots, \gamma_n)$$
$$(w_1, \ldots, w_n) = \mathsf{particularSol}(\Sigma_{i=1}^m \beta_i b_i, \gamma_1, \ldots, \gamma_n)$$
$$\mathsf{pre} \equiv \gcd(\gamma_1, \ldots, \gamma_n) | (\Sigma_{i=1}^m \beta_i b_i)$$
$$\lambda_1, \ldots, \lambda_{n-1} - \text{fresh variable names}$$
$$\vec{\Psi} = (w_1, \ldots, w_n) + \lambda_1 \vec{s}_1 + \ldots + \lambda_{n-1} \vec{s}_{n-1}$$

Fig. 1. Algorithm for Synthesis Based on Integer Equations

A. Solving Equality Constraints for Synthesis

Because equality constraints are suitable for deterministic elimination of output variables, our procedure groups all equalities from a conjunction and solves them first, one by one. Let E be one such equation, so the entire formula is of the form $E \wedge F$. Let $\vec{y}$ be the output variables that appear in E.

Given an output variable y_1 and E of the form $cy_1 + t = 0$ for $c \neq 0$, a simple way to solve it would be to impose the precondition $c|t$, use the witness $y_1 = -t/c$ in synthesized code, and substitute $-t/c$ instead of y_1 in the remaining formula. However, to keep the equations within linear integer arithmetic, this would require multiplying the remaining equations and disequations in F by c, potentially increasing the sizes of coefficients substantially.

We instead perform synthesis based on one of the improved algorithms for solving integer equations. This algorithm avoids the multiplication of the remaining constraints by simultaneously replacing all n output variables $\vec{y}$ in E with $n-1$ fresh output variables $\vec{\lambda}$. Using this algorithm we obtain the synthesis procedure in Figure 1. An invocation of $\mathsf{eqSyn}(\vec{y}, F)$ is similar to $[\![\vec{y}, F]\!]$ but returns a triple $(\mathsf{pre}, \vec{\Psi}, \vec{\lambda})$, which in addition to the precondition pre and the witness term tuple $\vec{\Psi}$ also has the fresh variables $\vec{\lambda}$.

1) The eqSyn *Synthesis Algorithm:* Consider the application of eqSyn in Figure 1 to the equation $\Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0$. If there is only one output variable, y_1, we directly eliminate it from the equation. Assume therefore $n > 1$. Let $d = \gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n)$. If $d > 1$ we can divide all coefficients by d, so assume $d = 1$.

Our goal is to derive an alternative definition of the set $K = \{\vec{y} \mid \Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0\}$ which will allow a simple and effective computation of elements in K. Note that the set K describes the set of all solutions of a Presburger arithmetic formula.

Here is an overview of an algorithm that describes all the solutions of a formula:

1) obtain a linear set representation of the set

$$S_H = \{\vec{y} \mid \sum_{j=1}^{n} \gamma_j y_j = 0\}$$

of solutions for the homogeneous part using the function linearSet (defined in Theorem III.1) to compute $\vec{s}_1, \ldots, \vec{s}_{n-1}$ such that

$$S_H = \{\vec{y} \mid \exists \lambda_1, \ldots, \lambda_{n-1} \in \mathbb{Z}.\ \vec{y} = \sum_{i=1}^{n-1} \lambda_i \vec{s}_i\}$$

2) find one particular solution, that is, use the function particularSol (defined in Figure 2) to find a vector of terms $\vec{w}$ (containing the parameters b_i) such that $t + \sum_{j=1}^n \gamma_j w_j = 0$ for all values of parameters b_i.

3) return as the solution $\vec{w} + \sum_{i=1}^{n-1} \lambda_i \vec{s}_i$

To see that the algorithm is correct, fix the values of parameters and let $\vec{\gamma} = (\gamma_1, \ldots, \gamma_n)$. From linearity we have $t + \vec{\gamma} \cdot (\vec{w} + \sum_j \lambda_j \vec{s}_j) = t - t + 0 = 0$, which means that each $\vec{w} + \sum_j \lambda_j \vec{s}_j$ is a solution. Conversely, if $\vec{y}$ is a solution of the equation then $\vec{\gamma}(\vec{y} - \vec{w}) = 0$, so $\vec{y} - \vec{w} \in S_H$, which means $\vec{y} - \vec{w} = \sum_{i=1}^n \lambda_i \vec{s}_i$ for some λ_i. Therefore, the set of all solutions of $t + \sum_{j=1}^n \gamma_j w_j = 0$ is the set $\{\vec{w} + \sum_{i=1}^{n-1} \lambda_i \vec{s}_i \mid \lambda_i \in \mathbb{Z}\}$. It remains to define linearSet to find $\vec{s}_i$ and particularSol to find $\vec{w}$.

2) Computing a Linear Set for a Homogeneous Equation: This section describes our version of the algorithm $\mathsf{linearSet}(\gamma_1, \ldots, \gamma_n)$ that computes the set of solutions of an equation $\Sigma_{i=1}^n \gamma_i y_i = 0$. A related algorithm is a component of the Omega test [24].

Theorem III.1. *Let $\gamma_1, \ldots, \gamma_n \in \mathbb{Z}$ be integer coefficients. The set of all solutions of $\Sigma_{i=1}^n \gamma_i y_i = 0$ is described with:*

$$\mathsf{linearSet}(\gamma_1, \ldots, \gamma_n) = (\vec{s}_1, \ldots, \vec{s}_{n-1})$$

where $\vec{s}_j = (K_{1j}, \ldots, K_{nj})$ and the integers K_{ij} are computed as follows:

- *if $i < j$, $K_{ij} = 0$ (the matrix K is lower triangular)*
- *$K_{jj} = \dfrac{\gcd((\gamma_k)_{k \geq j+1})}{\gcd((\gamma_k)_{k \geq j})}$*
- *for each index j, $1 \leq j \leq n-1$, we compute K_{ij} as follows. Consider the equation*

$$\gamma_j K_{jj} + \sum_{i=j+1}^{n} \gamma_i u_{ij} = 0$$

and find any solution. That is, compute

$$(K_{(j+1)j}, \ldots, K_{nj}) = \mathsf{particularSol}(-\gamma_j K_{jj}, \gamma_{j+1}, \ldots, \gamma_n)$$

where particularSol *is given in Figure 2.*

Proof. Let $S_H = \{\vec{y} \mid \Sigma_{i=1}^n \gamma_i y_i = 0\}$ and let

$$S_L = \{\lambda_1 \vec{s}_1 + \ldots + \lambda_n \vec{s}_n \mid \lambda_1, \ldots, \lambda_n \in \mathbb{Z}\} = \left\{ \lambda_1 \begin{pmatrix} K_{11} \\ \vdots \\ K_{n1} \end{pmatrix} + \ldots + \lambda_{n-1} \begin{pmatrix} K_{1(n-1)} \\ \vdots \\ K_{n(n-1)} \end{pmatrix} \middle| \lambda_i \in \mathbb{Z} \right\}$$

We claim $S_H = S_L$.

First we show that each vector $\vec{s}_j$ belongs to S_H. Indeed, by definition of K_{ij} we have $\gamma_j K_{jj} + \sum_{i=j+1}^n \gamma_i K_{ij} = 0$. This means precisely that $\vec{s}_j \in S_H$, by definition of $\vec{s}_j$ and S_H. Next, observe that S_H is closed under linear combinations. Because S_L is the set of linear combinations of vectors $\vec{s}_j$, we have $S_L \subseteq S_H$.

To prove that the converse also holds, let $\vec{y} \in S_H$. We will show that the triangular system of equations $\sum_{i=1}^{n-1} \lambda_i \vec{s}_i = \vec{y}$ has some solution $\lambda_1, \ldots, \lambda_{n-1}$. We start by showing that we can find λ_1. Let $G_1 = \gcd((\gamma_k)_{k \geq 1})$. From $\vec{y} \in S_H$ we have $\Sigma_{i=1}^n \gamma_i y_i = 0$, that is, $G_1(\Sigma_{i=1}^n \beta_i y_i) = 0$ for $\beta_i = \gamma_i / G_1$. This implies $\beta_1 y_1 + \Sigma_{i=2}^n \beta_i y_i = 0$ and $\gcd((\beta_k)_{k \geq 1}) = 1$. Let $G_2 = \gcd((\beta_k)_{k \geq 2})$. From $\beta_1 y_1 + \Sigma_{i=2}^n \beta_i y_i = 0$ we then

obtain $\beta_1 y_1 + G_2(\Sigma_{i=2}^n \beta_i^{'} y_i) = 0$ for $\beta_i' = \beta_i / G_2$. Therefore $y_1 = -G_2(\Sigma_{i=2}^n \beta_i^{'} y_i)/\beta_1$. Because $\gcd(\beta_1, G_2) = 1$ we have $\beta_1 | \Sigma_{i=2}^n \beta_i^{'} y_i$ so we can define the integer $\lambda_1 = -\Sigma_{i=2}^n \beta_i^{'} y_i / \beta_1$ and we have $y_1 = \lambda_1 G_2$. Moreover, note that

$$G_2 = \gcd((\beta_k)_{k\geq 2}) = \gcd((\gamma_k)_{k\geq 2})/G_1 = K_{11}$$

Therefore, $y_1 = \lambda_1 K_{11}$, which ensures that the first equation is satisfied.

Consider now a new vector $\vec{z} = \vec{y} - \lambda_1 \vec{s}_1$. Because $\vec{y} \in S_H$ and and $\vec{s}_1 \in S_H$ also $\vec{z} \in S_H$. Moreover, note that the first component of $\vec{z}$ is 0. We repeat the described procedure on $\vec{z}$ and $\vec{s}_2$. This way we derive the value for an integer α_2 and a new vector that has 0 as the first two components.

We continue with the described procedure until we obtain a vector $\vec{u} \in S_H$ that has all components set to 0 except for the last two. From $\vec{u} \in S_H$ we have $\gamma_{n-1} u_{n-1} + \gamma_n u_n = 0$. Letting $\beta_{n-1} = \gamma_{n-1}/\gcd(\gamma_{n-1}, \gamma_n)$ and $\beta_n = \gamma_n/\gcd(\gamma_{n-1}, \gamma_n)$ we conclude that $\beta_{n-1} u_{n-1} + \beta_n u_n = 0$, so u_{n-1}/β_n is an integer and we let $\lambda_{n-1} = u_{n-1}/\beta_n$. By definitions of β_i it follows $\lambda_{n-1} = u_{n-1} \cdot \gcd(\gamma_{n-1}, \gamma_n)/\gamma_n$. Next, observe the special form of the vector $\vec{s}_{n-1}$: $\vec{s}_{n-1}$ has the form $(0, \ldots, 0, \gamma_n/\gcd(\gamma_{n-1}, \gamma_n), -\gamma_{n-1}/\gcd(\gamma_{n-1}, \gamma_n))$. It is then easy to verify that $\vec{u} = \lambda_{n-1} \vec{s}_{n-1}$.

This procedure shows that every element of S_H can be represented as a linear combination of vectors $\vec{s}_j$, which shows $S_H \subseteq S_L$ and concludes the proof. □

3) Finding a Particular Solution of an Equation: We finally describe the particularSol function to find a solution (as a vector of terms) for an equation $t + \Sigma_{i=1}^n \gamma_i u_i = 0$. We use the Extended Euclidean algorithm (for a detailed description see for example, [3, Figure 31.1]). Given the integers a_1 and a_2, the Extended Euclidean algorithm finds their greatest common divisor d and two integers w_1 and w_2 such that $a_1 w_1 + a_2 w_2 = d$. Our algorithm generalizes the Extended Euclidean Algorithm to arbitrary number of variables and uses it to find a solution of an equation with parameters. We chose the algorithm presented here because of its simplicity. Other algorithms for finding a solution of an equation $t + \Sigma_{i=1}^n \gamma_i u_i = 0$ can be found in [2], [6]. They also run in polynomial time. [2] additionally allows bounded inequality constraints, whereas [6] guarantees that the returned numbers are no larger than the largest of the input coefficients divided by 2.

The equation $t + \Sigma_{i=1}^n \gamma_i u_i = 0$ has a solution iff $\gcd((\gamma_k)_{k\geq 1}) | t$, and the result of particularSol is guaranteed to be correct under this condition. Our synthesis procedure ensures that when the results of this algorithm are used, the condition $\gcd((\gamma_k)_{k\geq 1}) | t$ is satisfied.

We start with the base case where there are only two variables, $t + \gamma_1 u_1 + \gamma_2 u_2 = 0$. By the Extended Euclidean Algorithm let v_1 and v_2 be integers such that $\gamma_1 v_1 + \gamma_2 v_2 = \gcd(\gamma_1, \gamma_2)$. With d we denote $d = \gcd(\gamma_1, \gamma_2)$ and let $r = t/d$. Then one solution is the pair of terms $(-v_1 r, -v_2 r)$:

$$\begin{array}{l} \mathsf{particularSol2}(t, \gamma_1, \gamma_2) = (-v_1 r, -v_2 r), \\ \quad \text{where} \\ \qquad (d, v_1, v_2) = \mathsf{ExtendedEuclid}(\gamma_1, \gamma_2) \\ \qquad r = t/d \end{array}$$

If there are more than two variables, we observe that $\Sigma_{i=2}^n \gamma_i u_i$ is a multiple of $\gcd((\gamma_k)_{k\geq 2})$. We introduce the new variable u' and find a solution of the equation $t + \gamma_1 u_1 + \gcd((\gamma_k)_{k\geq 2}) \cdot u' = 0$ as described above. This way we obtain terms (w_1, w') for (u_1, w'). To derive values of $u_2, \ldots, u_n$ we solve the equation $\Sigma_{i=2}^n \gamma_i u_i = \gcd((\gamma_k)_{k\geq 2}) \cdot w'$. Given that the initial equation was assumed to have a solution, the new equation can also be showed to have a solution. Moreover, it has one variable less, so we can solve it recursively:

$$\begin{array}{l} \mathsf{particularSol}(t, \gamma_1, \ldots, \gamma_n) = (w_1, \ldots, w_n), \\ \quad \text{where} \\ \qquad (w_1, w') = \mathsf{particularSol2}(t, \gamma_1, \gcd((\gamma_k)_{k\geq 2})) \\ \qquad (w_2, \ldots, w_n) = \\ \qquad\quad \mathsf{particularSol}(-\gcd((\gamma_k)_{k\geq 2}) w', \gamma_2, \ldots, \gamma_n) \end{array}$$

Fig. 2. Algorithm for Computing one Solution of the Equation

Example. We demonstrate the process of eliminating equations on an example. Consider the following synthesis problem

$$[\![(x, y, z), 2a - b + 3x + 4y + 8z = 0 \wedge 5x + 4z \leq 2y - b]\!]$$

To eliminate an equation from the formula and to reduce a number of output variables, we first invoke $\mathsf{eqSyn}((x, y, z), 2a - b + 3x + 4y + 8z = 0)$, which works in two phases. In the first phase, we compute the linear set describing a set of solutions of the homogeneous equality $3x + 4y + 8z = 0$. Applying Theorem III.1, the resulting set S_L is:

$$S_L = \left\{ \lambda_1 \begin{pmatrix} 4 \\ -3 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 2 \\ -1 \end{pmatrix} \middle| \lambda_1, \lambda_2 \in \mathbb{Z} \right\}$$

The second phase computes a witness vector $\vec{w}$ and a precondition formula. Applying the procedure described in Section III-A1 results in the vector $\vec{w} = (2a - b, b - 2a, 0)$ and the formula $1 | 2a - b$. Finally, we compute the output of eqSyn applied to $2a - b + 3x + 4y + 8z = 0$: it is a triple consisting of

1) a precondition $1 | 2a - b$
2) a list of terms denoting witnesses for (x, y, z):

$$\begin{array}{l} \Psi_1 = 2a - b + 4\lambda_1 \\ \Psi_2 = b - 2a - 3\lambda_1 + 2\lambda_2 \\ \Psi_3 = -\lambda_2 \end{array}$$

3) a list of fresh variables (λ_1, λ_2).

We then replace each occurrence of x, y and z by the corresponding terms in the rest of the formula. This results in a

new formula $7a - 3b + 13\lambda_1 \leq 4\lambda_2$, that has the same input variables, but the output variables are now λ_1 and λ_2. To find a solution for the initial problem, we let

$$(\mathsf{pre}_X, (\Phi_1, \Phi_2)) = [\![(\lambda_1, \lambda_2), 7a - 3b + 13\lambda_1 \leq 4\lambda_2]\!]$$

Since $1|2a-b$ is a valid formula, we do not add it to the final precondition. Therefore, the final result has the form

$$(\mathsf{pre}_X, (2a - b + 4\Phi_1, b - 2a - 3\Phi_1 + 2\Phi_2, -\Phi_2))$$

B. Solving Inequality Constraints for Synthesis

In the following, we assume that all equalities are already processed and that a formula is a conjunction of inequalities. Dealing with inequalities in the integer case is similar to the case of rational arithmetic: we process variables one by one and proceed further with the resulting formula.

Let x be an output variable that we are processing. Every conjunct can be rewritten in one of the two following forms:

$$\begin{array}{ll} \text{[Lower Bound]} & A_i \leq \alpha_i x \\ \text{[Upper Bound]} & \beta_j x \leq B_j \end{array}$$

As for rational arithmetic, x should be a value which is greater than all lower bounds and smaller than all upper bounds. However, this time we also need to enforce that x must be an integer. Let $a = \max_i \lceil A_i/\alpha_i \rceil$ and $b = \min_j \lfloor B_j/\beta_j \rfloor$. If b is defined (i.e. at least one upper bound exists), we use b as the witness for x, otherwise we use a.

The corresponding formula with which we proceed is a conjunction stating that each lower bound is smaller than every upper bound:

$$\bigwedge_{i,j} \lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \tag{2}$$

Because of the division, floor, and ceiling operators, the above formula is not in integer linear arithmetic. However, in the absence of output variables, it can be evaluated using standard programming language constructs. On the other hand, if the terms A_i and B_j contain output variables, we convert the formula into an equivalent linear integer arithmetic formula as follows.

With lcm we denote the least common multiple. Let $L = \mathrm{lcm}_{i,j}(\alpha_i, \beta_j)$. We introduce new integer linear arithmetic terms $A'_i = \frac{L}{\alpha_i} A_i$ and $B'_j = \frac{L}{\beta_j} B_j$. Using these terms we derive an equivalent integer linear arithmetic formula:

$$\begin{aligned} \lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor &\Leftrightarrow \lceil A'_i/L \rceil \leq \lfloor B'_j/L \rfloor \Leftrightarrow \\ \frac{A'_i}{L} \leq \frac{B'_j - B'_j \bmod L}{L} &\Leftrightarrow B'_j \bmod L \leq B'_j - A'_i \\ &\Leftrightarrow B'_j = L \cdot l_j + k_j \wedge k_j \leq B'_j - A'_i \end{aligned}$$

Formula (2) is then equivalent to

$$\bigwedge_j (B'_j = L \cdot l_j + k_j \wedge \bigwedge_i (k_j \leq B'_j - A'_i))$$

Although this formula belongs to linear integer arithmetic, we still cannot simply apply the synthesizer on that formula. Let $\{1, \ldots, J\}$ be a range of j indices. The newly derived formula contains J equalities and $2 \cdot J$ new variables. The process of eliminating equalities as described in Section III-A will at the end result in a new formula which contains J new output variables and this way we cannot assure termination. Therefore, this is not a suitable approach.

However, we observe that the value of k_j is always bounded: $k_j \in \{0, \ldots, L-1\}$. Thus, if the value of k_j were known, we would have a formula with only J new variables and J additional equations. The equation elimination procedure described before would then result in a formula that has one variable less than the original starting formula, and that would guarantee termination of the approach.

Since the value of each k_j variable is always bounded, there are finitely many $(J \cdot L)$ possible instantiations of k_j variables. Therefore, we need to check for each instantiation of all k_j variables whether it leads to a solution. As soon as a solution is found, the generated code stops and proceeds with the obtained values of output variables. If no solution is found, we raise an exception, because the original formula has no integer solution. This leads to a translation schema that contains $J \cdot L$ conditional expression. In our implementation we generate this code as a loop with constant bounds.

We finish the description of the synthesizer with an example that illustrates the above algorithm.

a) Example.: Consider the formula $2y - b \leq 3x + a \wedge 2x - a \leq 4y + b$ where x and y are output variables and a and b are input variables. If the resulting formula $\lceil 2y - b - a/3 \rceil \leq \lfloor 4y + a + b/2 \rfloor$ has a solution, then the synthesizer emits the value of x to be $\lfloor 4y + a + b/2 \rfloor$. This newly derived formula has only one output variable y, but it is not an integer linear arithmetic formula. It is converted to an equivalent integer linear arithmetic formula $(4y + a + b) \cdot 3 = 6l + k \wedge k \leq 8y + 5a + 5b$, which has three variables: y, k and l. The value of k is bounded: $0 \leq k \leq 5$, so we treat it as a parameter. We start with elimination of the equality: it results in the precondition $6|3a+3b-k$, the list of terms $l = (3a+3b-k)/6+2\alpha, y = \alpha$ and a new variable: α. Using this, the inequality becomes $k - 5a - 5b \leq 8\alpha$. Because α is the only output variable, we can compute it as $\lceil (k - 5a - 5b)/8 \rceil$. The synthesizer finally outputs the following code, which computes values of the initial output variables x and y:

```
val kFound = false
for k = 0 to 5 do {
  val v1 = 3 * a + 3 * b - k
  if (v1 mod 6 == 0){
    val alpha = ((k - 5 * a - 5 * b)/8).ceiling
    val l = (v1 / 6) + 2 * alpha
    val y = alpha
    val kFound = true
    break } }
if (kFound)
  val x = ((4 * y + a + b)/2).floor
else
  throw new Exception("No solution exists")
```

The precondition formula is $\exists k.\ 0 \leq k \leq 5 \wedge\ 6|3a + 3b - k$,

which our synthesizer emits as a loop that checks $6|3a+3b-k$ for $k \in \{0, \ldots, 5\}$ and throws an exception if the precondition is false.

IV. Further Applications of Software Synthesis

We have seen that developing a complete functional synthesis procedure, even for well-understood logic, such as Presburger arithmetic, is non- trivial task. In addition, even when there are synthesis procedures, writing a complete specification is not always feasible. We outline here two approaches to deriving a specification: programming by example and inferring the specification from type constraints.

A. Programming by Example

Instead of writing code, the user provides a list of relevant examples and the synthesis tool automatically generates a program. In this way, the examples can be seen as an easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it still might not correspond to the user's intentions. Examples are, by nature, an incomplete specification. The programming by example paradigm is suitable for *interactive* and *incremental* synthesis approach. If the user is not happy with the returned program, she provides more examples, thus refining her specification, and a synthesis tool automatically synthesizes a program that satisfies all the given examples.

To be able to understand a programming by example better, we developed a working tool called StriSynth [10]. To avoid unnecessary language and implementation details, we restrict ourselves to the synthesis of a small, yet still real world and useful, programs. We achieve this by focusing on the synthesis of scripts.

Many tedious and repetitive tasks, including file manipulations and organizing data, can easily be automated by writing a program in some scripting language. However, such languages usually require a good knowledge of regular expressions, and often their syntax does not correspond to modern high level programming languages. Additionally, small errors in the scripts can lead to malicious behavior, such as data loss. Consider, for instance an attempt at removing all backup emacs files with the command `rm * ~`. For these reasons, many end-users search for help on on-line forums when they need to write some script. We analyzed on-line forums and mailing lists, and noticed that when a non-expert user seeks for help in writing a script, she usually provides a few illustrative examples that convey her intentions about what this script is supposed to do.

The following example is from a StackOverflow post[1], where the users discuss complex and challenging regular expressions. The user asked for a script that will create a link from every item in a directory. To better illustrate her intentions, the user provided two examples: for given files

```
Document1.docx
Document2.docx
```

[1]http://stackoverflow.com/q/800813/2137996

the script should output

```
<A HREF='Document1.docx'>Document1</A>
<A HREF='Document2.docx'>Document2</A>
```

Following a discussion of the forum users a following script can do the required task using the popular `sed` tool:

```
sed/\(^[a-zA-Z0-9]+\)\.\([a-z]+\)/
 \<a href\=\'\1\.\2\' \>\1\<\/a\>/g
```

While it was very easy to express the initial intentions of the user by providing examples, the resulting script is arguably less readable, even for such a simple problem. Furthermore, small changes might require an entirely new regular expression to be written.

Using our tool StriSynth, it is enough to provide one good illustrative example, such as:

```
"a.doc" ==> "<a href='a.doc'>a</a>"
```

and the tool learns a correct script.

We conducted a performance study on a preliminary version of the tool [10]. We collected 60 real world tasks, from various forums and mailing-lists. By testing our tool on those benchmarks, we showed that in most cases, the user only needs to provide a few examples (one or two) and StriSynth generates the required script. The synthesis process took less than one second in most cases.

B. Program Synthesis from Type Constraints

In our previous work we established the theoretical foundation for type-driven synthesis, and developed a tool called InSynth [11], [12]. InSynth is a tool that automatically generates code snippets from specifications implicitly given in the form of type constraints, similarly as in the case of auto-completion of code. By invoking InSynth, the user asks our tool to suggest a list of suitable code fragments for the given program point. InSynth displays a ranked list of suggested code snippets for that program point and the user can chose the best solution. In practical evaluation, our technique scales to programs with thousands of visible declarations in scope and succeeds in 0.5 seconds to suggest program expressions of interest.

Finding a code snippet of the right type is closely related to the type inhabitation problem [31] where we ask whether for a type environment and some calculus, there exists an inhabitant of a given type. As providing the correct answer fast is a very important requirement, we introduce a succinct representation for type judgments that merges types into equivalence classes to reduce the search space. Furthermore, it is not enough to provide a code fragment of the correct type - we should also be able to guess what the user had in mind. For this purpose we introduce a ranking of solutions based on a weight function. Moreover, the weight function is used to direct the search for type inhabitants. The weight function is defined on all the symbols appearing in the program, and it is based on the proximity to the program point at which InSynth was invoked. Additionally, the weight is also partially derived from a corpus of code. We have found our system to be useful for synthesizing code fragments for common programming tasks,

and we believe it is a good platform for exploring software synthesis techniques.

C. Program Repair through Program Synthesis

In our on-going work we extend the theoretical foundations of type-driven synthesis to type-driven program repair. We implemented an algorithm that automatically repairs code expressions based on the provided almost-correct code [25]. At the core of our algorithm is a graph construction that expresses the relationships between the language's types and methods.

As an illustration, a programmer might expect the Java code

```
BufferedReader br = new BufferedReader("file.txt");
```

to open a file called "file.txt". However, this code will not compile since `BufferedReader` accepts only a `Reader` interface implementation as an argument.

We developed a tool called Winston [25] which automatically repairs code expressions based on the hinted structure of the ill-typed code - we call such an input expression, a backbone expression. Winston finds well typed expressions that are as close as possible to the given (potentially) ill-typed expression.

Additionally, Winston can also be seen as a synthesis tool. In the light of the program repair, the synthesis aspect of Winston can be considered as a repair of the empty expression. A user does not need to provide a backbone expression – it is sufficient to declare a variable of an arbitrary type. Based on that type Winston can synthesize corresponding code fragments. The synthesized code has the given type and it can contain user defined values, as well as methods from the API.

At the core of Winston's algorithm is a graph construction that expresses the relationships between the language's types and methods.

V. Conclusions

In the software synthesis research there was an amazing progress from initial attempts to automatically generate simple sorting algorithms, to a methodology which found their way towards the millions of users. We believe that the success of synthesis tools is closely linked to advances and major improvements that we have witnessed in the last decade in the field of automated reasoning and SMT solvers. We have shown in this paper how to extend a decision procedure into a synthesis procedure. There is no general method and each such extension for some logic requires a deep understand of its decision procedure. In order for synthesis to scale we described several approaches where the user does not need to write down a complete specification, but the specification is automatically inferred through an interaction with the user.

References

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[2] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

[4] A. Cypher and D.C. Halbert. *Watch what I Do: Programming by Demonstration*. MIT Press, 1993.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[6] David Ford and George Havas. A new algorithm and refined bounds for extended gcd computation. In *ANTS*, pages 145–150, 1996.

[7] Cordell Green. Application of theorem proving to problem solving. In *Proc. Int'l. Joint Conf. Artificial Intelligence*, pages 219–239. Morgan Kaufmann, 1969.

[8] Sumit Gulwani. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012. Invited talk paper.

[9] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[10] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. Strisynth: Synthesis for live programming. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 701–704, 2015.

[11] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[12] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[13] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[14] Ali Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of z3: Integrating smt and programming. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer Berlin / Heidelberg, 2011.

[15] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, pages 430–433, 2010.

[16] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

[17] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Commun. ACM*, 55(2):103–111, 2012.

[18] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[19] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[20] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[21] Michał Moskal. *Satisfiability Modulo Software*. PhD thesis, University of Wrocław, 2009.

[22] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[23] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM.

[24] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[25] Alex Reinking and Ruzica Piskac. A type-directed approach to program repair. In *Proceedings of the 27th International Conference on Computer Aided Verification CAV 2015, San Francisco, USA, July 18-24, 2015.*, 2015.

[26] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[27] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[28] Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bit-vector arithmetic. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 499–513, 2012.

[29] Saurabh Srivastava, Sumit Gulwani, and Jeff Foster. From program verification to program synthesis. In *POPL*, 2010.

[30] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

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

[32] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.