Results and Analysis of SyGuS-Comp'15

Results and Analysis of SyGuS-Comp'15

Rajeev Alur, University of Pennsylvania
Dana Fisman, University of Pennsylvania
Rishabh Singh, Microsoft Research
Armando Solar-Lezama, Massachusetts Institute of Technology

Syntax-Guided Synthesis (SyGuS) is the computational problem of finding an implementation f that meets both a semantic constraint given by a logical formula φ in a background theory T, and a syntactic constraint given by a grammar G, which specifies the allowed set of candidate implementations. Such a synthesis problem can be formally defined in SyGuS-IF, a language that is built on top of SMT-LIB.

The *Syntax-Guided Synthesis Competition (SyGuS-Comp)* is an effort to facilitate, bring together and accelerate research and development of efficient solvers for SyGuS by providing a platform for evaluating different synthesis techniques on a comprehensive set of benchmarks. In this year's competition we added two specialized tracks: a track for conditional linear arithmetic, where the grammar need not be specified and is implicitly assumed to be that of the LIA logic of SMT-LIB, and a track for invariant synthesis problems, with special constructs conforming to the structure of an invariant synthesis problem. This paper presents and analyzes the results of SyGuS-Comp'15.

1 Introduction

Syntax guided synthesis is a form of synthesis where in addition to a specification, the user provides a syntactic description of the space of possible programs that the synthesizer is allowed to consider. This style of synthesis was popularized by the Sketch project [19] and has become an important approach underpinning a variety of synthesis efforts (e.g. [9, 20, 22]). Within the common framework of syntax guided synthesis, however, there are a number of possible approaches that can be used to solve a given synthesis problem, and until recently, research on the relative merits of these different approaches had been hampered by the difficulty of making direct comparisons about algorithms implemented in different systems and with very different interfaces.

The Syntax Guided Synthesis competition (SyGuS-Comp) was started as a way to accelerate advances in the field by offering a uniform standard notation and formalism for describing these problems, allowing for direct comparisons between different solution methods. The SyGuS formalism [13] is derived from the SMT-LIB format [5], a popular format for describing SMT problems, with the goal of making it easy to learn for users and easy to support for synthesizer developers. In the short time that the formalism has been in public circulation, it has already performed well in many of these goals. It has provided important insights into the relative merits of different algorithms [1, 2] which have been exploited to help develop new algorithms [8], and has been used to evaluate brand new algorithms [3, 10, 14]. Beyond synthesizer developers, there is a growing community of users that is coalescing around the formalism.

The SyGuS problem is formally defined as the problem of finding an implementation of a function f that meets both a semantic constraint given by a logical formula φ in a background theory T, and a syntactic constraint given by a grammar G. More concretely, let φ be a formula in some theory T that involves some free variables $x_0, \ldots x_i$ and uses an unknown function f. The goal of synthesis is to discover an implementation f_{imp} such that replacing f_{imp} for f in φ makes the formula valid for all values of the free variables.

$$\forall x_0, \ldots x_i.\ \varphi[f/f_{imp}]$$

Černý, Kuncak, Madhusudan (Eds.): Fourth Workshop on Synthesis (SYNT 2015)
EPTCS 202, 2016, pp. 3–26, doi:10.4204/EPTCS.202.3

The syntactic restriction implies that the function f_{imp} must belong to the grammar G also provided by the user.

Example As an example, consider the problem of finding a bit-vector function that returns a bit-vector with a 1 in the position of the least significant zero in its input bit-vector and zero everywhere else. This problem is defined in terms of the theory of bit-vectors, which includes operations such as bitwise `and`, bitwise `or`, as well as left and right shifts and basic arithmetic among bit-vectors. The example problem can be formalized in terms of two constraints that must be satisfied by the desired function. The first constraint is that if x is not zero, $f(x)$ should have a one in the same position as x has a zero, so bitwise `and` (`&`) of $f(x)$ and the bitwise negation of x ($\sim$`x`) should not be zero.

$$\forall x.\ x > 0 \Rightarrow (f(x)\ \texttt{\&}\ \sim x) > 0$$

The way we enforce that there is a single one and that it corresponds to the least significant zero is by enforcing that if we shift $f(x)$ right by any amount, then bitwise anding with the bitwise negation of x will now yield zero.

$$\forall x, y.\ y > 0 \Rightarrow (f(x) \gg y\ \texttt{\&}\ \sim x) = 0$$

These two constraints can be expressed succinctly in SyGuS with the following notation.

```
(set-logic BV)

(synth-fun f ((x (BitVec 32))) (BitVec 32)
             ((Start (BitVec 32) (x 0 1
                                  (bvand Start Start)
                                  (bvor Start Start)
                                  (bvnot Start)
                                  (bvadd Start Start)))))

(declare-var x (BitVec 32))
(declare-var y (BitVec 32))

(constraint (=> (bvult 0 x) (bvult 0 (bvand (f x) (bvnot x)))))
(constraint (=> (bvult 0 y) (= 0 (bvand (bvshr (f x) y) (bvnot x)))))

(check-synth)
```

The `set-logic` directive indicates that the constraints should be interpreted in terms of the theory of bitvectors. The directive `declare-var` is used to declare `x` and `y` as 32-bit bitvector variables. The constraints are introduced with the directive `constraint`, and `check-synth` marks the end of the problem and prompts the synthesizer to solve for the missing function. Crucially, in order for the synthesizer to generate f, it needs a grammar, which is provided as part of the `synth-fun` directive. In this example, we have significantly reduced the space of possible functions by restricting the search to expressions involving bitwise `and`, `or`, bitwise negation, sum and the constants 0 and 1, instead of asking the system to consider completely arbitrary expressions in the theory of bitvectors.

1.1 Conditional Linear Integer Arithmetic Track

For problems where the grammar consists of the set of all possible integer linear arithmetic terms, it is sometimes possible to apply specialized solution techniques that exploit the information that decision procedures for integer linear arithmetic are able to produce. For this reason, the 2015 SyGuS competition included a separate track where the grammar for all the unknown functions was assumed to be the entire theory of Integer Linear Arithmetic with ITE conditionals.

Example As a simple example, consider the problem of synthesizing a function `max2` that produces the maximum of two integers. The problem can be specified with the constraint below:

```
(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int)
(declare-var x Int)
(declare-var y Int)
(constraint (>= (max2 x y) x))
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y)) (or (= y (max2 x y)))))
(check-synth)
```

Note that the definition of the unknown function `max2` does not include a grammar this time, but because the problem is defined in the theory of linear integer arithmetic (`LIA`), the default grammar consists of all the operations available in the theory.

1.2 Invariant Synthesis Track

One of the main applications of SyGuS is invariant synthesis. For this problem, the goal is to discover an invariant that makes the verification condition for a given loop valid. Such a problem can be easily encoded in SyGuS, but invariant synthesis problems have structure that some solution algorithms are able to exploit and that can be lost when encoding it into SyGuS. For this reason, the 2015 installment of the competition also included a separate track for invariant synthesis problems where the additional structure is made apparent. In the invariant synthesis version of the SyGuS format, the constraints are separated into pre-condition, post-condition and transition relation, and the grammar for the unknown invariant is assumed to be the same as that for the conditional linear arithmetic track.

Example For example, consider the following simple loop.

```
Pre: i >= 0  and  j=j0 and i=i0;
while(i > 0){
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

Suppose we want to prove that the value of j at the end of the loop equals the value of i + j at the beginning of the loop. The verification condition for this loop would check that (a) the precondition implies the invariant, (b) that the invariant is inductive, so if it holds before an iteration and the loop condition is true, then it will hold after that iteration, and (c) that the invariant together with the negation of the loop condition implies the postcondition. All of these constraints can be expressed in the standard

SyGuS format, but they can be expressed more concisely using the extensions explicitly defined for this purpose. Specifically, the encoding will be as follows.

```
(set-logic LIA)

(synth-inv inv-f ((i Int) (j Int) (i0 Int) (j0 Int)))

(declare-primed-var i0 Int)
(declare-primed-var j0 Int)
(declare-primed-var i  Int)
(declare-primed-var j  Int)

(define-fun pre-f ((i Int) (j Int) (i0 Int) (j0 Int)) Bool
                  (and (>= i 0) (and (= i i0) (= j j0))))

(define-fun trans-f ((i Int) (j Int) (i0 Int) (j0 Int)
                     (i! Int) (j! Int) (i0! Int) (j0! Int)) Bool
                     (and (and (= i! (- i 1)) (= j! (+ j 1)))
                          (and (= i0! i0) (= j0! j0))))

(define-fun post-f ((i Int) (j Int) (i0 Int) (j0 Int)) Bool
                   (= j (+ j0 i0)))

(inv-constraint inv-f pre-f trans-f post-f)

(check-synth)
```

The directive `(declare-primed-var i)` is equivalent to separately declaring `i` and `i!`, where the primed version of the variables is used to distinguish their value before and after the loop body. Just like in the earlier example, the function to be synthesized `inv_f` does not include a grammar, so the entire `LIA` grammar is assumed. Here the return type is also not given and is assumed to be `Bool` since invariant are assumed to be predicates. The constraint `inv-constraint` is syntactic sugar for the full verification condition involving the invariant, precondition, postcondition and transition function.

1.3 SyGuS-Comp'14 summary

The first SyGuS competition, SyGuS-Comp'14 consisted of a single track — the general track — in which the benchmark provides the grammar describing the desired syntactic restrictions for that benchmark. The background theory could be either linear interger arithmetic or bitvectors. Five solvers competed in SyGuS-Comp'14. The solver who won the first place was the ENUMERATIVE solver which solved 126 out of 241 benchmarks.

1.4 SyGuS-Comp'15 summary

The 2015 instance of SyGuS-Comp was the second iteration of the competition and the first iteration to include the separate conditional linear integer arithmetic and invariant synthesis tracks. As elaborated in Section 2, there were a total of eight solvers submitted to the competition which represent a range of solution strategies. In the rest of the paper, we describe the details of the benchmarks used for the

competition, the solution approaches used by each of the participants and the results of the competition on each of the different categories of benchmarks.

2 Competition Settings

2.1 Participating Benchmarks

We collected several benchmarks from various sources using an open call for benchmarks, from which we sampled 309 benchmarks for the General Track, 73 benchmarks for the Invariant Synthesis Track, and 67 benchmarks for the Conditional Linear Arithmetic track. These benchmarks can broadly be classified into following 13 categories (11 categories for the General Track). The number of benchmarks in each of these categories and the contributors for the new set of benchmarks is shown in Table 1.

- **Array Benchmarks**: This category consists of benchmarks that involve synthesizing a function over an integer array of a bounded size. Since the current SyGuS-IF format does not support arrays, we represent them using an ordered sequence of integer variables. One major class of benchmarks in this category is `array-search`, which requires to synthesize a loop-free function to find an appropriate index for insertion of a given value into a sorted array of size n. Another major class of benchmark is `array-sum-m-n`, which requires to synthesize a function to find first two elements in an array of size n whose sum is greater than m.
- **Let Benchmarks**: The benchmarks in this category use the `let` construct as specified in the SyGuS-IF format. The let expressions allow synthesizers to represent multiple occurrences of common subexpressions succinctly, which helps in improving the scalability. The let expressions in specification constraints can be desugared, but they can not be desugared when used in grammar productions. For example, the production `T := (let [z=U] z + z)` denotes sum of two identical subexpressions built using addition operator, but the same can not be specified using a context-free grammar.
- **Invariant Generation Benchmarks (Bounded and Unbounded Integers)**: This category of benchmarks consists of loop invariant synthesis problems from the domain of program verification. Some of these benchmarks are obtained from the Competition of Software Verification (SV-COMP) and others are from the literature on invariant synthesis. It involves two sub-categories: one that uses unbounded integers and second that bounds the range of integers that can be used for the invariant expression.
- **ICFP Benchmarks**: This benchmarks in this category are some of the challenging problems taken from the 2013 ICFP Programming Competition on program synthesis. These benchmarks involve synthesizing bit-vector functions from a domain-specific language of bit-vectors consisting of primitive bitwise operators such as shift, not, add, or, xor etc. The language also consists of a constrained comparison operator and a fold operator. The correctness specification for these benchmarks is provided using 64-bit input-output bit-vector examples.
- **Hacker's Delight Benchmarks**: This category involves benchmarks that are derived from 20 different bit-manipulation problems from the Hackers Delight book. For these benchmarks, there are 3 increasingly challenging levels of grammars ($d0$, $d1$, and $d5$) that are provided for the unknown function, where the level $d0$ consists of only operators that are necessary for the unknown function and the level $d5$ consists of complete bit-vector grammar.

Benchmark Cateogry	Number of Benchmarks	Contributors (in 2015)	
Arrays	31		
Bitvectors	5		
CompilerOptimizations	21	Nissim Ofek (Yale):	21
HackersDelight	44		
ICFP	50		
Integers	34	Shambwaditya Saha (UIUC):	14
InvariantGeneration	28		
InvariantGenerationUbdd	28		
Let	15		
MotionPlanning	12	Sarah Chasins (UC Berkeley):	12
MultipleFunctions	41		
INV Track	73	Pranav Garg (UIUC):	51
LIA Track	67		

Table 1: The number of benchmarks in each individual category used in SyGuS-COMP 2015 together with the number of new contributions and their contributors.

- **Integer Benchmarks**: These benchmarks involve synthesizing functions that involve complex branching structure over linear arithmetic expressions. For example, the `max` benchmarks require to synthesize a function using comparison operator to compute maximum of n integers.
- **BitVector Benchmarks**: This category of benchmarks involve synthesizing a complex function over a set of Boolean values represented using a bit-vector. For example, the `parity` benchmarks compute the parity of a given set of Boolean values, and the `Morton` benchmarks compute the Morton numbers.
- **Compiler Optimization Benchmarks**: The benchmarks in this category come from the domain of compiler optimization where the goal is to synthesize a simpler expression (constrained by a grammar) that is functionally equivalent with a given complicated expression.
- **Multiple Functions Benchmarks**: This class of benchmarks consists of synthesis problems that involve synthesizing multiple unknown functions that satisfy a given set of constraints.
- **Motion Planning Benchmarks**: The benchmarks in this category come from the domain of robot motion planning, where the task is to synthesize a function to control robot movement (constrained by a grammar of possible movements) that reach from one point in space to another while maintaining some invariant constraints such as no collision.
- **LIA Track Benchmarks**: The benchmarks in this category come from the General Track categories such as integer benchmarks, array benchmarks, and motion planning benchmarks. The grammar for the unknown functions for this category comprises of expressions from the entire theory of Integer Linear Arithmetic with ITE conditionals as described in Section 3.2.
- **INV Track Benchmarks**: The benchmarks in this category come from the domain of program verification, where the pre-condition, transition function, and the post-condition are specified using explicit constructs in the SyGuS-IF format.

	Solvers							
Tracks	ALCHEMIST-CS	ALCHEMIST-CSDT	CVC4-1.5-SYGUS	ENUMERATIVE	ICE-DT	SKETCH-AC	SOSYTOAST	STOCHASTIC
LIA	1	1	1	0	0	0	0	0
INV	1	0	1	0	1	0	0	0
General	0	0	1	1	0	2	2	1

Table 2: Solvers participating in each track

2.2 Participating Solvers

A total of eight solvers were submitted to this year's competition, two of the solvers were submitted with two configurations. Table 2 summarizes which solver participated in which track and in how many configurations. Thus a total of 3 solvers participated in both the LIA and the INV tracks. In the general track participated 5 solvers, out of which 2 with two configurations. Table 2 lists all the submitted solvers and their authors.

Six of the solvers are based on the CEGIS approach: the *enumerative* solver (ENUMERATIVE), the *stochastic* solver (STOCHASTIC), the *Sketch* solver (SKETCH-AC), the *abstract solution Analyzing Synthesis Tool* (SOSYTOAST), the *Alchemist CS* solver (ALCHEMIST-CS), and the *Alchemis CSDT* solver (ALCHEMIST-CSDT). The approaches of these solvers differ in the way the candidate expression is chosen.

The *enumerative* solver enumerates the candidate solutions by increasing length, and tries the smallest candidate that was not refuted so far. In building new expressions it uses a classification to equivalence classes based on correctness on the current set of examples. For further details on the enumerative solver see [1, 2].

The *stochastic* solver chooses the candidate expression by a probabilistic walk on a graph where nodes are expressions and edges capture single-edits of the expression parse tree. It uses the Metropolis-Hastings Algorithm using a score function that is inversely proportional to the number of instances in the current set of examples, on which the expression is incorrect. For further details see [1, 2].

The *Sketch* solver follows a symbolic approach to CEGIS and uses an SMT solvers to choose the next candidate expression [18]. The *Sketch Adaptive-Concretization* combines symbolic and explicit CEGIS approaches [8] by randomly concretizing highly influential unknowns. This approaches lends itself to parallelism. Indeed SKETCH-AC is the only solver that used all four cores in this year's competition (all the other solvers used a single core).

The SOSYTOAST solver refines the enumerative CEGIS approach by considering *abstract expressions*; these are expressions that combine grammar non-terminals and concrete symbols. The idea is to use an SMT solver to refute impossible abstract expressions, instead of refuting all their respective concretizations. A check of the feasbility of an abstract expression to be a solution involves qunatifer alternation. The SMT calls are thus bounded by a small time unit (10 seconds) [15].

The tools ALCHEMIST-CS [12] and ALCHEMIST-CSDT [17] are specialized solvers for the LIA track. They are base on the Alchemist [16], they both synthesize nested *if-then-else* expressions, and com-

Solver	Authors
ALCHEMIST-CS	Daniel Neider (UIUC), Shambwaditya Saha (UIUC) and P. Madhusudan (UIUC)
ALCHEMIST-CSDT	Shambwaditya Saha (UIUC), Daniel Neider (UIUC) and P. Madhusudan (UIUC)
CVC4-1.5-SYGUS	Andrew Reynolds (EPFL), Viktor Kuncak (EPFL), Cesare Tinelli (Univ. of Iowa), Clark Barrett (NYU), Morgan Deters (NYU) and Tim King (Verimag)
ENUMERATIVE	Abhishek Udupa (Penn)
ICE-DT	Daniel Neider (UIUC), P. Madhusudan (UIUC) and Pranav Garg (UIUC)
SKETCH-AC	Jinseong Jeon (UMD), Xiaokang Qiu (MIT), Armando Solar-Lezama (MIT) and Jeffrey S. Foster (UMD)
SOSYTOAST	Heinz Riener (DLR) and Ruediger Ehlers (DLR)
STOCHASTIC	Mukund Raghothama (Penn)

Table 3: Submitted solvers and their authors

bine enumeration and constraint-solver based technique to find leaf expressions The ALCHEMIST-CSDT works in two modes, in the first all variables (qunatified and unquantified) are assumed to be integers, and in the second variables may be either integers or Booleans. The ALCHEMIST-CS tool can only solve *point-wise* SyGuS problems. A SyGuS problem is said to be point-wise if the following holds: let $\mathscr{F}$ be the class of all functions that are solutions to the specification; then any function h that maps each input to an output consistent to some function in $\mathscr{F}$, is also a solution to the specification.

The ICE-DT [11] tool is a specialized solver for the invariant synthesis track. It builds on the ICE learning approach [6] — an invariant synthesis approach in which the teacher responses to a given hypothesis may be concrete examples, counterexamples or implication. ICE-DT extends the ICE approach by using decision trees as suggested in [7].

The CVC4-1.5-SYGUS solver is the only solver that is implemented inside an SMT solver [14]. It reduces the synthesis problem to an unsatisfibility of quantified SMT formulae. Indeed, as noted in [2, 1], for linear integer arithmetic problems, suppose the synthesis problem involves two variables x and y, then it can be articulated as the following quantified SMT satisfiability query: $\exists a,b,c.\ \forall x,y.\ \varphi[f/ax+by+c]$. Since traditional SMT solvers are focused on instnatiation-based methods to show *unsatisfiability*, the CVC4-1.5-SYGUS tool reformulates LIA problems as unsatisfiability of the synthesis conjecture $\forall f, \exists x_0, x_1, \ldots, x_i, \neg\varphi[f, x_0, x_1, \ldots, x_i]$. For non-LIA problems, the CVC4-1.5-SYGUS uses an encoding of the syntax restrictions using first order variables and uses a deep embedding into an extension of the background theory T with a theory of algebraic data types.

2.3 Experimental Setup

The solvers were run on the StarExec platform [21] with a dedicated cluster of 12 nodes, where each node consisted of two 4-core 2.4GHz Intel processors with 256GB RAM and a 1TB hard drive. The memory usage limit of each solver run was set to 128GB. The wallclock time unit was set to 3600 seconds (thus, a solver that used all cores could consume at most 14400 seconds cpu time).

The solutions that the solvers produce are being checked for both syntactic and semantic correctness. That is, a first post-processor checks that the produced expression adheres to the grammar specified in the given benchmark, and if this check passes, a second post-processor checks that the solution adheres to semantic constraints given in the benchmark (by invoking an SMT solver).

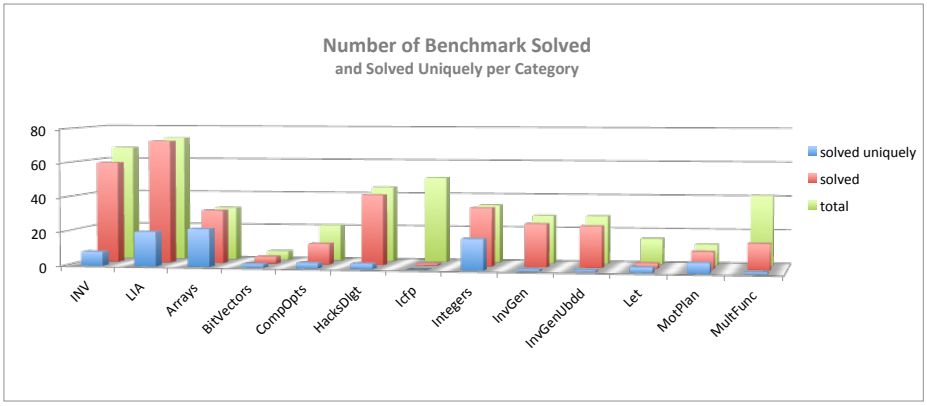

Figure 1: Results on the number of solved benchmarks.

3 Competition Results and Analysis

In this section we present the results of SyGuS-COMP'15. We start with an overview of the results in subsection 3.1. In subsection 3.2 we try to understand the different features a certain benchmark may or may not have, and analyze each benchmark category in terms of the mentioned features. In subsection 3.3 we provide detailed results of the competition — going category by category, we present for each benchmark the range of times the solver took to solve that benchmark and which solver was the fastest, and the range of obtained expression sizes, and which solver produced the smallest expression. Finally, we conclude in subsection 3.4 with some interesting observations and conjectures.

3.1 Results Overview

Figure 1 shows for each track and category (a) the total number of benchmarks (in green), (b) the number of benchmarks solved by at least one solver (in red), and (c) the number of benchmarks solved by only one of the solvers (in blue).

Figure 2 on the top shows for each track and category the number of benchmarks solved by each of the solvers. It is evident from the figure that on the INV track, ICE-DT solved more instances than the others; it solved 57 out of 67 benchmarks. The Alchemist solver solved 53 benchmarks in the INV track.

In the LIA track the leading solver is CVC4-1.5-SYGUS, which solved 70 out of 73 benchmarks. The second place goes to ALCHEMIST-CSDT which solved 47 instances.

CVC4-1.5-SYGUS is also the winner in the general track, in which it solved 179 out of 309 benchmarks. The second place in this track goes to the ENUMERATIVE solver, the winner of last year's competition, which solved 139 instances. The third place goes to the STOCHASTIC solver, which solved 109 instances.

Figure 2 on the bottom shows for each track and category the number of benchmarks solved uniquely by each of the solvers. We say that a solver solved a benchmark uniquely if it is the only solver to solve that benchmark. It is evident that on the INV track, the ICE-DT solver solved more benchmarks uniquely, whereas in the LIA track the CVC4-1.5-SYGUS solver solved more benchmarks uniquely. In the general track, both CVC4-1.5-SYGUS and ENUMERATIVE solved many benchmarks that all the other solvers did not. It is interesting to see that for each benchmark category, there is a clear winner in terms of the number of benchmarks solved uniquely.

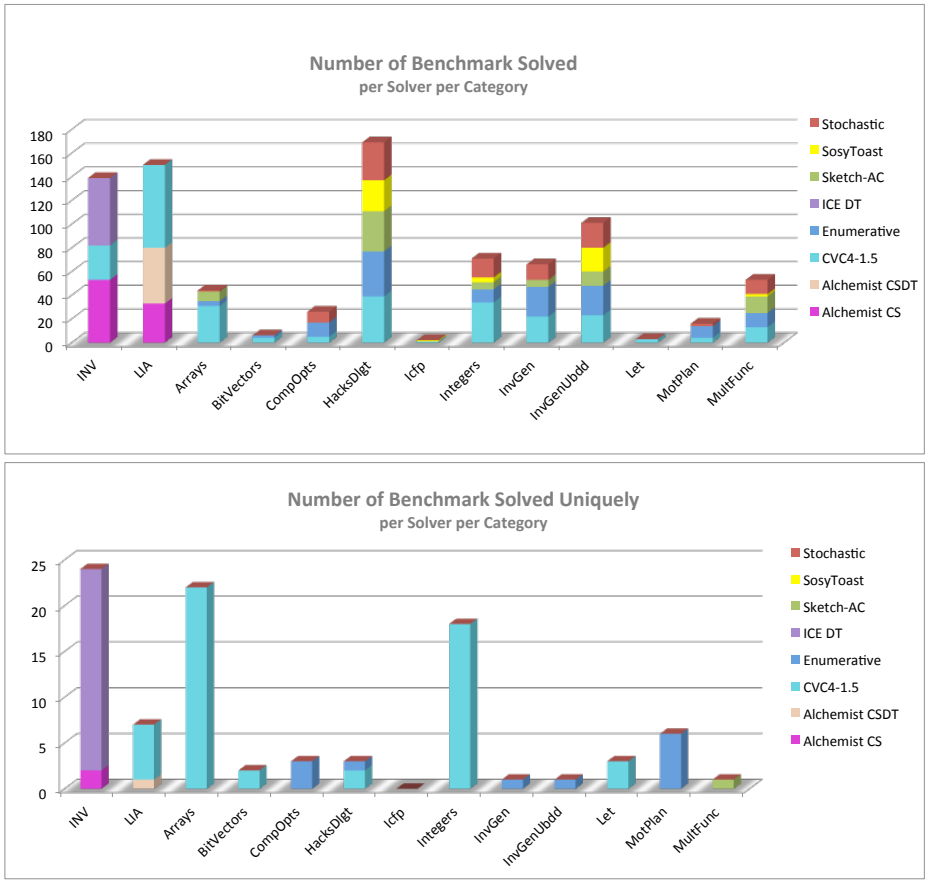

Figure 2: Results on the number of solved benchmarks per solver per category.

3.2 Benchmarks' Features

Since the solving strategies for synthesizers can be greatly influenced by the problem representation, we quantify here some different features that are exhibited by the benchmarks from different categories in Table 3.2.[1] We consider here four main categories: 1) whether the invocation of the unknown function is always present with the same set of arguments (*single invocation*) or there are multiple different function invocations (*multiple invocation*), 2) whether there is a single unknown function (*single unknown*) or multiple unknown functions (*multiple unknown*), 3) whether the specification constraint is complete (*complete spec*) or partially specified using input-output examples (*partial spec*), and finally 4) whether the grammar for the unknown function is *restricted*, *semi-general*, or *general*.

Function Invocation We say a benchmark belongs to the *single invocation* class if the unknown function is called (possibly multiple times) with the same set of arguments in the specification constraint. For example, consider the specification constraint for the `max4` benchmark. This benchmark belongs to the single invocation class because the `max4` function is always called with the same set of arguments `x`, `y`, `z`, and `w`.

```
(constraint (>= (max4 x y z w) x))
(constraint (>= (max4 x y z w) y))
(constraint (>= (max4 x y z w) z))
(constraint (>= (max4 x y z w) w))
(constraint (or (= x (max4 x y z w))
            (or (= y (max4 x y z w))
            (or (= z (max4 x y z w))
                (= w (max4 x y z w))))))
```

Similarly, we say a benchmark belongs to the *multiple invocation* class if the unknown function is called multiple times with different set of arguments. For example, consider the `icfp_7_10` benchmark, where the unknown function `f` is called multiple times with different set of arguments.

```
(constraint (= (f #x1be88589ba201842) #xe4177a7645dfe7bd))
(constraint (= (f #x49ea2ae53e599623) #x93d455ca7cb32c46))
(constraint (= (f #xea82cc5e6104247d) #xea82cc5e6104247d))
(constraint (= (f #x75820d31bed79b87) #xeb041a637daf370e))
(constraint (= (f #xe682665199ee31a8) #x197d99ae6611ce57))
```

Number of Unknown Functions We say a benchmark belongs to the class of *single unknown* class if there is only one unkown function that needs to be synthesized. For example, the `max4` and `icfp_7_10` benchmarks above require to synthesize a single function `max4` and `f` respectively. The benchmarks that require to synthesize multiple unknown functions are said to belong to the *multiple unkown* class. For example, consider the specification constraint from the `s8` benchmark that requires to synthesize three unknown functions `f1`, `f2`, and `f3`, such that their sum is equal to a given linear arithmetic expression.

```
(constraint (= (+ (+ (f1 x y z) (f2 x y z)) (f3 x y z)) (+ (+ x y) z)))
(constraint (= (f2 x y z) (- y 1)))
```

[1] The mentioned features may not exist in *all* benchmarks of the category, but they are present in most of the benchmarks in that category.

Benchmark Category	Function Invocations	# Unkown Functions	Specification Constraint	Grammar Generality
Array	Single	Single	Complete	Restricted
Let	Single	Multiple	Complete	Restricted
Invariant Generation	Multiple	Single	Complete	General
ICFP	Multiple	Single	Partial	General
HackerDel-d0	Single	Single	Complete	Restricted
HackerDel-d1	Single	Single	Complete	Semi-General
HackerDel-d5	Single	Single	Complete	General
Integer	Single	Single	Complete	Semi-General
Bitvector	Single	Single	Complete	Semi-General
CompilerOpts	Single	Single	Complete	Restricted
MultipleFuncs	Multiple	Multiple	Partial	Restricted
Motion Planning	Multiple	Single	Complete	Semi-General
LIA Track	Single	Single	Complete	General
INV Track	Multiple	Single	Partial	General

Table 4: The list of features present in benchmarks from different categories.

We note that multiple unknown functions can be theoretically represented as a single unknown function by taking the union of respective grammars of the functions and adding a new `Start` non-terminal. But, we still distinguish benchmarks into different categories based on number of unknown functions as some solvers might exploit the additional structural information present in the benchmarks.

Specification Constraint Since the specification language for the SyGuS-IF is based on constraints and is therefore quite general, the specification for different benchmarks can be defined with varying levels of completeness. We say a benchmark belongs to the *partial spec* class if the specification constraint allows for multiple possible *semantic* solutions for the unkown functions. We say two functions are semantically different if there exists some input for which they produce different outputs. For example, the specification for the `icfp_7_10` benchmark is partial as it is specified using a set of input-output examples. The specification for the `s8` benchmark shown above is also partial as it allows for multiple possible functions `f1`, `f2`, and `f3`.

On the other hand, we say a benchmark belongs to the *complete spec* class if the complete functional specification of the unknown function is provided. For example, consider the specification for the compiler optimization benchmark `qm_loop_1`, where the constraint on the unknown function `qm-loop` fully specifies the result of the function for every input.

```
(set-logic LIA)

(define-fun qm ((a Int) (b Int)) Int (ite (< a 0) b a))

(synth-fun qm-loop ((x Int)) Int
    ((Start Int (x 0 1 3
                 (- Start Start)
                 (+ Start Start)
```

```
                (qm Start Start)))))

(declare-var x Int)

(constraint (= (qm-loop x) (ite (<= x 0) 3 (- x 1))))

(check-synth)
```

Grammar Generality We divide the benchmarks into three categories based on the generality of the grammars for the unknown functions. The *restricted grammar* class consists of benchmarks where the grammars are restricted to only contain operators and variables that are strictly necessary for the unknown function. The second class of grammar, *semi-general grammar*, consists of a few more choices for operators and rules that might not be necessarily needed for synthesizing the unknown function. The third class of grammar, *general grammar*, consists of a very general grammar that consists of all possible operators and variables from a given theory. For example, consider the grammars for the `hd-17-d0-prog` (restricted grammar), `hd-17-d1-prog` (semi-general), and `hd-17-d5-prog` (general grammar) benchmarks where all the constraints are same except for the generality of the grammar. The benchmark `hd-17-prog` is taken from the Hacker's Delight book that needs to synthesize a bitvector function to turn-off the rightmost contiguous string of 1 bits in the input bitvector `x`.

```
;; hd-17-d0-prog

(synth-fun f ((x (BitVec 32))) (BitVec 32)
    ((Start (BitVec 32) ((bvand Start Start)
                         (bvadd Start Start)
                         (bvsub Start Start)
                         (bvor Start Start)
                          x
                         #x00000001))))



;; hd-17-d1-prog

(synth-fun f ((x (BitVec 32))) (BitVec 32)
    ((Start (BitVec 32) ((bvand Start Start)
                         (bvadd Start Start)
                         (bvxor Start Start)
                         (bvsub Start Start)
                         (bvor Start Start)
                         (bvnot Start)
                         (bvneg Start)
                         x
                         #x00000001
                         #x00000000
                         #xFFFFFFFF))))
```

```
;; hd-17-d5-prog

(synth-fun f ((x (BitVec 32))) (BitVec 32)
    ((Start (BitVec 32) ((bvnot Start)
                         (bvxor Start Start)
                         (bvand Start Start)
                         (bvor Start Start)
                         (bvneg Start)
                         (bvadd Start Start)
                         (bvmul Start Start)
                         (bvudiv Start Start)
                         (bvurem Start Start)
                         (bvlshr Start Start)
                         (bvashr Start Start)
                         (bvshl Start Start)
                         (bvsdiv Start Start)
                         (bvsrem Start Start)
                         (bvsub Start Start)
                         x
                         #x0000001F
                         #x00000001
                         #x00000000
                         #xFFFFFFFF))))
```

The grammars for the unknown functions in the Conditional Linear Arithmetic (LIA) track and the Invariant Synthesis (INV) track belong to the class of general grammars, where the following grammar is used for all unknown functions in the benchmarks.

```
(synth-fun f ((x1 Int) ... (xn Int)) Int
    ((Start Int (StartInt))
     (StartInt Int (x1 ... xn ConstantInt
        (+ StartInt StartInt)
        (- StartInt StartInt)
        (* StartInt ConstantInt)
        (* ConstantInt StartInt)
        (div StartInt ConstantInt)
        (mod StartInt ConstantInt)
        (ite StartBool StartInt StartInt)))
     (ConstantInt (Constant Int))
     (StartBool Bool (true false
        (and StartBool StartBool)
        (or StartBool StartBool)
        (=> StartBool StartBool)
        (xor StartBool StartBool)
        (xnor StartBool StartBool)
```

```
(nand StartBool StartBool)
(nor StartBool StartBool)
(iff StartBool StartBool)
(not StartBool)
(= StartBool StartBool)
(<= StartInt StartInt)
(= StartInt StartInt)
(>= StartInt StartInt)
(> StartInt StartInt)
(< StartInt StartInt)))))
```

3.3 Detailed Results

In the following section we show the results of the competition from the benchmark's perspective. For a given benchmark we would like to know: how many solvers solved it, what is the min and max time to solve, what are the min and max size of the expressions produced, which solver solved the benchmark the fastest, and which solver produced the smallest expression.

We represents the results per benchmarks in groups organized per tracks and categories. For instance, Fig. 3 shows all benchmarks of the INV tracks (first half in the upper figure, and second half in the lower figure). The black bars show the range of the time to solve among the different solvers in psuedo logarithmic scale (as indicated on the upper part of the y-axis). Inspect for instance benchmark `anpf-new.sl`. The black bar indicates that the fastest solver to solve it used between 10 to 30 second, and the slowest did not solve it within the given time bound (3600 second). The black number above the black bar indicates the exact number of seconds (floor-rounded to the nearest second) it took the slowest solver to solve a benchmark (and ∞ if at least one solver exceeded the time bound). Thus, we can see that the slowest solver to solve `array-new.sl` took 167 seconds to solve it. The white number at the lower part of the bar indicates the time of the fastest solver to solve that benchmark. Thus, we can see that the fastest solver to solve `anpf-new.sl` required 21 second to do so, and the fastest solver to solve `array-new.sl` took less than a second. The colored squares/rectangles next to the lower part of the black bar, indicate which solvers were the fastest to solve that benchmark (according to the solvers' legend at the top). Here, *fastest* means in the same logarithmic scale as the absolute fastest solver. For instance, we can see that ALCHEMIST-CS was the fastest to solve `anpf-new.sl`, that both ALCHEMIST-CS and ICE-DT solved `cegar-new.sl` in less than a second, and that all solvers (ICE-DT,ALCHEMIST-CS, and CVC4-1.5-SYGUS) solved `dec-new.sl` in less than a second.

Similarly, the gray bars indicate the range of expression sizes in psuedo logarithmic scales (as indicated on the lower part of the y-axis), where the size of an expression is determined by the number of nodes in its parse tree. The black number at the bottom of the gray bar indicates the exact size expression of the largest solution (or ∞ if it exceeded 1000), and the white number at the top of the gray bar indicates the exact size expression of the smallest solution. The colored squares/rectangles next to the upper part of the gray bar indicates which solvers (according to the legend) produced the smallest expression (where *smallest* means in the same logarithmic scale as the absolute smallest expression). For instance, for `tacas.sl` the smallest expression produced had size 7, the biggest expression had size 121, and the solver which produced the smallest expression is CVC4-1.5-SYGUS. For `treax1.sl` all solvers produced an expression of size smaller than 10, and the absolute value of the smallest expression was 7.

Finally, at the top of the figure above each benchmark there is a number indicating the number of solvers that solved that benchmark. For instance, one solver solved `anpf-new.sl`, two solvers

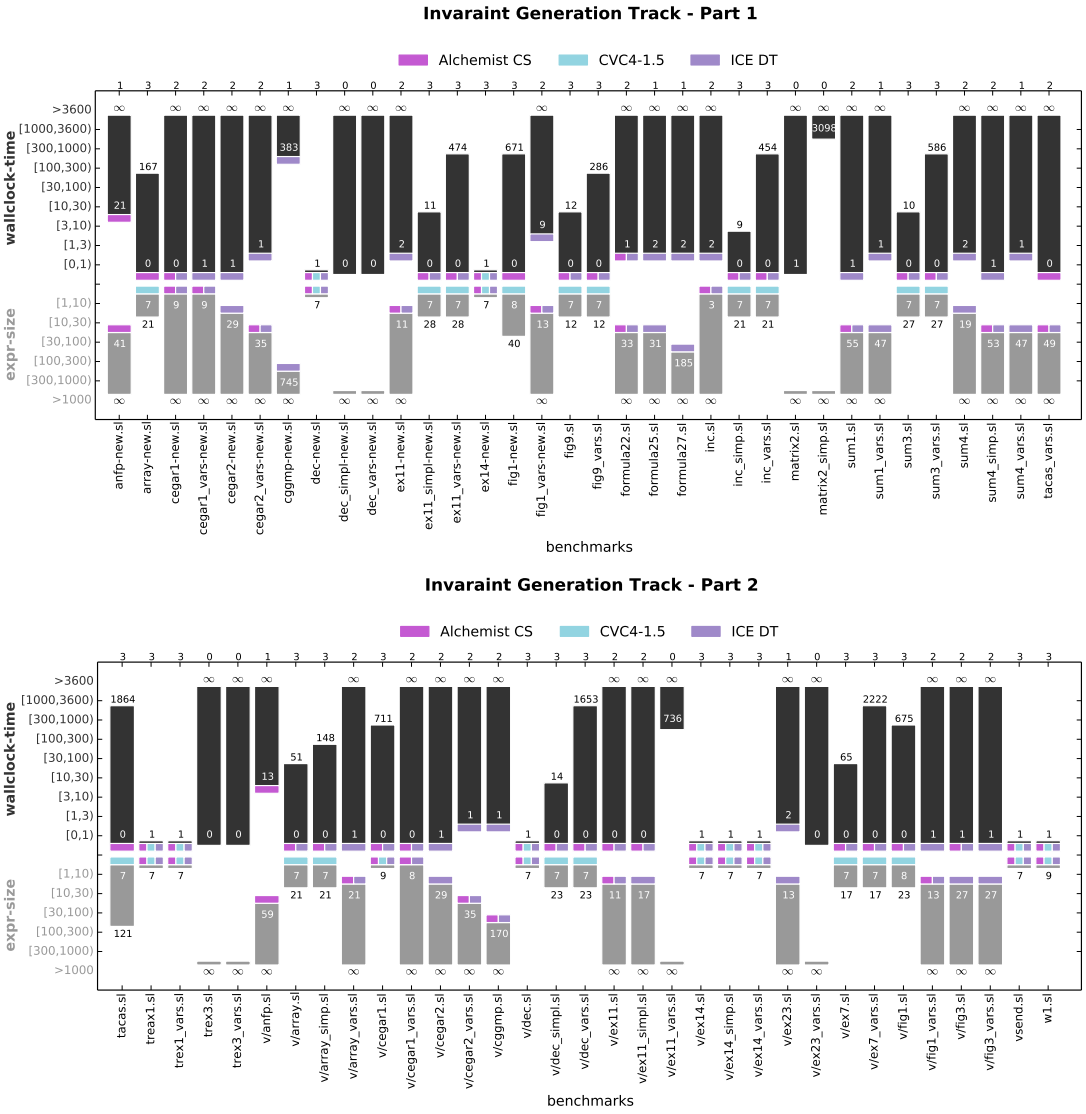

Figure 3: Evaluation of INV benchmarks.

solved `cegar-new.sl`, three solvers solved `array-new.sl`, and no solver solved `matrix2.sl` or `matrix2-simp.sl`. Note that the reason `matrix2.sl` has 1 as the lower time bound, is that that is the time to terminate rather than the time to solve. Thus, one of the solvers has terminated within less than 1 second, but either it did not produce a result, or it produced an incorrect result. When no solver produced a correct result, there will be no colored squares/rectangles next to the lower parts of the bars.

Figures 3-11 presents the results of each benchmark in the competition. The results are organized per category. The results for ICFP category are not included, since only one benchmark was solved. This benchmark is the `icfp_28_10.sl` benchmark, it was solved by CVC4-1.5-SYGUS and SOSYTOAST, in both configurations. The two solvers (in the overall three configurations) terminated in less than two seconds, producing an expression of size two. It seems that the solvers find it difficult to handle the benchmarks in this category since the specification constraints are too partial.

3.4 Observations

Correlating the number of instances solved by the different solvers (as depicted in Fig. 2) and the features a certain category has (as detailed in Table. 3.2) the following observations can be made:

- The CVC4-1.5-SYGUS solver preforms best on benchmarks with single invocation and a single unknown.
- The ICE-DT solver, designated to solve invariant synthesis problems, does best on invariant synthesis problems, which are multiple invocation.
- The ENUMERATIVE and STOCHASTIC solver preform well on multiple invocation problems (of the general track).
- The SKETCH-AC solver preforms well on benchmarks with a restricted grammar.

We note that we have tested the solvers submitted to the general track on the instances of the INV and LIA track translated to general SyGuS. None of them solved any of those instances. For most of the solvers the problem seemed to be lack of support for arbitrary constants — the translation to general SyGuS included the grammar term `(Constant Int)` which allows any integer to appear in the specification, and the solvers for the general track that mostly build on guiding the search according to the syntactic structure do not support this construct. The reason CVC4-1.5-SYGUS failed on these benchmarks is that it wasn't designed to support the less common Boolean connectives such as `nand` and `xnor` that the translation to general SyGuS allowed in order to be as permissive as possible. For next year's competition we plan to test the general solvers on a restricted, yet general, subset of LIA, e.g. one consisting of just zero and one as constants and a small subset of the Boolean operators from which the others can be derived (e.g. `and`, `or`, and `not`).

With regard to size expressions we note that the ENUMERATIVE solver by definition always produces the smallest expression. Inspecting figures 3-11 we observe that in many cases most solvers produced expressions of relatively the same size. Though the CVC4-1.5-SYGUS solver often produces very large expressions. For instance, on the `array_search_7.s` benchmark, in which SKETCH-AC produced an expression of size 31, CVC4-1.5-SYGUS's expression was of size 3,311, and on the `LinExpr_inv1_ex.sl`, in which both ENUMERATIVE and SKETCH-AC produced an expression of size 14, STOCHASTIC produced an expression of size 42, CVC4-1.5-SYGUS produced an expression of size 4,022. The largest expression produced was for benchmark `array_search_15.sl`. Its size is 1,843,271. It is the solution of CVC4-1.5-SYGUS which is the only solver to solve this benchmark.

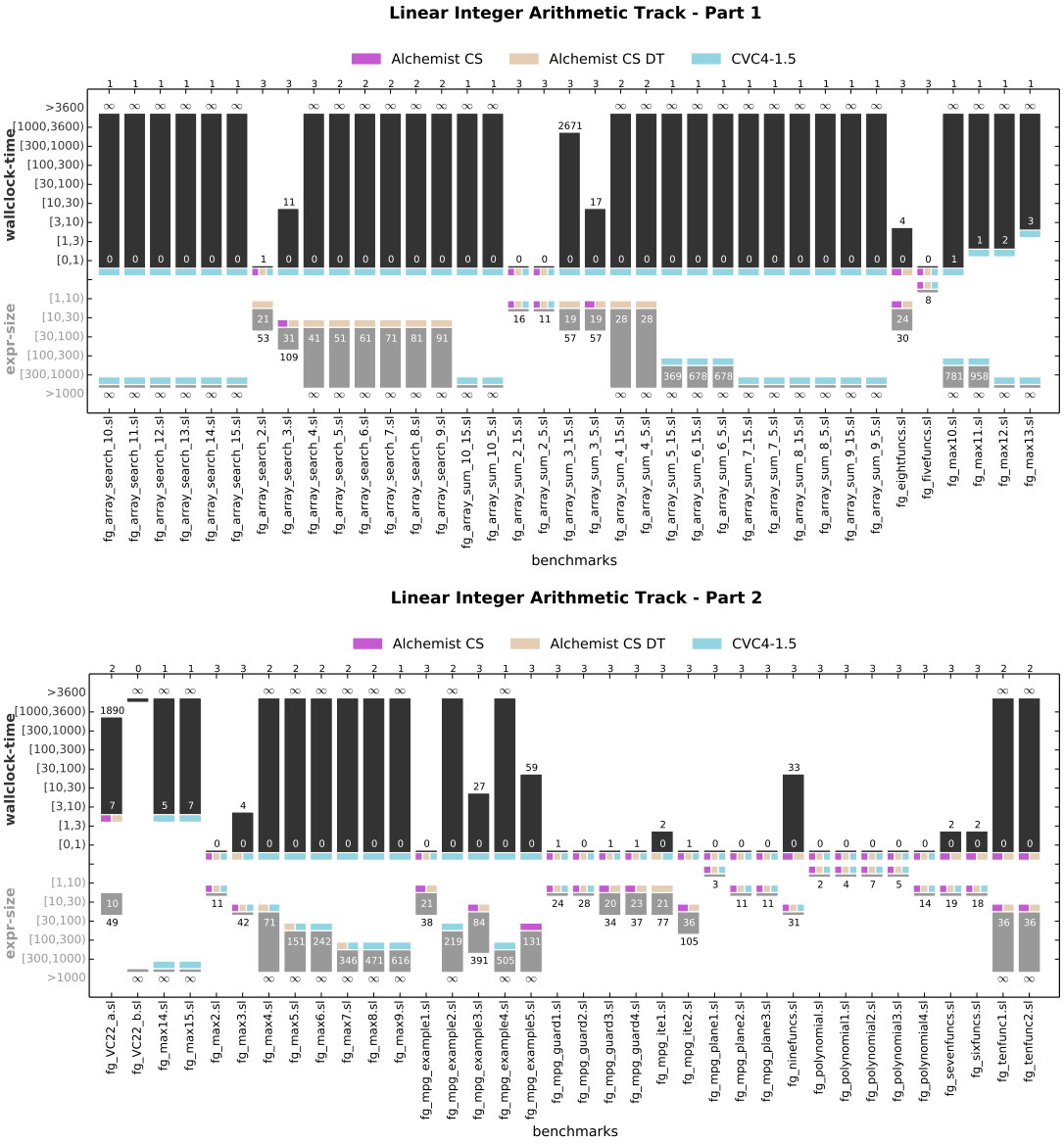

Figure 4: Evaluation of LIA benchmarks.

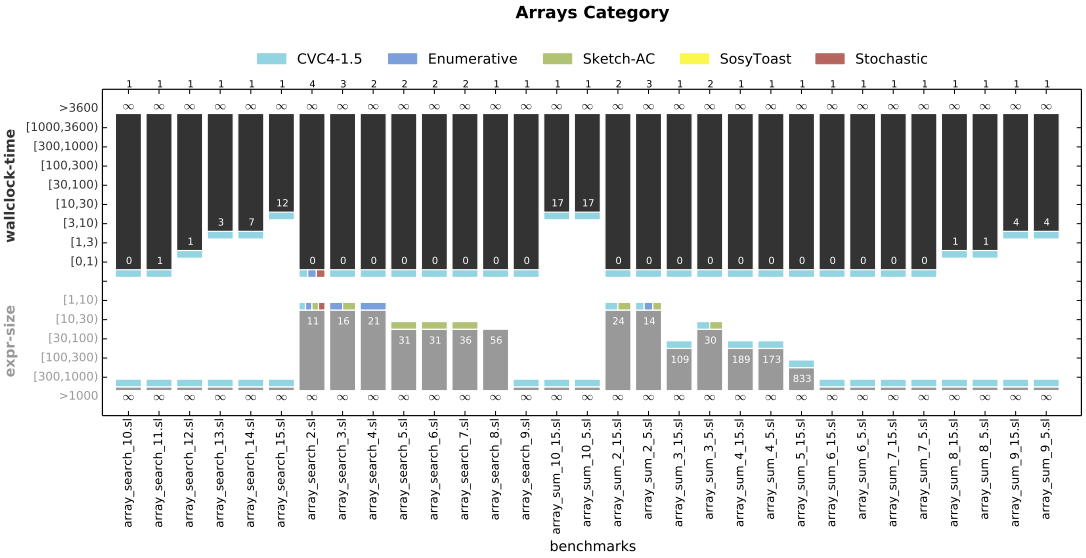

Figure 5: Evaluation of Arrays benchmarks.

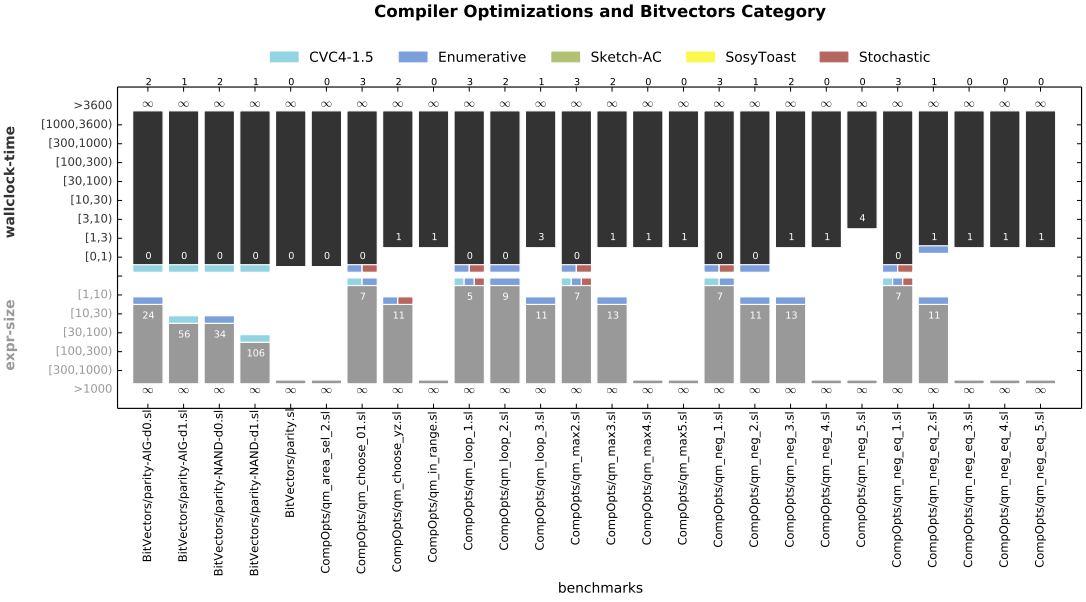

Figure 6: Evaluation of Compiler Optimization and Bitvectors benchmarks.

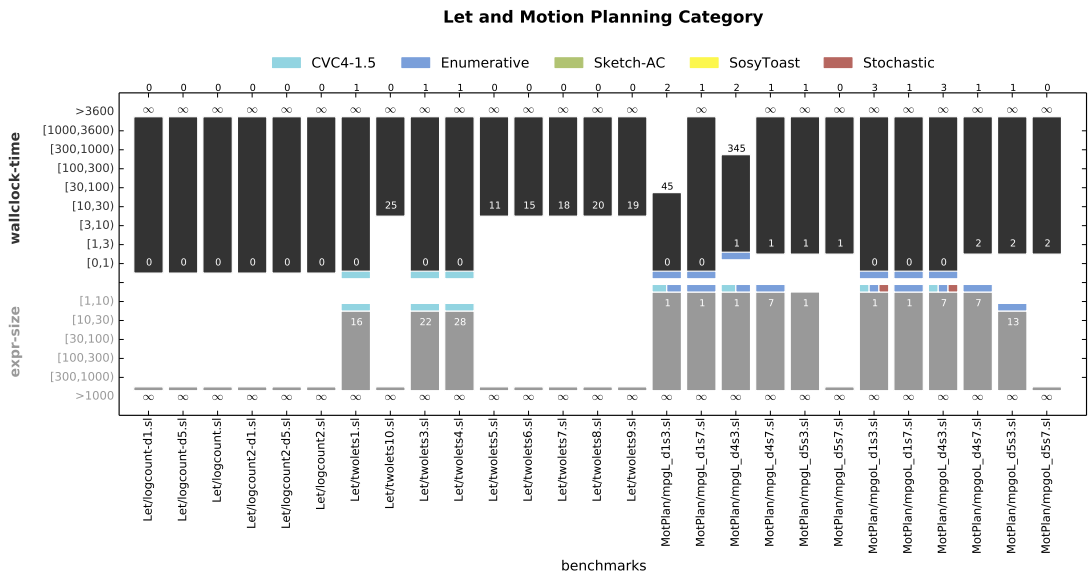

Figure 7: Evaluation of Let and Motion Planning benchmarks.

4 Discussion

The syntax-guided synthesis competition of this year (SyGuS-Comp'15) added two special tracks, above the general track of last year's competition: The LIA track where the background theory is conditional linear integer arithmetic and no grammar is given, instead the grammar is implicitly assumed to be that of the LIA logic of SMT-LIB; and the INV track (also using LIA and no syntactic constraints) consisting of invariant synthesis problems and using special constructs that convey the structure of an invariant synthesis problem.

A total of eight solvers were submitted to this year's competition. On the LIA and INV tracks competed three solvers (with an overlap of two), and in the general track competed five solvers in seven configurations. The winner of the LIA track and general track was the CVC4-1.5-SYGUS solver [14], which is the only solver to participate in all tracks and the first solver to implement a syntax-guided synthesis engine inside an SMT solver. The winner of the INV track was the ICE-DT solver [11], a special solver for invariant synthesis problems.

We classified the benchmarks participating in the competition into several categories and analyzed the features a certain category has or has not. Correlating the results with the features we observe that CVC4-1.5-SYGUS preforms best on benchmarks where there is a single function to be synthesized, and that function is always invoked using the same set of parameters. On benchmarks with multiple functions to synthesize or in which functions are invoked with different parameters, the ENUMERATIVE and STOCHASTIC solver (which won first and second place in last year's competition) seem to preform better.

We hope this report sheds some light on the correlations of solver's performance and benchmark features, and can be of help to improve the development of future solvers. In next year's competition we are considering devoting a track to benchmarks where obtaining a small expression is more crucial than obtaining it fast, as is the case for instance, in compiler optimization problems.

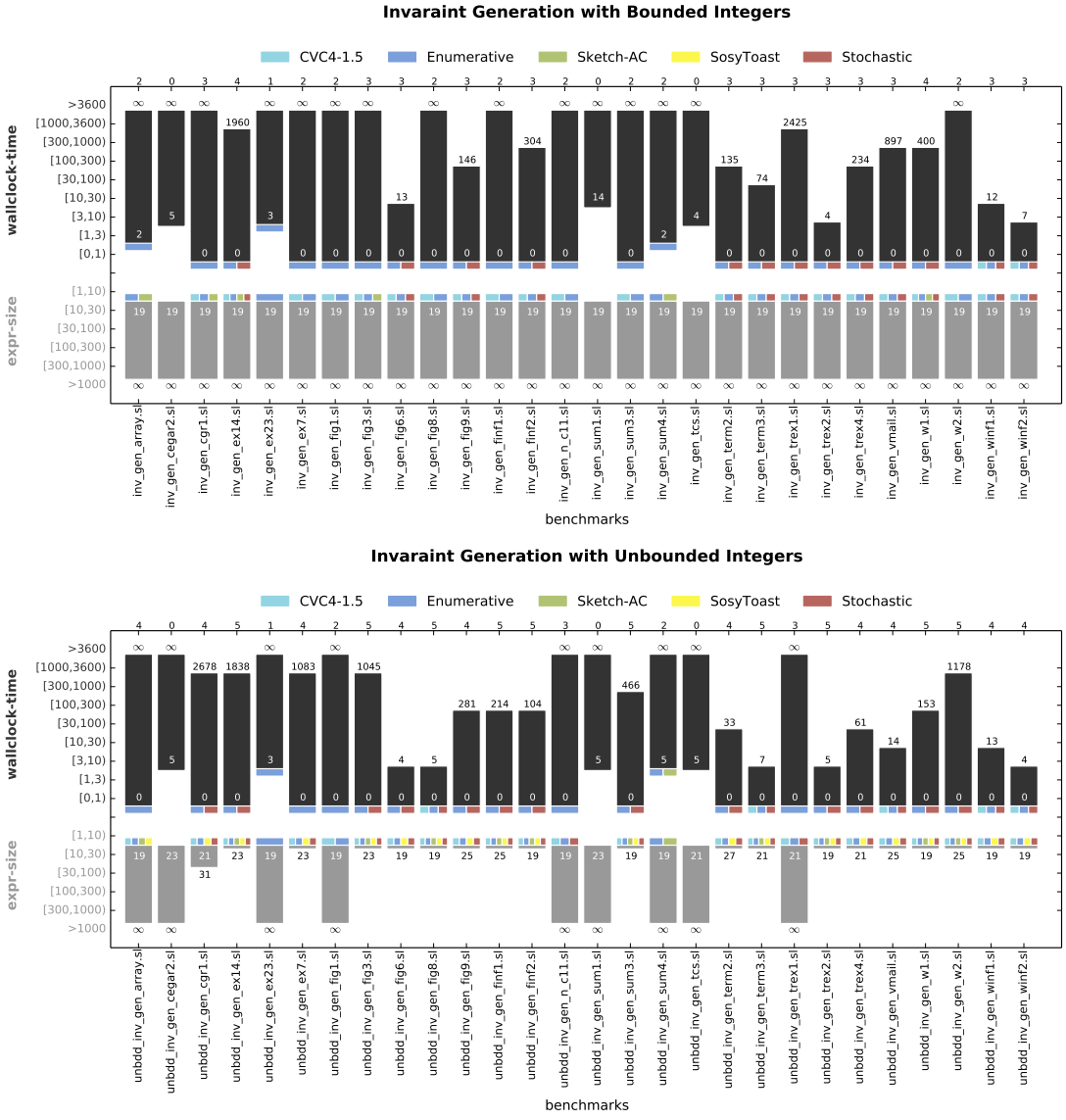

Figure 8: Evaluation of Invariant Generations with/without unbounded integers benchmarks. (These are benchmarks from the general track.)

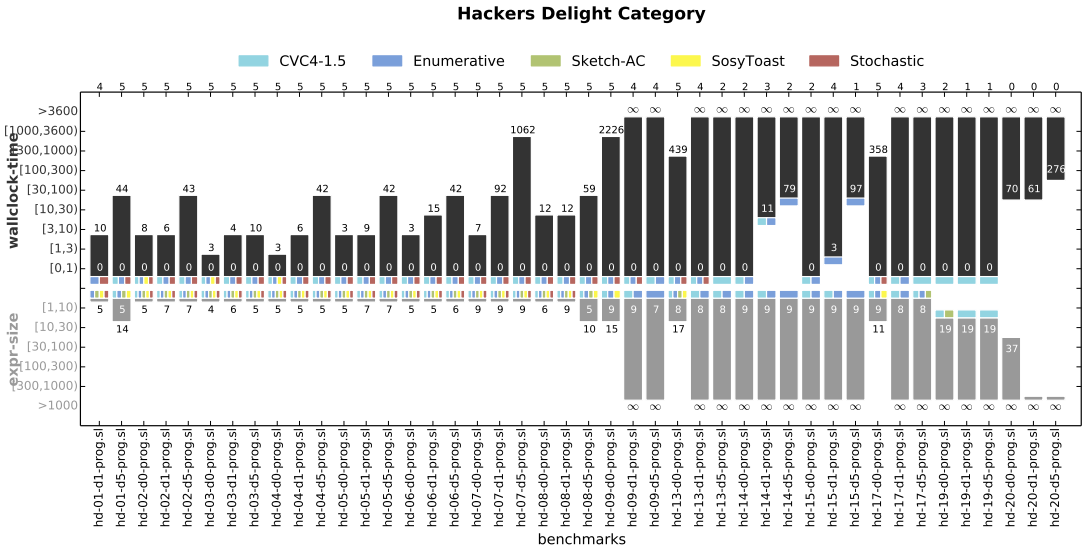

Figure 9: Evaluation of Hacker's Delight benchmarks.

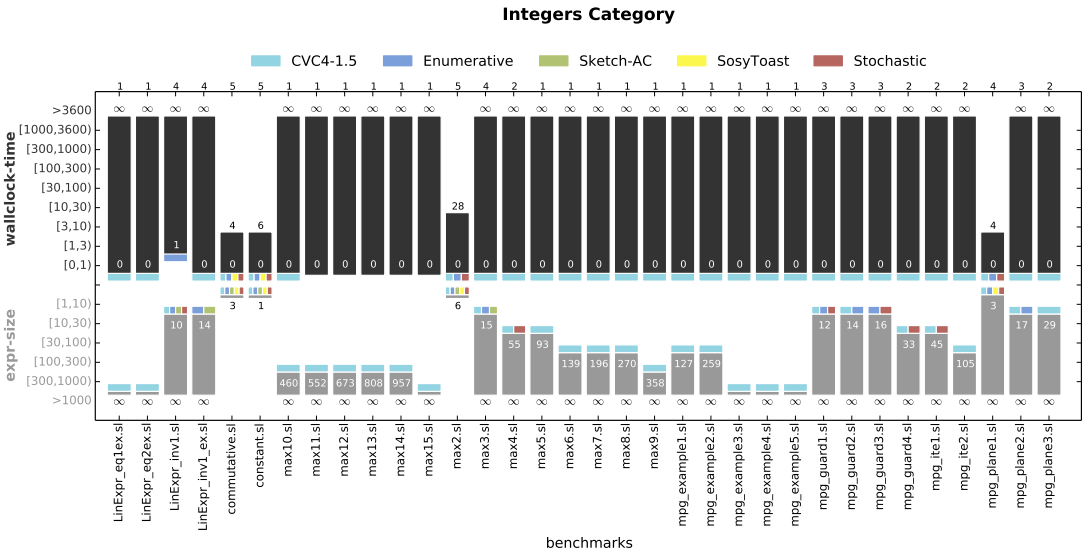

Figure 10: Evaluation of Integers benchmarks.

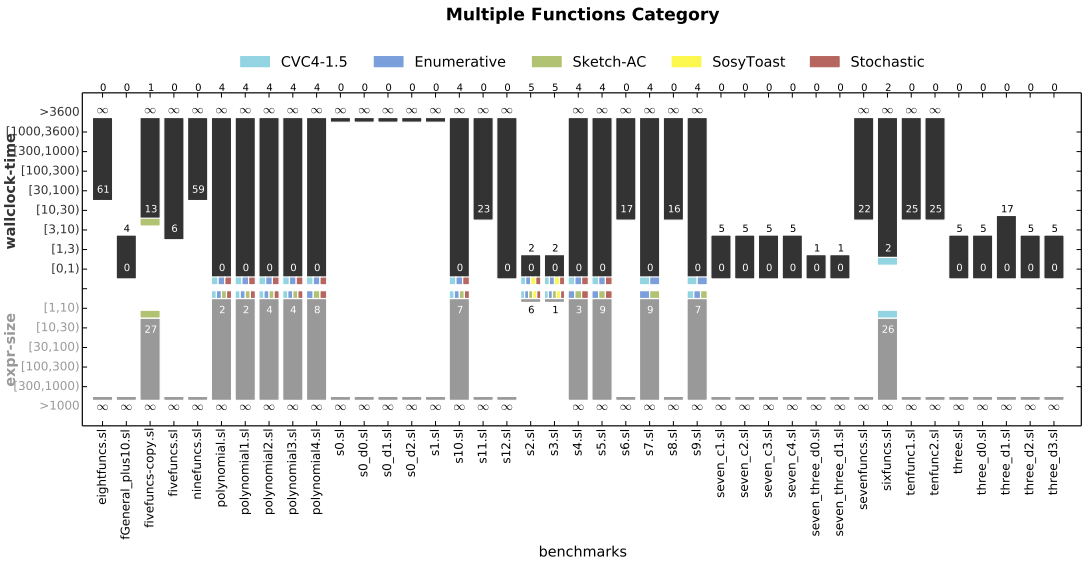

Figure 11: Evaluation of the Multiple Functions benchmarks.

Acknowledgments

We would like to thank the following people for various interesting discussions related to the competition, its tracks, the SyGuS format and various other topics related to syntax-guided synthesis: Sarah Chasins, Ruediger Ehlers, Pranav Garg, Viktor Kuncak, P. Madhusudan, Ken McMillan, Daniel Neider, Nissim Ofek, Arjun Radhakrishna, Mukund Raghothama, Andrew Reynolds, Heinz Riener, Shambwaditya Saha and Abhishek Udupa.

We would like to thanks the StarExec team, and especially Aaron Stump, for allowing us to use their platform and for their remarkable support for SyGuS-Comp's special needs.

This research was supported by US NSF grant CCF-1138996 (ExCAPE).

References

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2015): *Syntax-Guided Synthesis*. In: *Dependable Software Systems Engineering*, 40, IOS Press, pp. 1–25. doi:10.3233/978-1-61499-495-4-1.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-guided synthesis*. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8.

[3] Rajeev Alur, Pavol Cerný & Arjun Radhakrishna (2015): *Synthesis Through Unification*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 163–179. doi:10.1007/978-3-319-21668-3_10.

[4] Rajeev Alur, Pavol Cerný & Arjun Radhakrishna (2015): *Synthesis Through Unification*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 163–179. doi:10.1007/978-3-319-21668-3_10.

[5] David R. Cok (2013): *The SMT-LIBv2 Language and Tools: A Tutorial*.

[6] Pranav Garg, Christof Löding, P. Madhusudan & Daniel Neider (2014): *ICE: A Robust Framework for Learning Invariants*. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pp. 69–87. doi:10.1007/978-3-319-08867-9_5.

[7] Pranav Garg, Daniel Neider, P. Madhusudan, & Dan Roth (2015): *Learning Invariants using Decision Trees and Implication Counterexamples*. Technical report.

[8] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama & Jeffrey S. Foster (2015): *Adaptive Concretization for Parallel Program Synthesis*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 377–394. doi:10.1007/978-3-319-21668-3_22.

[9] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Oracle-guided Component-based Program Synthesis*. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 215–224. doi:10.1145/1806799.1806833.

[10] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang & Daniel Jackson (2015): *Alloy*: A General-Purpose Higher-Order Relational Constraint Solver*. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pp. 609–619. doi:10.1109/ICSE.2015.77.

[11] Daniel Neider, P. Madhusudan & Pranav Garg (2015): *ICE DT: Learning Invariants using Decision Trees and Implication Counterexamples*. Private Communication.

[12] Daniel Neider, Shambwaditya Saha & P. Madhusudan (2015): *Alchemist CS: An SMT-based synthesizer for Functions in Linear Integer Arithmetic*. Private Communication.

[13] Mukund Raghothaman & Abhishek Udupa (2014): *Language to Specify Syntax-Guided Synthesis Problems*. *CoRR* abs/1405.5590.

[14] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli & Clark W. Barrett (2015): *Counterexample-Guided Quantifier Instantiation for Synthesis in SMT*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 198–216. doi:10.1007/978-3-319-21668-3_12.

[15] Heinz Riener & Rudiger Ehlers (2015): *absTract sOlution Analyzing Synthesis Tool (System Description)*. Private Communication.

[16] Shambwaditya Saha, Pranav Garg & P. Madhusudan (2015): *Alchemist: Learning Guarded Affine Functions*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pp. 440–446. doi:10.1007/978-3-319-21690-4_26.

[17] Shambwaditya Saha, Daniel Neider & P. Madhusudan (2015): *Alchemist CS DT: Synthesizing Guarded Affine Functions using Constraint Solving and Decision-tree Learning*. Private Communication.

[18] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík & Kemal Ebcioglu (2005): *Programming by sketching for bit-streaming programs*. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pp. 281–294. doi:10.1145/1065010.1065045.

[19] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat & Sanjit Seshia (2006): *Combinatorial Sketching for Finite Programs*. In: *ASPLOS '06*, ACM Press, San Jose, CA, USA, pp. 404–415. doi:10.1145/1168857.1168907.

[20] Saurabh Srivastava, Sumit Gulwani & Jeffrey Foster (2010): *From program verification to program synthesis*. *POPL*. doi:10.1145/1706299.1706337.

[21] Aaron Stump, Geoff Sutcliffe & Cesare Tinelli (2014): *StarExec: A Cross-Community Infrastructure for Logic Solving*. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pp. 367–373. doi:10.1007/978-3-319-08587-6_28.

[22] Emina Torlak & Rastislav Bodík (2014): *A lightweight symbolic virtual machine for solver-aided host languages*. In: *PLDI*, p. 54. doi:10.1145/2594291.2594340.