On Complete Functional Synthesis

LARA-REPORT-2009-006

Philippe Suter Ruzica Piskac Mikaël Mayer Viktor Kuncak *

School of Computer and Communication Sciences (I&C) - École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

firstname.lastname@epfl.ch

Abstract

Synthesis of program fragments from specifications can make programs easier to write and easier to reason about. To integrate synthesis into programming languages, synthesis algorithms should behave in a predictable way—they should succeed for a well-defined class of specifications. They should also support unbounded data types such as numbers and data structures. We propose to generalize decision procedures into predictable and complete synthesis procedures. Such procedures are guaranteed to find code that satisfies the specification if such code exists. Moreover, we identify conditions under which synthesis will statically decide whether the solution is guaranteed to exist, and whether it is unique. We demonstrate our approach by extending decision procedures for integer linear arithmetic and data structures into synthesis procedures, and establishing results on the size and the efficiency of the synthesized code. We show that such procedures are useful as a language extension with implicit value definitions, and we show how to extend a compiler to support such definitions. Our constructs provide the benefits of synthesis to programmers, without requiring them to learn new concepts or give up a deterministic execution model.

1. Introduction

Synthesis of software from specifications [Manna and Waldinger 1980, 1971] promises to make programmers more productive. Despite substantial recent progress [Solar-Lezama et al. 2006, 2008; Srivastava et al. 2010; Vechev et al. 2009], 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 (and may be harder to debug than the code itself).

We therefore expect that practical applications of synthesis lie in its integration into the compilers of general-purpose programming languages. To make this integration feasible, we aim to identify well-defined classes of expressions and synthesis algorithms guaranteed to succeed for these classes of expressions. 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 necessary functionality, many decision procedure implementations additionally generate a satisfying assignment (a model) in case the given formula is satisfiable. Such model generation functionality has many uses, from better error reporting in verification, to test-case generation. This functionality could also be used as an advanced computation mechanism, which, given a set of values for some of the variables, finds the values of remaining variables such that a given constraint holds. Such a run-time mechanism is promising in supporting declarative programming style. However, it involves expensive and unpredictable search at run-time, and requires the deployment of a decision procedure as part of the run-time system. Our goal is to provide 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 that computes the desired values of variables at run-time. This approach can generate more efficient code that is specific to the constraint that needs to be solved at a given program point. Furthermore, 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 exists and is unique.

We demonstrate this approach by describing synthesis algorithms for domains of linear arithmetic and for collections of objects. We have found that, using these expressions 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 how these invariants are established.

In the area of integer arithmetic, we obtain a language extension that can implicitly define integer variables to satisfy given constraints. The applications of integer arithmetic synthesizer include conversions of quantities expressed in terms of multiple units, as well as a substantially more general notion of pattern matching on integers, going well beyond matching on constants or $(n + k)$-patterns of Haskell (`http://haskell.org`).

In the area of data structures, we describe a synthesis procedure that can compute sets of elements subject to constraints expressed in terms of basic set operations (union, intersection, set difference, subset, equality) as well as linear constraints on sizes of sets. We have found these constraints to be useful for implicitly defining sets of objects in algorithms, from simple operations such as choosing an element from a set and returning the rest, to picking fresh elements or splitting sets subject to given size constraints.

We have implemented these synthesis algorithms and deployed them as a compiler extension of the Scala programming language [Odersky et al. 2008].

Contributions. This paper makes the following contributions.

- We describe an approach for deploying algorithms for synthesis within programming languages. Our approach introduces a higher-order library function choose of type $(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha$, which takes as an argument a function F of type $\alpha \Rightarrow \mathsf{bool}$. Our compiler extension rewrites calls to choose into efficient code that finds a value x of type α such that $F(x)$ is true. Building on the choose primitive, we also show how to support substantially

* The author list has been sorted according to the inverse alphabetical order; this should not be used to determine the extent of authors' contributions. This research is supported in part by the Swiss National Science Foundation Grant #120433 "Precise and Scalable Analyses for Reliable Software".

more expressive pattern matching expressions in programming languages.

- We describe a methodology to convert decision procedures for a class of formulas into synthesis procedures that can rewrite the corresponding class of expressions into efficient executable code.
- We describe synthesis procedures for rational and integer linear arithmetic, as well as a logic of sets with size constraints. We show that, compared to invocations of constraint solvers at run-time, the synthesized code can have better worst-case complexity in the number of variables. This is because our synthesis procedure converts the given constraint (at compile time) into a solved form that can be executed while avoiding most of the search. The synthesized code is guaranteed to be correct by construction.
- We describe our experience of using synthesis as a plugin for the Scala compiler. Our implementation is publicly available.[1]

2. Example

We first illustrate the use of a synthesis procedure for integer linear arithmetic. Consider the following example to break down a given number of seconds (stored in the variable `totsec`) into hours, minutes, and leftover seconds.

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

Our synthesizer succeeds, because the constraint is in integer linear arithmetic. However, the synthesizer 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, s are not strict. After replacing `m <= 60` with `m < 60` and `s <= 60` with `s < 60`, the synthesizer emits no warnings. The generated code corresponds to the following:

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

The absence of a warning guarantees that the solution always exists and that it is unique. By writing the code in this style, the developer directly ensures that the condition `h * 3600 + m * 60 + s == totsec` will be satisfied, which eases the understanding of the program. Note that, if the developer imposes the constraint

```
val (hours, minutes, seconds) = 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 for too large parameter `totsec`.

In addition to the `choose` function, programmers can use synthesis for more flexible pattern matching on integers. In existing deterministic programming languages, matching on integers either tests on constant types, or, in the case of Haskell's $(n+k)$ patterns, on some very special forms of patterns. The following code illustrates the use of synthesis to describe a fast exponentiation function by doing case analysis on whether the argument is even or odd:

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

The correctness of the function follows from the observation that $\mathtt{fp}(m, b, i) = mb^i$, which we can prove by induction. Indeed, if we consider the case $2 * j + 1$, we observe:

$$\begin{array}{rcl} \mathtt{fp}(m,b,i) & = & \mathtt{fp}(m,b,2j+1) = \mathtt{fp}(mb,b^2,j) \\ \text{(by ind. hyp.)} & = & mb(b^2)^j = mb^{2j+1} = mb^i \end{array}$$

Note how the pattern matching on integer arithmetic expressions exposes the equations that make the inductive proof simpler. The pattern matching compiler generates the code that decomposes i into the appropriate new exponent j. Moreover, it checks that the pattern matching is exhaustive. The construct supports arbitrary expressions of linear integer arithmetic, and can prove e.g. that the set of patterns $2 * k$, $3 * k$, $6 * k - 1$, $6 * k + 1$ is exhaustive. The system also accepts implicit definitions, such as

```
val 42 ∗ x + 5 ∗ y = z
```

The system ensures that the above definition matches every integer z, and emits the code to compute x and y from z.

In addition to integer linear arithmetic, other decidable theories are amenable to synthesis and provide similar benefits. Consider the problem of splitting a set collection in a balanced way. The following code attempts to do that:

```
val (a1,a2) = choose((a1:Set[O],a2:Set[O]) ⇒
  a1 union a2 == s && a1 intersect a2 == empty &&
  a1.size == a2.size)
```

There are cases where the constraint above has no solution. It is possible to decide whether this is the case and generate an example value of a set s for which there is no solution (any set of odd size). If instead we weaken the requirement to:

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

the system can prove that the code has a solution for all possible input sets s. The nature of sets is such that there are typically many solutions for such constraints. Our synthesizer resolves these choices at compile time, which means that the generated code is deterministic.

Another example of synthesis is efficient support for more expressive algebraic data type patterns, including non-linear patterns. Such support reduces to the decision procedure for algebraic data types [Barrett et al. 2007; Oppen 1978; Suter et al. 2010].

3. From Decision- to Synthesis Procedures

We next define the notion of synthesis procedure and describe in general terms our approach for deploying predictable synthesis procedures based on decision procedures.

[1] `http://lara.epfl.ch/dokuwiki/comfusy`

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

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

Here $F(\vec{x}, \vec{a})$ is a formula in a decidable logic, which has variables $\vec{x}$ and parameters $\vec{a}$. The parameters $\vec{a}$ are program variables known at the time the statement is executed, whereas $\vec{x}$ are values that need to be computed so that $F(\vec{x}, \vec{a})$ holds.

We can translate the choose construct into the following sequence of commands in the guarded command languages [Dijkstra 1976]:

assert $(\exists \vec{x}. F(\vec{x}, \vec{a}))$;
havoc $(\vec{r})$;
assume$(F(\vec{r}, \vec{a}))$;

The simplicity of the translation of the choose construct also means that such construct is easier to use in verification systems such as [Barnett et al. 2004; Cohen et al. 2009; Flanagan et al. 2002; Zee et al. 2008, 2009] compared to the standard imperative code that would have the same effect.

Model-generating decision procedures. As a starting point for our synthesis algorithms we consider model-generating decision procedures. We assume that a decision procedure works on a class of first-order formulas Formulas defined in terms of terms Terms. The formulas can contain free variables, and we denote $\mathsf{FV}(F)$ the set of free variables in a formula F. By $F[x := e]$ we denote the result of replacing the free occurrences of x by e in F. 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). Given $F \in \mathsf{Formulas}$ we expect a model-generating decision procedure δ to produce either

a) a substitution $\sigma : \mathsf{FV}(F) \to C$ such that $F\sigma$ is a true, or

b) a special value unsat indicating that the formula is unsatisfiable.

We assume that the decision procedure is deterministic and behaves as a function δ. We write $\delta(F){=}\sigma$ or $\delta(F){=}\mathsf{unsat}$ to denote the result of applying the decision procedure δ to F.

Baseline: invoking a decision procedure at run-time. Just like an interpreter can be considered as a baseline implementation for a compiler, deploying a decision procedure at runtime can be considered as a baseline for our approach. In this scenario, we replace the invocation of (1) with

F = makeFormulaTree(makeVars($\vec{x}$), makeGroundTerms($\vec{a}$));
$\vec{r}$ = ($\delta(F)$ **match** {
  case $\sigma \Rightarrow (\sigma(x_1), \ldots, \sigma(x_n))$
  case unsat $\Rightarrow$ **throw new** Exception("No solution exists")
})

The dynamic invocation approach is flexible and useful. It can give some advantages of constraint logic programming [Jaffar and Maher 1994] and can also be done using e.g. the Z3 SMT solver [de Moura and Bjørner 2008] with quotations of the $F\#$ language [Syme et al. 2007]. However, there are important advantages of the compilation approach in terms of performance and predictability, as we discuss next.

Synthesis based on decision procedures. Our goal is to explore a compilation approach where a modified decision procedure is invoked at compile time, converting the formula $F(\vec{x}, \vec{a})$ into a solved form $\vec{x} = \vec{\Psi}(\vec{a})$ that implies the formula. More precisely, we have the following definitions.

Preliminaries. Let $\mathsf{FV}(q)$ denotes the set of free variables in a formula or 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, $\vec{x} = (x_1, \ldots, x_n)$ a vector of variables and $\vec{t} = (t_1, \ldots, t_n)$ a vector of terms, then $q[\vec{x} := \vec{t}]$ denotes the term resulting from subsstituting in q free variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, respectively. If we introduce q by writing $q(\vec{x})$ then we sometimes also denote $q[\vec{x} := \vec{t}]$ by $q(\vec{t})$. Below we only identify the output variables $\vec{x}$. Where needed, we write $\mathsf{FV}(F) \setminus \vec{x}_s$ to denote $\vec{a}_s$.

DEFINITION 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:

$$\exists \vec{x}. F \to \mathsf{pre}$$
$$\mathsf{pre} \to F[\vec{x} := \vec{\Psi}]$$

OBSERVATION 2. *The above definition implies that the the three formulas* $\exists \vec{x}. F$, pre, *and* $F[\vec{x} := \vec{\Psi}]$ *are all equivalent, because the third implication always holds:*

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

Consequently, if we can define a function $\mathsf{witn}(\vec{x}, F) = \vec{\Psi}$ *with* $\mathsf{FV}(\vec{\Psi}) \subseteq \mathsf{FV}(F) \setminus \vec{x}_s$ *such that* $\exists \vec{x}. F$ *is equivalent to* $F[\vec{x} := \vec{\Psi}]$, *then we can define*

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

The reason we use the translation that computes pre in addition to $\mathsf{witn}(\vec{x}, F)$ is that the synthesizer performs simplifications when generating pre, which can produce a formula faster to evaluate than $F[\vec{x} := \mathsf{witn}(\vec{x}, F)]$.

The synthesizer emits the terms $\vec{\Psi}$ in compiler intermediate representation and compiles them along with the rest of the code. We identify the syntax tree of $\vec{\Psi}$ with its meaning as a function from $\vec{a}$ to $\vec{x}$.

The overall compile-time processing of the choose statement (1) involves the following:

- emit a non-feasibility warning if the formula $\neg\mathsf{pre}$ is satisfiable, reporting the counterexample for which the synthesis problem has no solutions;
- 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;
- as the compiled code, emit the code that behaves as

  assert(pre)
  $\vec{r} = \vec{\Psi}$

  In practice it is often the case that the computation of $\vec{\Psi}$ already raises an exception in case pre does not hold, so there is no need for an explicit assert.

The existence of a model-generating decision procedure implies the existence of a trivial synthesis procedure (in the sense of Definition 1), which simply invokes the decision procedure at run-time.

The usefulness of the notion of synthesis procedure comes from the fact that we can use domain knowledge of the decision procedure to create compiled code that avoids this trivial solution. Among the potential advantages of the compilation approach are:

- improved run-time efficiency because part of the reasoning is done at compile-time;
- improved error reporting: the existence and uniqueness of solutions can be checked at compile time;
- simpler deployment: the emitted code can be compiled to any of the targets of the compiler, and requires no additional run-time support.

This paper therefore pursues the compilation approach. (As in processing of more standard programming language constructs, we do believe that there is space in the future for mixed approaches, such as just-in-time and profiling-guided synthesis.)

Efficiency of synthesis. We introduce the following measures to quantify the behavior of our synthesis procedure:

- time to synthesize the code, as a function of F;
- size of the synthesized code, as a function of F;
- running time of the synthesized code as a function of F and a measure of the run-time values of $\vec{a}$.

When using F as the argument of the above measures, we often consider not only the size of F, but also the dimension of the variable vector $\vec{x}$ and the parameter vector $\vec{a}$ in F.

From quantifier elimination to synthesis. The precondition pre can be viewed as a result of applying quantifier elimination (see e.g. [Nipkow 2008]) to remove $\vec{x}$ from F, with the following differences.

1. Synthesis procedures strengthen quantifier elimination procedures by identifying not only pre but also emitting the code $\vec{\Psi}$ that efficiently computes a witness for $\vec{x}$.
2. Quantifier elimination is typically applied to arbitrary quantified formulas of first-order logic and aims to successively eliminate all variables. Therefore, pre must be in the same language of formulas as F. This condition is not required in our case. Whatever the language of pre, it is still very useful for it to have *some* decision procedure, to enable accurate generation of compile-time warnings about the existence of solutions.
3. Worst-case bounds on quantifier elimination algorithms measure the size of the generated formula and the time needed to generate it, but not the size of $\vec{\Psi}$ or the time to evaluate $\vec{\Psi}$.

Despite the differences, we have found that we can naturally extend existing quantifier elimination procedures with explicit computation of witnesses that constitute the program $\vec{\Psi}$.

4. Selected Generic Techniques

We next describe some basic observations and techniques for synthesis that are independent of a particular theory.

4.1 Synthesis for Multiple Variables

Suppose we have function $\mathsf{witn}(x, F)$ that corresponds to constructive quantifier elimination step for one variable and produces a term Ψ such that $F[x := \Psi]$ holds iff $\exists x.F$ holds. We then lift $\mathsf{witn}(x, F)$ to synthesis for any number of variables, using the following translation scheme:

$$\begin{array}{l}
[\![(), F]\!] = (F, ()) \\
[\![(x_1, \ldots, x_n), F]\!] = \\
\quad \texttt{let}\ \ \Psi_n = \mathsf{witn}(x_n, F) \\
\qquad\quad \mathsf{pre}_n = \mathsf{simplify}(F[x_n := \Psi_n]) \\
\qquad\quad (\mathsf{pre}, (\Psi_1, \ldots, \Psi_{n-1})) = [\![(x_1, \ldots, x_{n-1}), \mathsf{pre}_n]\!] \\
\quad \texttt{in} \\
\quad\ (\mathsf{pre}, (\Psi_1, \ldots, \Psi_{n-1}, \Psi_n[x_1 := \Psi_1, \ldots, x_{n-1} := \Psi_{n-1}]))
\end{array}$$

Note that in practice we use local variable definitions instead of substitutions. Given (1), we generate, as $\vec{\Psi}$, a Scala code block

$$\left\{\begin{array}{ll}
\textbf{val}\ x_1 & = \Psi_1 \\
\ldots & \\
\textbf{val}\ x_{n-1} & = \Psi_{n-1} \\
\textbf{val}\ x_n & = \Psi_n \\
\vec{x} &
\end{array}\right\}$$

where the variables in Ψ_n directly refer to variables computed in $\Psi_1, \ldots, \Psi_{n-1}$ and where $\mathsf{FV}(\Psi_i) \subseteq \mathsf{FV}(F) \setminus \{x_i, \ldots, x_n\}$. A consequence of this recursive translation pattern is that the synthesized code computes values in the reverse order compared to the steps of a quantifier elimination procedure. This observation can be helpful in understanding the output of our synthesis procedures.

4.2 One-Point Rule Synthesis

If $x \notin \mathsf{FV}(t)$ we can define

$$\mathsf{witn}(x,\ x = t \land F) = t$$

If the formula does not have the form $x = t \land F$, we can often transform it into such form using theory-specific reasoning.

4.3 Output-Independent Preconditions

Note that if we can apply the following synthesis rule

$$\begin{array}{rl}
[\![\vec{x}, F_1 \land F_2]\!] = & \texttt{let}\ \ (\mathsf{pre}, \vec{\Psi}) = [\![\vec{x}, F_2]\!]\ \texttt{in} \\
& (\mathsf{pre} \land F_1, \vec{\Psi})
\end{array}$$

whenever $\mathsf{FV}(F_1) \cap \vec{x}_s = \emptyset$. We assume that this rule is applied whenever applicable and do not explicitly mention it in the sequal.

4.4 Propositional Connectives in First-Order Theories

Consider a quantifier-free formula in some first-order theory and suppose first that we wish to check formula satisfiability or apply quantifier elimination. We can then transform the formula to disjunctive normal form and process each disjunct independently. This allows us to focus on handling conjunctions of literals as opposed to arbitrary propositional combination.

We can similarly apply disjunctive normal form transformation to synthesis. Let $D_1, \ldots, D_n$ be the disjuncts in disjunctive normal form of a formula. We then apply synthesis to each D_i yielding a precondition pre_i and the solved form $\vec{\Psi}_i$. We generate code with conditionals that selects the first $\vec{\Psi}_i$ that applies:

$$\begin{array}{l}
[\![\vec{x}, D_1 \lor \ldots \lor D_n]\!] = \\
\quad \texttt{let}\ \ (\mathsf{pre}_1, \vec{\Psi}_1) = [\![\vec{x}, D_1]\!] \\
\qquad\quad \ldots \\
\qquad\quad (\mathsf{pre}_n, \vec{\Psi}_n) = [\![\vec{x}, D_n]\!] \\
\quad \texttt{in} \\
\quad \left(\bigvee_{i=1}^{n} \mathsf{pre}_i, \left\{ \begin{array}{ll}
\quad\ \ \textbf{if}\ (\mathsf{pre}_1) & \vec{\Psi}_1 \\
\textbf{else if}\ (\mathsf{pre}_2) & \vec{\Psi}_2 \\
\ldots & \\
\textbf{else if}\ (\mathsf{pre}_n) & \vec{\Psi}_n \\
\textbf{else} & \\
\quad \textbf{throw new}\ \text{Exception("No solution")} &
\end{array} \right\} \right)
\end{array}$$

While the disjunctive normal form can be exponentially larger than the original formula, the transformation to disjunctive normal form is used in practice [Pugh 1992] and has advantages in terms of the quality of synthesized code generated for individual disjuncts. What further justifies this approach is that we expect a small number of disjuncts in our specifications, and expect to need different synthesized values for variables in different disjuncts. Other methods can have better worst-case quantifier elimination complexity [Cooper 1972; Ferrante and Rackoff 1979; Nipkow 2008; Weispfenning 1997] and we also discuss their properties in the sequel, but disjunctive normal form is the method we currently use in our implementation.

4.5 Synthesis for Propositional Logic

Our paper focuses on synthesis for formulas over *unbounded* domains. However, to illustrate the potential asymptotic gain of precomputation in synthesis, consider the following simple approach when F is a propositional formula (see e.g. [Kukula and Shiple 2000] for a more sophisticated approach). Suppose that $\vec{x}$ are output variables and $\vec{a}$ are the remaining propositional variables (parameters).

Build an ordered binary decision diagram (OBDD) [Bryant 1986] for F, treating both $\vec{a}$ and $\vec{x}$ as variables for OBDD construction, and using a variable ordering that puts all parameters $\vec{a}$ before all output variables $\vec{x}$. Then split the OBDD graph at the point where all the decisions on $\vec{a}$ have been made. That is, consider the set of nodes that appear after all decisions on $\vec{a}$ have been made and no decisions on $\vec{x}$ have been made. For each of these OBDD nodes, we precompute whether this node reaches the `true` sink node. As the result of synthesis, emit the code that consists of nested if-then-else tests encoding the decisions on $\vec{a}$, followed by the code that, for each node that reaches `true` emits one path to the `true` node.

Although the size of the code can be singly exponential, the code executes in time linear in the total number of variables $\vec{a}$ and $\vec{x}$. This is in contrast to NP-hardness of finding a satisfying assignment for a propositional formula F, which would occur in the baseline approach of invoking a SAT solver at run-time. In summary, for propositional synthesis we can precompute solutions to an NP-hard problem and generate code that computes unknown propositional values in polynomial time.

In the next several sections, we describe synthesis procedures for several useful decidable logics over *infinite* domains (numbers and data structures) and discuss the efficiency improvements due to synthesis.

5. Synthesis for Linear Rational Arithmetic

We next consider synthesis for quantifier-free formulas of linear arithmetic over rationals. In this theory, variables range over rational numbers, terms are linear expressions $c_0 + c_1x_1 + \ldots + c_nx_n$, and the relations in the language are $<$ and $=$. Synthesis for this theory can be used to describe exact fractional arithmetic computations or prototype floating-point computations. It also serves as an introduction to the more complex problem of integer arithmetic synthesis.

Given a quantifier-free formula, we can efficiently transform it to negation-normal form. Furthermore, we observe that $\neg(t_1 < t_2)$ is equivalent to $(t_2 < t_1) \vee (t_1 = t_2)$ and that $\neg(t_1 = t_2)$ is equivalent to $(t_1 < t_2) \vee (t_2 < t_1)$. Therefore, there is no need to consider negations in the formula. We can also normalize the equalities to the form $t = 0$ and the inequalities to the form $0 < t$.

5.1 Solving Conjunctions of Literals

Given the observations in Section 4.4, we consider conjunctions of literals. The method follows Fourier-Motzkin elimination [Schrijver 1998]. Consider the elimination of a variable x.

Equalities. If x occurs in an equality constraint $t = 0$, then solve the constraint for x and rewrite it as $x = t'$ where t' does not contain x. Then apply one-point rule synthesis (Section 4.2). This step is Gaussian elimination, and we use it whenever it is applicable. We therefore eliminate first those variables that occur in some equalities and only then proceed to inequalities.

Inequalities. Next, suppose that x occurs only in strict inequalities $0 < t$. Depending on the sign of x in t, we can rewrite these inequalities into $a_p < x$ or $x < b_q$ for some terms a_p, b_p. Consider the more general case when there is both at least one lower bound a_p and at least one upper bound b_q. We can then define:

$$\mathsf{witn}(x, F) = (\max_p\{a_p\} + \min_q\{b_q\})/2$$

As one would expect from quantifier elimination, the pre corresponding to this case results from F by replacing the conjunction of all inequalities containing x with the conjunction

$$\bigwedge_{p,q} a_p < b_q$$

In case there are no lower bounds a_p, we define $\mathsf{witn}(x, F) = \min_q\{b_q\} - 1$; if there are no upper bounds b_q, we define $\mathsf{witn}(x, F) = \max_p\{a_p\} + 1$.

Complexity of synthesis for conjunctions. Consider a formula with N inequality literals, E equality literals, A input variables and V output variables (with $V \geq E$) whose values need to be synthesized.

The number of operations required to synthesize a program is bounded from above (modulo multiplication by a constant) by

$$\frac{2V(A+V)\cdot N^{2^V}}{2^{2^V-1}} + V(A+V)(E+N)$$

This bound is explained in details in appendix A.1.

The size of the generated program is bounded by:

$$O\left((A+V)\left(E + \frac{N^{2^{V+1}-1}}{2^{2^{V+1}-2}}\right)\right)$$

The generated program is a sequence of linear arithmetic operations; if we assume that the arithmetic operations take constant time, its execution time is proportional to program size.

Note that the algorithm has good efficiency in the absence of inequalities. In any case, it is polynomial when V is constant (e.g. synthesizing individual variable that satisfies a constraint).

5.2 Time-Efficient Code for Linear Rational Arithmetic

One way to lift synthesis for rational arithmetic from conjunctions of literals to arbitrary propositional combinations is to apply the disjunctive normal form method of Section 4.4. We then obtain complexity that is one exponential higher in formula size than the complexity of synthesis for conjunctions.

In the rest of this section we consider an alternative to disjunctive normal form. This alternative synthesizes code that can execute exponentially faster (even though it is not smaller) compared to the approach of Section 4.4.

The starting point of this method is quantifier elimination technique that avoids disjunctive normal form transformation, see e.g. [Ferrante and Rackoff 1979], [Nipkow 2008], [Bradley and Manna 2007, Section 7.3]. To remove a variable from negation normal form, this method finds relevant lower bounds a_p and upper bounds

b_q in the formula, then computes the values $m_{pq} = (a_p + b_q)/2$ and replaces a variable x_i with the values from the set $\{m_{pq}\}_{p,q}$ extended with "sufficiently small" and "sufficiently large" values [Nipkow 2008]. This quantifier elimination method gives us a way to compute pre.

To extend this method to synthesis (computation of $\mathsf{witn}(\vec{x}, F)$), we propose to do the following. Whenever applying a substitution that replaces x_i with m in quantifier elimination, attach a special substitution syntactic form $x_i \mapsto m$ as an additional auxiliary information to the literal. When using this process to eliminate one variable, the size of the formula can increase quadratically. After removing all variables, the size of the formula pre is bounded by $n^{2^{O(V)}}$. Note that, although it is doubly exponential in V, this quantity is not exponential in n. Build a decision tree that evaluates the values of all $n^{2^{O(V)}}$ literals in pre. On each complete path of this tree, we can statically determine whether the truth values of literals imply that pre is true; this is reduces to evaluating the truth value of a propositional formula in a given assignment to all variables. In the cases when the literals imply that pre holds, we use the attached substitution $x_i \mapsto m$ in true literals to recover the synthesized values of variables x_i. Such decision tree has depth $n^{2^{O(V)}}$ and we return it as the result of $\mathsf{witn}(\vec{x}, F)$. For a constant number of variables V, this tree represents a synthesized program whose running time is polynomial in n.

6. Synthesis for Linear Integer Arithmetic

We next describe our main algorithm, 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 $c_0 + c_1x_1 + \ldots + c_nx_n$, $n \geq 0$, c_i is an integer constant and x_i is an integer variable. Atoms are built using relations $\geq$, $=$ and $|$. The atom $c|t$ is interpreted as true iff an integer constant c divides term t. We also sometimes use $a < b$ as a shorthand for $a \leq b \wedge \neg(a = b)$. We describe a synthesis algorithm which works for conjunction of literals.

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 while adding a fresh variable, l. The obtained value of the fresh variable l is ignored in the final synthesized program:

$$\begin{array}{l} [\![\vec{x}, (c|t) \wedge F]\!] = \\ \texttt{let}\ \ (\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1})) = [\![(\vec{x}, q),\ t = cq \wedge F]\!] \\ \texttt{in}\ (\mathsf{pre}, \vec{\Psi}) \end{array}$$

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

$$\begin{array}{l} [\![\vec{x}, \neg(c|t) \wedge F]\!] = \\ \texttt{let}\ \ F' \equiv t + r = cq \wedge 1 \leq r \leq c - 1 \wedge F \\ \qquad (\mathsf{pre}, (\vec{\Psi}, \Psi_{n+1}, \Psi_{n+2})) = [\![(\vec{x}, q, r), F']\!] \\ \texttt{in}\ (\mathsf{pre}, \vec{\Psi}) \end{array}$$

In the rest of this section we consider a formula without negation or divisibility constraints.

6.1 Equality Constraints

Because equality constraints are suitable for deterministic elimination of variables, our procedure groups all equalities from a conjunction and solves them first. For this we use the $\texttt{eqSyn}$ algorithm described in Section 6.1.1. We can formalize this translation as a generalization of the scheme in Section 4.1 that solves for multiple variables and returnes a solution parameterized by a smaller number of variables. In the following, $\vec{y}$ are variables that are solved using equations and $\vec{z}$ are fresh variables introduced to represent the parameterized space of solutions for $\vec{y}$.

$$\begin{array}{l} [\![(\vec{y}, \vec{x}), E \wedge F]\!] = \\ \texttt{let}\ (\mathsf{pre}_Y, \vec{\Psi}_Y, \vec{z}) = \texttt{eqSyn}(\vec{y}, E) \\ \qquad F' = \mathsf{simplify}(F[\vec{y} := \vec{\Psi}_Y]) \\ \qquad (\mathsf{pre}, (\vec{\Psi}_Z, \vec{\Psi}_X)) = [\![(\vec{z}, \vec{x}), F]\!] \\ \qquad \mathsf{pre}_{Y0} = \mathsf{pre}_Y[\vec{x} := \vec{\Psi}_X, \vec{z} := \vec{\Psi}_Z] \\ \qquad \vec{\Psi}_{Y0} = \vec{\Psi}_Y[\vec{x} := \vec{\Psi}_X, \vec{z} := \vec{\Psi}_Z] \\ \texttt{in} \\ \ \ (\mathsf{pre}_{Y0} \wedge \mathsf{pre}, (\vec{\Psi}_{Y0}, \vec{\Psi}_X)) \end{array}$$

6.1.1 Reducing the Number of Output Variables

In this section we describe the algorithm $\texttt{eqSyn}$. Let $\Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0$ be an equality. We assume that the equality is already simplified in the sense that $\gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n) = 1$, where $\gcd$ stands for the greatest common divisor.

First we consider the case when there is only one output variable in the equality. In that case the algorithm $\texttt{eqSyn}$ returns:

$$\begin{array}{l} \texttt{eqSyn}(\Sigma_{i=1}^m \beta_i b_i + \gamma y = 0) = \\ \qquad\qquad (\gamma | - \Sigma_{i=1}^m \beta_i b_i,\ \ t = (-\Sigma_{i=1}^m \beta_i b_i)/\gamma, ()) \end{array}$$

From now on we assume that there is more than one output variable in the equality. Out 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 and following [Ginsburg and Spanier 1964, 1966] there is a semilinear set describing it . A *semilinear set* is finite union of linear sets. Given an integer vector $\vec{b}$ and a finite set of integer vectors S, a *linear set* is a set $\{\vec{x} \mid \vec{x} = \vec{b} + \vec{s}_1 + \ldots + \vec{s}_n; s_i \in S; n \geq 0\}$. Vector $\vec{b}$ is called a base vector while vectors in S are called step vectors. Every semilinear set is a solution of some Presburger arithmetic formula. Ginsburg and Spanier showed that converse holds as well: the set of all solutions of a Presburger arithmetic formula can be described with a semilinear set. However, we cannot apply this result immediately because there are also input variables whose values are not known until the execution time. We overcome this problem by introducing witnesses. We now explain in details three steps in defining a set describing set K.

Given the equality $\Sigma_{i=1}^m \beta_i b_i + \Sigma_{j=1}^n \gamma_j y_j = 0$ in the first step we define the set $S_H = \{\vec{y} \mid \Sigma_{j=1}^n \gamma_j y_j = 0\}$ which describes a solution set of a homogeneous equality. This is a linear set and it has a form $\{\vec{y} \mid \vec{y} = \alpha_1 \vec{s}_1 + \ldots + \alpha_k \vec{s}_k; \alpha_i \in \mathbb{Z}\}$. Vectors $\vec{s}_i$ are known and their effective computation is described in Section 6.1.2. What is important is that the number of s_i vectors is strictly smaller than n.

In the second step we compute a witness vector $\vec{w}$. For this we use generalization of Bézout's identity: for any numbers $k_1, \ldots, k_n$ with greatest common divisor d there exist integers $\alpha_1, \ldots, \alpha_n$ such that $\alpha_1 k_1 + \cdots + \alpha_n k_n = d$. A fast algorithm for computing those integers is described in Section 6.1.3.

Let $d = \gcd(\gamma_1, \ldots, \gamma_n)$ and let $I = \Sigma_{i=1}^m \beta_i b_i$. Note that this means that $d|I$ and this fact should be output as a required precondition. Let $J = I/d$. We apply Bézout's identity on numbers $\gamma_1, \ldots, \gamma_n$ and compute numbers $v_1, \ldots, v_n$ such that $d = v_1\gamma_1 + \cdots + v_n\gamma_n$. Multiplying this equality with J results in $d * J = v_1 * J * \gamma_1 + \cdots + v_n * J * \gamma_n$. We define $w_i = -v_i * J$ and form vector $\vec{w}$. It can easily be verified that vector $\vec{w}$ belongs to K.

In the last step we show that $K = S_H + \{\vec{w}\}$, i.e. $\vec{y} \in K \Leftrightarrow \vec{y} = \vec{y}_h + \vec{w} \wedge \vec{y}_h \in S_H$. If $\vec{y} \in K$, we need to show that $\vec{y} - \vec{w} \in S_H$. Let $z_i = y_i - w_i$. Applying few simple computation steps we show that $\Sigma_{j=1}^{n} \gamma_j z_j = 0$ and thus $\vec{z} \in S_H$. The other direction is analogous.

In summary, the algorithm `eqSyn` returns three pieces of information: the precondition $d | \Sigma_{i=1}^{m} \beta_i b_i$, the list of terms t_i, and the list of fresh variables λ_i. Using the computed values for generators of set S_H and a witness $\vec{w}$, terms t_i are computed as: $t_i = w_i + \lambda_1 s_{1i} + \ldots + \lambda_k s_{ki}$.

6.1.2 Efficient Computation of Linear Sets

To complete handling of equalities in our linear integer arithmetic synthesizer, the last hurdle we need to address is an efficient computation of a set describing the set of solutions of an equation $\Sigma_{i=1}^{n} \gamma_i y_i = 0$. Following the Omega test [Pugh 1992], we know the structure of this set. It is a linear set with $\vec{0}$ as the base vector and at most $n-1$ step vectors: $\{\alpha_1 \vec{s}_1 + \ldots + \alpha_{n-1} \vec{s}_{n-1} \mid \alpha_i \in \mathbb{Z}\}$. The Omega test is an algorithm which describes, among others, a computation of those step vectors. However, we find it too complex for our purposes, so here we propose direct computation of those step vectors without applying the Omega test.

Let $S = \{\vec{y} \mid \Sigma_{i=1}^{n} \gamma_i y_i = 0\}$. Note that S is always a non-empty set, since $\vec{0} \in S$. We will show that S is equal to the following set:

$$S_L = \left\{ \alpha_1 \begin{pmatrix} K_{11} \\ \vdots \\ K_{n1} \end{pmatrix} + \ldots + \alpha_{n-1} \begin{pmatrix} K_{1(n-1)} \\ \vdots \\ K_{n(n-1)} \end{pmatrix} \middle| \alpha_i \in \mathbb{Z} \right\}$$

where integer values K_{ij} are computed as follows:

- if $i < j$, $K_{ij} = 0$
- $K_{jj} = \frac{\gcd((\gamma_k)_{k \geq j+1})}{\gcd((\gamma_k)_{k \geq j})}$
- remaining values K_{ij} are computed as follows: for each index j, $1 \leq j \leq n-1$, consider the equation

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

  and find any solution. Let k_{ij} be a value of a variable u_{ij} in the found solution. For all the remaining K_{ij} for this fixed j, output $K_{ij} = k_{ij}$. In Section 6.1.3 we describe how to find a solution using only the Euclidean algorithm.

If one considers a matrix formed with coefficients K_{ij}, it is a lower triangular matrix. The reason for this is because vectors $\vec{s}_j$ are forming a basis for the set S and we compute them in a way that guarantees their mutual independence.

We next show the correctness of the construction by showing that $S = S_L$. First we show that each vector $\vec{s}_j$ belongs to S: $\vec{s}_j \in S \Leftrightarrow \Sigma_{i=1}^{n} \gamma_i K_{ij} = 0 \Leftrightarrow \gamma_j K_{jj} + \sum_{i=j+1}^{n} \gamma_i K_{ij} = 0$ which trivially holds by construction. Set S is a homogeneous set and therefore any linear combination of its elements is again an element in S.

To prove that the converse also holds, we show that a vector $\vec{x} \in S$ can be written as a linear combination of $\vec{s}_j$ vectors. Let $G_1 = \gcd((\gamma_k)_{k \geq 1})$: $\vec{x} \in S \Leftrightarrow \Sigma_{i=1}^{n} \gamma_i x_i = 0 \Leftrightarrow G_1 (\Sigma_{i=1}^{n} \beta_i x_i) = 0$, where $\beta_i = \gamma_i / G_1$. This implies that $\beta_1 x_1 + \Sigma_{i=2}^{n} \beta_i x_i = 0$ and all β_i values are coprime, ie. $\gcd((\beta_k)_{k \geq 1}) = 1$. Let $G_2 = \gcd((\beta_k)_{k \geq 2})$. We can then further rewrite the fact $\vec{x} \in S$ as: $\vec{x} \in S \Leftrightarrow \beta_1 x_1 + G_2 (\Sigma_{i=2}^{n} \beta_i^{'} x_i) = 0 \Leftrightarrow x_1 = -G_2 (\Sigma_{i=2}^{n} \beta_i^{'} x_i) / \beta_1$. Since β_1 and G_2 are coprime, it means that $\beta_1 | \Sigma_{i=2}^{n} \beta_i^{'} x_i$ and x_1 can be written as $x_1 = \alpha_1 G_2$ for the integer $\alpha_1 = -\Sigma_{i=2}^{n} \beta_i^{'} x_i / \beta_1$. Applying the definitions of G_2, β_i and G_1 results in $x_1 = \alpha_1 K_{11}$. Consider now a new vector $\vec{y} = \vec{x} - \alpha_1 \vec{s}_1$. Since $\vec{x}$ and $\vec{s}_1$ are elements of S, vector $\vec{y}$ is also an element of S. However, vector $\vec{y}$ has a special structure: its first component is 0. We repeat the described procedure on $\vec{y}$ and $\vec{s}_2$. This way we derive the value for an integer α_2 and a new vector $\vec{z}$ who has the first two components 0.

We continue with the described procedure until we obtain a vector $\vec{u}$ that has all components 0 except for the last two components. Since it is also an element of S, it holds $\gamma_{n-1} u_{n-1} + \gamma_n u_n = 0$. Using this, we conclude that $u_{n-1} \cdot \gcd(\gamma_{n-1}, \gamma_n) / \gamma_n$ is an integer. Our goal is to show that $\vec{u} = \alpha_{n-1} \vec{s}_{n-1}$, for some integer value α_{n-1}. Next we observe that vector $\vec{s}_{n-1}$ has a form $(0, \ldots, 0, \gamma_n / \gcd(\gamma_{n-1}, \gamma_n), -\gamma_{n-1} / \gcd(\gamma_{n-1}, \gamma_n))$. By defining α_{n-1} to be $\alpha_{n-1} = u_{n-1} \cdot \gcd(\gamma_{n-1}, \gamma_n) / \gamma_n$, it can easily be verified that $\vec{u} = \alpha_{n-1} \vec{s}_{n-1}$.

The entire procedure shows that every element of S can be represented as a linear combination of the $\vec{s}_j$ vectors and this finishes the proof of the correctness of the linear set construction.

6.1.3 Finding a Solution of an Equation

Finally, we describe a fast way of finding a solution for an equation $K + \Sigma_{i=1}^{n} \gamma_i u_i = 0$. This equation has an integer solution only if $\gcd((\gamma_k)_{k \geq 1}) | K$. For a purpose of constructing a linear set, this requirement holds in every equation for which we aim to find a solution. Therefore we are not addressing the case when the equation does not have a solution. The basis for the computation is again Bézout's identity: given integers a_1 and a_2 with greatest common divisor d there exist integers w_1 and w_2 such that $a_1 w_1 + a_2 w_2 = d$. The final solution of the equation will be constructed by using induction.

We start with a base case when there are only two variables: $K + \gamma_1 u_1 + \gamma_2 u_2 = 0$. Because $K / \gcd(\gamma_1, \gamma_2)$ is an integer, we introduce an integer $\alpha = K / \gcd(\gamma_1, \gamma_2)$. Following Bézout's identity there exist integers v_1 and v_2 such that $\gamma_1 v_1 + \gamma_2 v_2 = \gcd(\gamma_1, \gamma_2)$. We define $u_i = v_i \cdot (-\alpha)$ and verify that such computed u_1 and u_2 are correct solutions of the equation.

If there are more than two variables, we observe that $\Sigma_{i=2}^{n} \gamma_i u_i$ will be a multiple of $\gcd((\gamma_k)_{k \geq 2})$. We introduce the new variable u_N and find a solution of the equation $K + \gamma_1 u_1 + \gcd((\gamma_k)_{k \geq 2}) \cdot u_N = 0$ as described above. This way we obtain values of u_1 and u_N. To derive values of $u_2, \ldots, u_n$ we solve the equation $\Sigma_{i=2}^{n} \gamma_i u_i = \gcd((\gamma_k)_{k \geq 2}) \cdot u_N$. It satisfies the requirements to have a solution, has one variable less than the original equation and thus we can apply induction.

Another algorithm for finding a solution of an equation $K + \Sigma_{i=1}^{n} \gamma_i u_i = 0$ is presented in [Banerjee 1988]. It also runs in polynomial time and allows bounded inequality constraints as well. However, we chose the algorithm presented here because it of its simplicity. It can be easily implemented. Moreover, we are only interested in finding one solution of an equation. We have no additional constraints nor we are interested in a characterization of all solutions.

Here we did not describe an algorithm how to find integers w_1 and w_2 such that $a_1 w_1 + a_2 w_2 = \gcd(a_1, a_2)$, for given integers a_1 and a_2. It is a well-know standard algorithm, present in most of the textbooks on algorithms under the name Extended Euclidean algorithm, for example [Cormen et al. 2001][Figure 31.1].

6.1.4 Example

We demonstrate the process of eliminating equations on an example. Consider the translation

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

To eliminate an equation from the formula and to reduce a number of output variables, first we invoke `eqSyn`$(2a - b + 3x + 4y + 8z =$

0). It works in two phases. In the first phase, it computes the linear set describing a set of solutions of homogeneous equality $3x+4y+8z=0$. Using the algorithm described in Section 6.1.2, it returns:

$$S_L = \left\{ \alpha_1 \begin{pmatrix} 4 \\ -3 \\ 0 \end{pmatrix} + \alpha_2 \begin{pmatrix} 0 \\ 2 \\ -1 \end{pmatrix} \middle| \alpha_1, \alpha_2 \in \mathbb{Z} \right\}$$

The second phase computes a witness vector $\vec{w}$ and a precondition formula. Applying the procedure described in Section 6.1.1 results in vector $\vec{w} = (2a-b, b-2a, 0)$ and formula $1|2a-b$. Finally, we compute the output of eqSyn applied on $2a-b+3x+4y+8z=0$: it is a triple consisting of

1. a precondition $1|2a-b$
2. a list of terms denoting witnesses for (x,y,z):
$$\begin{array}{l} \Psi_1 = 2a - b + 4\alpha_1 \\ \Psi_2 = b - 2a - 3\alpha_1 + 2\alpha_2 \\ \Psi_3 = -\alpha_2 \end{array}$$
3. a list of fresh variables (α_1, α_2).

Next we replace each occurrence of x, y and z by the corresponding terms in the rest of the formula. This results in a new formula $7a - 3b + 13\alpha_1 \leq 4\alpha_2$. It has the same input variables, but the output variables are now α_1 and α_2. To find a solution for the initial problem, we let

$$(\mathsf{pre}_X, (\Psi_1, \Psi_2) = [\![(\alpha_1, \alpha_2), 7a - 3b + 13\alpha_1 \leq 4\alpha_2]\!]$$

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

$$(\mathsf{pre}_X, (2a - b + 4\Psi_1, b - 2a - 3\Psi_1 + 2\Psi_2, -\Psi_2))$$

6.2 Processing Inequality Constraints

From now on, 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 somehow similar to the case of rational arithmetic: we process variables one by one and then proceed further with the resulting formula.

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

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

As before, 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 take into an account that x has to be an integer. For this reason we define $a = \max_i \lceil A_i/\alpha_i \rceil$ and $b = \min_j \lfloor B_j/\beta_j \rfloor$. If b is defined, we define $x = b$, otherwise we set $x = a$.

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

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

Terms A_i and B_j may contain input and output variables and thus the obtained formula is not a linear arithmetic formula. In order to invoke our synthesizer on that formula, we have to convert it into an equivalent linear arithmetic formula. For this purpose we need to eliminate fractionals and floor and ceiling functions.

With lcm we denote the least common multiple. Let $L = \text{lcm}_{i,j}(\alpha_i, \beta_j)$. We introduce new terms $A'_i = \frac{L}{\alpha_i} A_i$ and $B'_j = \frac{L}{\beta_j} B_j$. Those terms are linear integer arithmetic terms and using them, we derive a new formula which is almost an integer linear arithmetic formula:

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

The obtained formula is an integer linear arithmetic formula and formula (2) is equivalent to

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

Still we cannot simply apply the synthesizer on that formula. Let $\{1, \ldots, J\}$ be a range of j indices. The newly derived formula contains J equations and $2 \cdot J$ new variables. The process of eliminating equalities as described in Section 6.1 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 notice that the value of k_j is always bounded: $k_j \in \{0, \ldots, L-1\}$. Thus, if the value of k_j would be known, we would have a formula with only J new variables and J additional equations. The equations elimination described before would then result with 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 solution. As soon as a solution is found, we stop and proceed with the obtained values of output variables. If no solution is found, we raise an exception, because the original formula has no integer solution.

We finish the description of the synthesizer with an example which illustrated the above algorithm.

Example Consider a 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 output variables: y, k and l. The value of k is bounded: $0 \leq k \leq 5$. We start with $k = 0$: this leads to a formula $4y + a + b = 2l \wedge 0 \leq 8y + 5a + 5b$, with a and b as input variables and l and y as output variables. Invoking the synthesizer on this code results in the precondition formula $2|a+b$ and the code:

```
val alpha = ((−5 ∗ a − 5 ∗ b)/8).ceiling
val l = (a + b)/2 + 2 ∗ alpha
val y = alpha
```

Because a and b are input variables, the validity of the precondition formula can be checked. If it is valid, we stop further executions of the algorithm and output the above code followed by the code computing the value of x. If the precondition formula is not valid, we repeat the procedure for the remaining values of k: $k = 1, \ldots, 5$. If none of those values returns the satisfying solution, we throw an exception.

6.3 Disjunctions in Presburger Arithmetic

We can again lift synthesis for conjunctions to synthesis for arbitrary propositional combinations is to apply the method of

Section 4.4. We also obtain complexity that is one exponential higher than the complexity of synthesis from previous section. Approaches that avoid disjunctive normal form can be used in this case as well [Ferrante and Rackoff 1979; Nipkow 2008; Weispfenning 1997], and we expect the lower and upper bounds on quantifier elimination [Weispfenning 1997] to apply to the size of the synthesized code.

6.4 Optimizations used in the Implementation

In this section we describe some optimizations and heuristics that we utilize in implementation. Using some of them we obtained a speedup by several orders of magnitude.

Merging inequalities. Whenever two inequalities $t_1 \leq t_2$ and $t_2 \leq t_1$ appear in a conjunction, we substitute them with equality $t_1 = t_2$. This makes the process of variable elimination more efficient.

Heuristic for choosing the right equality for elimination. When there are several equalities in a formula, we chose to eliminate an equality for which the least common multiple of all the coefficients is the smallest. We observed that this reduces the number of integers to iterate over.

Some optimizations on modulo operations. In processing inequalities, as described in Section 6.2, as soon as we introduce the $\mod$ operator, we are immediately aware of potential longer processing time. It is because finding the suitable value of the reminder in equation $B'_j \mod L \leq B'_j - A'_i$, requires invoking a loop. While searching for a witness, we might need to check for all possible L values. Therefore, we try not to introduce the $\mod$ operator in the first place. This is possible in few cases. One of them is when either $\alpha_i = 1$ or $b_j = 1$. In that case, if for example $\alpha_i = 1$, an equivalent integer arithmetic formula is easily derived:

$\lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \Leftrightarrow A_i \leq \lfloor B_j/\beta_j \rfloor \Leftrightarrow \beta_j A_i \leq B_j$

Another example for when we do not introduce the $\mod$ operator is the case when $A'_i - B'_j$ evaluates to a number N, such that $N > L$. In that case, it is clear that $B'_j \mod L \leq B'_j - A'_i$ is a valid formula and thus the returned formula is $\top$.

Finally, we describe an optimization that leads to reducing a number of a loop executions. This optimization is possible when there exists an integer N such that $B'_j = N \cdot T_j$ and $L = N \cdot L_1$. (Unless $L = \beta_j$, this is almost always the case). In the case that N exists, then k_j also has to be a multiple of N. Putting together all that, an equivalent formula of $B'_j \mod L \leq B'_j - A'_i$ is formula $T_j \mod L_1 = k_j \wedge N \cdot k_k \leq B'_j - A'_i$. This reduces the number of loop iterations for at least a factor N.

6.5 Complexity

We next describe the complexity of our algorithms, for both the synthesis process itself and the synthesized programs.

A conversion of the formula to Disjunctive Normal Form might increase by an exponential factor both the running time and the space of our synthesizer and also the size of the generated program (see 6.5). The execution time would also be multiplied by an exponential factor as we are checking the conditions in sequence.

In the sequel we analyze a conjunction of atomic equations.

Synthesizer Time Complexity The number of times $\Omega(E, N, V)$ given the number of equalities E, inequalities N and output variables V, is bounded from above by:

$$\Omega(E, N, V) = O\left(2 + \frac{N^{2^V}}{2^{2^{V+1}-1}} + \min(V, E)\right)$$

This result is proved in appendix A.2.

Note that, the algorithm has again good efficiency in the absence of inequalities. In any case, it is also polynomial when V is constant.

Generated Programs Size Each recursive call to remove an equality also means at least an assignment, so there can be at least doubly exponential assignments.

Generated programs Time Complexity Without inequalities, the complexity is linear in the number of equations. Else, it can also be doubly exponential.

6.6 Generalization to Parametrized Presburger Arithmetic

It is possible generalize our synthesizer in the case when the coefficients of the output variables are not only integers anymore, but they can be any arithmetic expression over the input variables. This extension allows us to write implicit programs like this one:

```
val (x, y) = choose((x: Int, y: Int) ⇒
    x ∗ (k³+1) + y ∗ (2k²−k) == k^4 &&
    x ∗ k > 3 ∗ k²+5
)
```

In that case, all the choices made during synthesis depending on the sign of the coefficients have to be done at run-time. Each choice on the sign generates two or more different solutions, so locally multiplies by two or three the execution time and the size of the generated program.

The coefficients of the Bezout function in this case become known at run-time only, so we have to integrate the Bezout function into the code as a library function. The situation is the same for the `gcd` function.

Furthermore, the running time of the programs is not constant anymore, it depends on the value of the inputs. For example, the upper bounds of the generated for loops in Section 6.2 might now be arithmetic expressions.

$$
\begin{array}{lcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid T_1 = T_2 \mid T_1 < T_2 \mid (K|T) \\
B & ::= & x \mid \emptyset \mid \mathcal{U} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
T & ::= & k \mid K \mid T_1 + T_2 \mid K \cdot T \mid \ |B| \\
K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 1. A Logic of Sets and Size Constraints

7. Synthesis for Sets with Size Constraints

In this section we define a logic of sets with cardinality constraints and describe a synthesis procedure for it. Our logic supports the standard set operators union, intersection and complement, and the subset and equality relations. In addition, it supports the size operator on sets, as well as integer linear arithmetic constraints over these sizes. Its syntax is given in Figure 1. This logic was considered in a number of applications [Feferman and Vaught 1959; Kuncak et al. 2006; Zarba 2004, 2005].

As in the previous sections, we consider the problem (1)

$$\vec{r} = \mathsf{choose}(\vec{x} \Rightarrow F(\vec{x}, \vec{a}))$$

where the components of vectors $\vec{a}, \vec{x}, \vec{r}$ are either set or integer variables.

Figure 2 describes a synthesis procedure that returns a precondition predicate $\mathsf{pre}(\vec{a})$ and a solved form Ψ. The procedure is based on the quantifier elimination algorithm presented in [Kuncak et al. 2006] which reduces a formula in our logic to an equisatisfiable Presburger arithmetic formula. The algorithm eliminates set variables in two phases. In the first phase all set expressions are rewritten as disjunctive unions of corresponding Venn regions. The second phase introduces for the cardinality of each Venn region a fresh integer variable, and thus reduces the whole formula to a Presburger arithmetic formula. The input variables in this Presburger arithmetic formula are the integer input variables from the original formula and fresh integer variables denoting cardinalities of Venn regions of the input set variables. Note that all values of all those input variables is known from the program. The output variables are the original integer output variables and freshly introduced integer variables denoting cardinalities of Venn regions that are contained in the output set variables. We adapt this algorithm and conjoin it with the synthesizer for Presburger arithmetic described in Section 6. The synthesizer outputs the precondition predicate pre and emits the code for computing values of the new output variables. Based on those returned integer values we reconstruct a model for the original formula and finally we emit the code that computes values of the original output set variables. Notice that the precondition predicate pre will be a Presburger arithmetic formula with the terms built using the original integer input variables and the cardinalities of Venn regions of the original input set variables. As an example, if i is an integer input variable and a and b are set input variables then the precondition predicate might be the following formula $\mathsf{pre}(i, a, b) = |a \cap b| < i \wedge |a| \leq |b|$.

In the last step of the algorithm, while outputting code, we use the commands `fresh` and `take`. The command `take` takes as arguments an integer k and a set S, and returns a subset of S of the size k. The command $\texttt{fresh}(k)$ is invoked when k fresh elements need to be generated. Those commands are used only in the code that will compute output values of set variables, because the linear integer arithmetic synthesizer produces code for computation of integer output variables. The set output variables are computed one by one. Given an output set variable Y_i, the code that effectively computes the value of Y_i is emitted in several steps. With S_i we denote a set containing set variables occurring in the original formula whose values are already known. Initially S_i contains only the input set variables. Our goal is to describe the construction of Y_i in terms of sets that are already in S_i. We start by computing the Venn regions for Y_i and all the sets in S_i in order to define Y_i as a union of those Venn regions. Therefore we are interested only in those Venn regions that are subset of Y_i. Let T_j be one such a Venn region. It can be represented as $T_j = Y_i \cap U_j$ where U_j has a form $U_j = \cap_{S \in S_i} S^{(c)}$ and $S^{(c)}$ denotes either S or S^c. On the other hand, T_j can also be represented as a disjoint union of the original R_u Venn regions. Those R_u are Venn regions that were constructed in the beginning of the algorithm for all input and output set variables. As the linear integer arithmetic synthesizer outputs the code that computes values h_u, where $h_u = |R_u|$, we can effectively compute the size of each T_j. If $T_j = R_{u_1} \cup \ldots \cup R_{u_k}$ then the size of T_j is $|T_j| = d_j = \sum_{l=1}^{k} h_{u_l}$. Note that d_j is easily computed from the linear integer arithmetic synthesizer and based on the value of d_j we define a set K_j as $K_j = \texttt{take}(d_j, U_j)$. Finally, we emit the code that defines Y_i as a finite union of K_j's: $Y_i = \cup_j K_j$.

Based of the values of d_j, we can introduce further simplifications. If $d_j = 0$, none of elements of U_j contributes to Y_i and thus $K_j = \emptyset$. On the other hand, if $d_j = |U_j|$, applying a simple rule $S = \texttt{take}(|S|, S)$ results in $K_j = U_j$. A special case is when $U_j = \cap_{S \in S_i} S^c$. If in this case also holds that $d_j > 0$, we need to take d_j elements that are not contained in any of already known sets, i.e. we need to generate fresh d_j elements. For this purpose we invoke the command `fresh`.

Example run of the algorithm Consider the choose statement

```
val s1 = choose((s: Set) ⇒ a subsetOf s && s.size ≤ b.size)
```

INPUT: a formula $F(\vec{X}, \vec{Y}, \vec{k}, \vec{l})$ in the logic defined in Figure 1, input variables $X_1, \ldots, X_n, k_1, \ldots, k_m$ and output variables $Y_1, \ldots, Y_s, l_1, \ldots, l_t$, where X_i and Y_j are set variables, k_i and l_j are integer variables

OUTPUT: code that computes values for the output variables from the input variables

1. Apply the first steps towards a Presburger arithmetic formula:
   (a) Replace each atom $S_1 = S_2$ with $S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$
   (b) Replace each atom $S_1 \subseteq S_2$ with $|S_1 \cap S_2^c| = 0$
2. Introduce the Venn regions of sets X_i's and Y_j's: let u be a binary word of the length $n+m$. The set variable R_u represents a Venn region where each '1' stands for a set and '0' stands for a complement. To illustrate, if $n = 2$, $m = 1$ and $u = 001$, then $R_{001} = X_1^c \cap X_2^c \cap Y_1$. Rewrite each set expression as a disjoint union of corresponding Venn regions.
3. Create a Presburger arithmetic formula: an integer variable h_u denotes the cardinality of a Venn region R_u. Use the fact that $|S_1 \cup S_2| = |S_1| + |S_2|$ iff S_1 and S_2 are disjoint to rewrite the whole formula as the Presburger arithmetic formula. The resulting formula we denote with $F_1(\vec{h_u}, \vec{k}, \vec{l})$.
4. Create a Presburger arithmetic formula which corresponds to quantifier elimination: let v be a binary word of length n. A set variable P_v denotes a Venn region of input set variables, which means that $|P_v|$ is a known value. Create a formula that expresses each $|P_v|$ as a sum of corresponding h_u's. Define the formula $F_2(\vec{h_u}, |\vec{P_v}|)$ as the conjunction of all those formulas.
5. Create code that computes values of output vectors. First invoke the linear arithmetic synthesizer described in Section 6 to generate the code corresponding to:

   val $(\vec{h_{un}}, \vec{l_n}) = \mathsf{choose}((\vec{h_u}, \vec{l}) \Rightarrow F_1(\vec{h_u}, \vec{k}, \vec{l}) \wedge F_2(\vec{h_u}, |\vec{P_v}|))$

   Invoking the synthesizer returns code that computes expressions for the integer output variables $\vec{l_n}$ and for the variables $\vec{h_u}$. For each set output variable Y_i, do the following: let S_i be a set containing already known or defined set variables, let T_j be a Venn region of $S_i \cup Y_i$ that is contained in Y_i. Now, for each T_j do: take all R_u that belong to T_j and let d_j be a sum of all corresponding h_{un}. Let $U_j = T_j \backslash Y_i$. Based on the value of d_j output the following code:

   - if $U_j = \cap_{S \in S_i} S^c$ and $d_j > 0$, output the assignment $K_j = \texttt{fresh}(d_j)$
   - if $d_j = 0$, output the assignment $K_j = \emptyset$
   - if $d_j = |U_j|$, output the assignment $K_j = U_j$
   - otherwise output the assignment $K_j = \texttt{take}(d_j, U_j)$

   Finally, construct Y_i as a union of all K_j sets: $Y_i = \cup_j K_j$

Figure 2. Algorithm for synthesizing a function Ψ such that $F[\vec{x} := \Psi(\vec{a})]$ holds, where F has the syntax of Figure 1

We apply the algorithm from Figure 2. After completing the third step, we obtain the formula

$$F_1(\vec{h}_u) \equiv h_{100} = 0 \wedge h_{110} = 0 \wedge h_{101} \leq h_{011} + h_{010}$$

We simplify the formula obtained in the fourth step using the constraints from the third step and obtain the formula

$$F_2(\vec{h}_u) \equiv h_{111} = |a \cap b| \wedge h_{101} = |a \cap b^c| \wedge h_{011} + h_{010} = |a^c \cap b|$$

We call the linear arithmetic synthesizer and the following values for h_u variables

$$h_{100} = 0^*, h_{110} = 0^*, h_{111} = |a \cap b|^*, h_{101} = |a \cap b^c|^*,$$
$$h_{010} = |a \cap b^c|, h_{011} = |a^c \cap b| - |a \cap b^c|, h_{001} = 0,$$
$$h_{000} = |a^c \cap b^c|^*$$

where * denotes the deterministic values of variables. The linear arithmetic synthesizer also outputs the precondition predicate pre: $\mathsf{pre}(a, b) \equiv |a^c \cap b| \geq |a \cap b^c|$. Finally, we emit the following code, written in the Scala-like syntax:

```
val k1 = a -- b
val k2 = a ** b
val k3 = take((b -- a).size - (a -- b).size, b -- a)
val S = k1 ++ k2 ++ k3
```

Here `x ++ y`, `x ** y` and `x -- y` denote $x \cup y$, $x \cap y$ and $x \cap y^c$ respectively, and `x.size` the cardinality of x.

Partitioning a set Consider the following invocation of the `choose` function that generalizes the example in Section 2.

```
val (setA, setB) = choose((a: Set[String], b: Set[String]) ⇒
    (−maxDiff ≤ a.size − b.size && a.size − b.size ≤ maxDiff
        && a ++ b == bigSet && a ** b == Set.empty
))
```

This example combines integer and set variables. Given a set `bigSet`, the goal is to divide it into two partition. The previously defined integer variable `maxDiff` specifies the maximum amount by which the sizes of the two partitions may differ. Our synthesizer successfully generates the code for this example which computes acceptable sizes for the Venn regions using the appropriate integer arithmetic expressions, selects elements into these Venn regions, and computes the sets a and b by taking the union of non-empty Venn regions in which these sets participate.

8. Implementation

We have implemented our synthesis procedures as a Scala compiler extension (please consult the non-anonymous appendix for the implementation URL). We chose Scala because it supports higher-order functions that make the concept of a choose function natural, and extensible pattern matching in the form of extractors [Emir et al. 2007]. Besides, the compiler supports plugins that can serve as additional phases in the compilation process.[2] We used an off-the-shelf decision procedure [de Moura and Bjørner 2008] to handle the compile-time checks.

Our plugin supports the synthesis of integer values through the `choose` function constrained by linear arithmetic predicates, as well as the synthesis of set values constrained by predicates of the logic described in Section 7. Additionally, it can synthesize code for pattern-matching expressions on integers such as the ones presented in Section 2.

Figure 3 shows the compile times for a set of benchmarks, with and without our plugin (in the latter case, the generated code is of course of no use). The examples *SecondsToTime*, *FastExponentiation* were presented in Section 2 and *SplitBalanced* in Section 7.

[2] `http://www.scala-lang.org/node/140`

	scalac	w/ plugin	w/ checks
SecondsToTime	3.05	3.2	3.25
FastExponentiation	3.1	3.15	3.25
ScaleWeights	3.1	3.4	3.5
PrimeHeuristic	3.1	3.1	3.1
SetConstraints	3.1	3.5	–
SplitBalanced	3.2	5.3	–
All	5.25	6.35	6.5

Figure 3. Measurement of compile times: without applying synthesis (scalac), with synthesis but with no call to Z3 (w/ plugin) and with both synthesis and compile-time checks activated (w/ checks). All times are in seconds. There are no compile-time checks for the synthesis of set values.

ScaleWeights computes solutions to a puzzle, *PrimeHeuristic* contains a long pattern-matching expression where every pattern is checked for reachability, and *SetConstraints* is a variant of *SplitBalanced*. We also measured the times with all benchmarks placed in a single file, as an attempt to balance out the time taken by the Scala compiler to start up. Our numbers show that the additional time required for the code synthesis is minimal. One should also note that the code we tested contained almost exclusively calls to the synthesizer, which is clearly not representative of what we expect will be the common practice of using a selective number of invocations.

9. Related Work

Our work differs from the past ones in 1) using decision procedures to guarantee the computation of synthesized functions whenever a synthesized function exists, 2) bounds on the running times of the synthesis algorithm and the synthesis code size and running time, and 3) deployment of synthesis in well-delimited pieces of code of a general-purpose programming language.

Early work on synthesis [Manna and Waldinger 1980, 1971] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle. Consequently, while it can synthesize interesting programs containing recursion, it cannot provide completeness and termination guarantees as synthesis based on decision procedures.

Recent work on synthesis [Srivastava et al. 2010] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. Furthermore, the work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures. This work is more ambitious and aims to synthesize entire algorithms. By nature, it cannot be both terminating and complete over the space of all programs that satisfy an input/output specification (thus the approach of specifying program resource bounds). In contrast, we provide completeness guarantees for a given specification, but focus on synthesis of program fragments with very specific control structure dictated by the nature of the decidable logical fragment.

Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [Solar-Lezama et al. 2006, 2007, 2008]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. In contrast, our synthesis uses the mathematical structure of a decidable theory to explore space of all functions that satisfy the specification. This enables our approaches to achieve completeness without putting any a priori bound on the syntax tree size. Indeed, some of the algorithms we describe can generate fairly large and efficient programs. We expect that our techniques could be fruitfully integrated into sketching frameworks.

Synthesis of reactive systems generates programs that run forever and interact with the environment. However, known complete algorithms for reactive synthesis work with finite-state systems [Pnueli and Rosner 1989] or timed systems [Asarin et al. 1995]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [Vechev et al. 2009]. These techniques usually take specifications in a fragment of temporal logic [Piterman et al. 2006] and have resulted in tools that can synthesize useful hardware components [Jobstmann and Bloem 2006; Jobstmann et al. 2007]. Our work examines non-reactive programs, but supports infinite data without any approximation, and incorporates the algorithms into a compiler for a general-purpose programming language.

Automata-based decision procedures, such as those implemented in the MONA tool [Klarlund and Møller 2001] could be used to synthesize efficient (even if large) code from expressive specifications. The work on graph types [Klarlund and Schwartzbach 1993] proposes to synthesize fields given by definitions in monadic second-order logic. The subsequent work [Møller and Schwartzbach 2001] has focused on verification as opposed to synthesis.

Our approach can be viewed as sharing some of the goals of partial evaluation [Jones et al. 1993]. However, we do not need to employ general-purpose partial evaluation techniques (which typically provide linear speedup), because we have the knowledge of a particular decision procedure. We use this knowledge to devise a synthesis algorithm that, given formula F, generates the code corresponding to the invocation of this particular decision procedure. This synthesis process checks the uniqueness and the existence of the solutions, emitting appropriate warnings. Moreover, the synthesized code can have reduced complexity compared to invoking the decision procedure at run time, especially when the number of variables to synthesize is bounded.

A. Derivation of Complexities

This part contains proof complements about the complexities of our synthesis algorithms.

A.1 Linear Rational complexity

We assume A input variables (containing the constant coefficient), V output variables, E equalities ($E \leq V$), and N inequalities.

We want the number of arithmetic operations during synthesis, which we write $\Omega(A, V, E, N)$.

We will prove that:

$$\Omega(A, V, E, N) \leq U(A, V, E, N)$$

where

$$U(A, V, E, N) = K_5 \cdot \left(2V(A+V) \sum_{k=2}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} + f(A, V, E, N) \right)$$

where

$$f(A, V, E, N) = V \cdot (A+V)(E+N)$$

After bounding from above the sum, we get the expected result:

$$\Omega(A, V, E, N) = O\left(\frac{2V(A+V) \cdot N^{2^V}}{2^{2^V-1}} + V(A+V)(E+N) \right)$$

A.1.1 Removing 1 equality

We take a variable x_V , and we solve one of its equations $x_V = t$. This takes $O(A+V-1)$ operations.

Then, for each other $(E-1+N)$ equations, we replace x_V by its expression, this takes $O(A+V-1)$ per equation, so total replacement takes $O((E-1+N) \cdot (A+V))$ operations.

Therefore, we have the following relation:

$$\Omega(A, V, E, N) = \Omega(A, V-1, E-1, N) + O((E-1+N)(A+V-1))$$

A.1.2 Removing E equalities

By summing up the terms while decreasing the number of equalties and variables, we obtain:

$$\begin{array}{rcl} \Omega(A, V, E, N) & = & \Omega(A, V-E, 0, N) \\ & + & O\left(\sum_{i=1}^{E}(E-i+N)(A+V-i)\right) \end{array}$$

Let us simplify the inner term:

$$\begin{array}{cll} & \sum_{i=1}^{E}(E-i+N)(A+V-i) & \\ = & (E+N)(A+V)\sum_{i=1}^{E} 1 & -\ (A+V+E+N)\sum_{i=1}^{E} i \\ & & +\sum_{i=1}^{E} i^2 \\ = & (E+N)(A+V)E & -\ (A+V+E+N)\frac{E(E+1)}{2} \\ & & +\frac{E(E+1)(2E+1)}{6} \\ \leq & \frac{E}{6}(6(E+N)(A+V) & -3(A+V+E+N)(E+1)) \\ & & +(E+1)(2E+1)) \\ & \ldots & \\ \leq & \frac{E}{6}\left((A+V)(3E+6I) - 3IE - E^2 - 3IE + 1\right) & \\ \leq & \frac{E}{6}\left((A+V)(6E+6I)\right) & \\ \leq & E \cdot (A+V)(E+N) & \end{array}$$

Therefore, we have the following relation:

$$\begin{array}{rcl} \Omega(A, V, E, N) & = & \Omega(A, V-E, 0, N) \\ & + & O\left(E \cdot (A+V)(E+N)\right) \end{array}$$

A.1.3 Removing V variable when $E = 0$, $N = 0$

Without equations nor inequations, we assign 0 to all remaining variables.

$$\Omega(A, V, 0, 0) = O(V)$$

A.1.4 Removing 1 variable when $E = 0$, $N = 1$

With only one inequation, we treat it as an equality $+1$, solve it and then assign 0 to all remaining variables.

Complexity :

$$\Omega(A, V, 0, 1) = O(A) + O(V)$$

A.1.5 Removing 1 variable when $E = 0$, $N \geq 2$

Once all equalities are removed ($E = 0$), what is the complexity of removing one variable if there are at least two inequalities ?

First, we take a variable, split the inequations between L inequations on the left, R on the right, and U nothing. Assuming the worst-case complexity, $U = 0$, and $L + R = N$

The split operation is done in $O((A+V)(L+R))$ operations, so $O((A+V) \cdot N)$ operations.

The expression $(\max(\ldots) + \min(\ldots))/2$ of section 5.1 is constructed, not computed, so this counts as $O(1)$.

After the split, we relaunch the same process with $N - L - R + L \cdot R$ inequalities, which is less than $\frac{N^2}{4}$.

Each merge takes $O(A+V)$ operations, so there are $O(\frac{N^2}{4}(A+V))$ operations, which is greater than the previous $O(N \cdot (A+V))$.

Therefore, we have the following relation:

$$\Omega(A, V, 0, N) = \Omega\left(A, V-1, 0, \frac{N^2}{4}\right) + O\left(\frac{N^2}{4} \cdot (A+V)\right)$$

A.1.6 Merging and upper bound

So we have the following results, and we will now prove that the upper bounds U on Ω holds by induction.

$$\begin{array}{llll}
(1) & \Omega(A,V,0,0) & \leq & K_1 \cdot V \\
(2) & \Omega(A,V,0,1) & \leq & K_2 \cdot A + K_3 \cdot V \\
(3) & \Omega(A,V,0,N) & \leq & K_4 \cdot \left(\frac{N^2}{4} \cdot (A+V)\right) \\
 & & & +\Omega\left(A, V-1, 0, \frac{N^2}{4}\right) \\
(4) & \Omega(A,V,E,N) & \leq & K_0 \cdot (E \cdot (A+V)(E+N)) \\
 & & & +\Omega(A, V-E, 0, N)
\end{array}$$

A.1.7 Proof by induction

Let us examine the base cases (1) and (2). They are all satisfied if we choose $K_5 \geq \max(K_1, K_2, K_3)$ in the provided formula of section A.1.

Now let us examine the cases (3) and (4) by induction to prove that the given upper bound expression U holds.

(4) The complete induction hypothesis let us assume that

$$\forall v < V. \quad \Omega(A, v, E, N) \leq U(A, v, E, N)$$

.

Therefore, for $v = V - E$:

$$\begin{array}{lll}
\Omega(A,v,0,N) & \leq & K_5 \cdot (2v(A+v) \sum_{k=2}^{v} \frac{N^{2^{k-1}}}{2^{2^k-1}} \\
 & & +f(A,v,0,N))
\end{array}$$

$$\leq K_5 \cdot \left(2V(A+V)\sum_{k=2}^{V-E} \frac{N^{2^{k-1}}}{2^{2^k-1}} + f(A, V-E, 0, N)\right)$$

Using this result in (4), we obtain:

$$\begin{array}{lll}
\Omega(A,V,E,N) & \leq & K_0 \cdot (E \cdot (A+V)(E+N)) \\
 & & +K_5 \cdot (\quad 2V(A+V) \sum_{k=2}^{V-E} \frac{N^{2^{k-1}}}{2^{2^k-1}} \\
 & & \qquad +f(A, V-E, 0, N))
\end{array}$$

This is trivial for $E = 0$, so let us assume $E > 0$. We regroup terms to form U, and then examine the remaining terms.

$$\begin{array}{ll}
\multicolumn{2}{l}{\Omega(A,V,E,N)} \\
\leq & U(A,V,E,N) \\
 & +K_0 \cdot (E \cdot (A+V)(E+N)) \\
 & +K_5 \cdot (\quad -2V(A+V) \sum_{k=V-E+1}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} \\
 & \qquad +f(A,V-E,0,N) - f(A,V,E,N))
\end{array}$$

Furthermore, if we assume $K_5 \geq K_0$:

$$\begin{array}{lllll}
 & K_0 \cdot (E \cdot (A+V)(E+N)) & + & K_5(f(A,V-E,0,N) \\
 & & & -f(A,V,E,N)) \\
\leq & K_5 \cdot (E \cdot (A+V)(E+N)) & + & (V-E) \cdot (A+V-E)(N) \\
 & & & -V \cdot (A+V)(N+E)) \\
\leq & K_5 \cdot (E \cdot (A+V)(E+N)) & + & (V-E) \cdot (A+V)(N+E) \\
 & & & -V \cdot (A+V)(N+E)) \\
\leq & 0 & &
\end{array}$$

So by simplification, we obtain:

$$\Omega(A,V,E,N) \leq U(A,V,E,N)$$

(3) The complete induction hypothesis let us assume that

$$\forall v < V. \quad \Omega(A, v, E, N) \leq U(A, v, E, N)$$

Using this result in (3) for $v = V - 1$, we obtain:

$$\begin{array}{ll}
\multicolumn{2}{l}{\Omega(A,V,0,N)} \\
\leq & K_4 \cdot (\frac{N^2}{4} \cdot (A+V)) + U(A, V-1, 0, \frac{N^2}{4}) \\
\leq & K_4 \cdot (\frac{N^2}{4} \cdot (A+V)) + \\
 & K_5 \cdot (\quad 2(V-1)(A+V-1) \sum_{k=2}^{V-1} \frac{(N^2/4)^{2^{k-1}}}{2^{2^k-1}} + \\
 & \qquad f(A, V-1, 0, N^2/4)) \\
\leq & K_4 \cdot (\frac{N^2}{4} \cdot (A+V)) + \\
 & K_5 \cdot (\quad 2V(A+V) \sum_{k=2}^{V-1} \frac{N^{2^{k+1-1}}}{2^{2^{k+1}-1}} + \\
 & \qquad f(A, V-1, 0, N^2/4)) \\
\leq & K_4 \cdot (\frac{N^2}{4} \cdot (A+V)) + \\
 & K_5 \cdot (\quad 2V(A+V) \sum_{k=2}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} + \\
 & \qquad f(A, V-1, 0, N^2/4)) \\
\leq & U(A,V,0,N) + \\
 & K_4 \cdot (\frac{N^2}{4} \cdot (A+V)) + \\
 & K_5 \cdot (\quad -2V(A+V)\frac{N^2}{4} + \\
 & \qquad f(A, V-1, 0, N^2/4) - f(A,V,0,N))
\end{array}$$

Assuming that $K_5 \geq K_4$:

$$\begin{array}{ll}
\multicolumn{2}{l}{\Omega(A,V,0,N)} \\
\leq & U(A,V,0,N) + \\
 & K_5 \cdot (\quad \frac{N^2}{4} \cdot (A+V) \\
 & \qquad -2V(A+V)\frac{N^2}{4} + \\
 & \qquad f(A, V-1, 0, N^2/4) - f(A,V,0,N)) \\
\leq & U(A,V,0,N) + \\
 & K_5 \cdot (\quad -\frac{N^2}{4} \cdot (A+V) \\
 & \qquad f(A, V-1, 0, N^2/4) - f(A,V,0,N))
\end{array}$$

By bounding from above :

$$\begin{array}{ll}
\multicolumn{2}{l}{\Omega(A,V,0,N)} \\
\leq & U(A,V,0,N) + \\
 & K_5 \cdot (\quad -V(A+V)\frac{N^2}{4} \\
 & \qquad +(V-1)(A+V-1)\frac{N^2}{4} - V(A+V)N) \\
\leq & U(A,V,0,N) + \\
 & K_5 \cdot (\quad -V(A+V)\frac{N^2}{4} \\
 & \qquad +V(A+V)\frac{N^2}{4} - V(A+V)N) \\
\leq & U(A,V,0,N) + K_5 \cdot (-V(A+V)N) \\
\leq & U(A,V,0,N)
\end{array}$$

QED.

A.1.8 Size and execution time

For each variable solved from an equality, the size of its assigned expression will be bounded from above by $P_0 \cdot (A + V - 1)$; where V is the number of variables at this point and P_0 a certain constant. For each variable solved from an inequality, the size of its assigned expression (the mean of the min of lower bounds and max of upper bounds) will be in $P_1((A + V - 1) \cdot N)$, where N is the number of inequaltiies at this point, knowing that the next time, there might be up to $N^2/2$ new inequalities.

Therefore, with E equalities, the size of the program is bounded from above by:

$$\begin{array}{lll}
P_0 \cdot (A+V-1) + & \ldots & +P_0 \cdot (A+V-E) + \\
P_1 \cdot (A+V-1)N + & \ldots & +P_1 \cdot (A+V-V)\frac{N^{2^{V-1}}}{2^{2^V-2}}
\end{array}$$

This can be bounded from above by

$$P_2 \cdot \left((A+V)\left(E + \frac{N^{2^{V+1}-1}}{2^{2^{V+1}-2}}\right)\right)$$

where $P_2 = \max(P_0, P_1)$

As we do not have any loops in the linear case, the execution time is roughly linear to the size of the program, so it has the same complexity.

A.2 Linear Integer complexity

Let us examine the number of times $\Omega(E, N, V)$ given the number of equalities E, inequalities N and output variables V.

By induction on the number of output variables V, we show that

$$\Omega(E, N, V) \leq U(E, N, V)$$

where

$$U(E, N, V) = 2 + 2\sum_{k=1}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} + \min(V, E)$$

By arithmetic properties, it implies the following expected result

$$\Omega(E, N, V) \leq 2 + \frac{N^{2^V}}{2^{2^{V+1}-1}} + \min(V, E)$$

The base case is $\Omega(E, N, 0) = 1$, so this holds. Indeed, without output variables, all equations go directly to the precondition. We suppose now that $V \geq 1$.

1. The first remark is that if there are equalities remaining ($E \geq 1$), we can remove one variable in one step.

$$\Omega(E, N, V) \leq \Omega(E-1, N, V-1) + 1$$

By induction hypothesis, we obtain:

$$\Omega(E, N, V) \leq 1 + 2 + 2\sum_{k=1}^{V-1} \frac{N^{2^{k-1}}}{2^{2^k-1}} + \min(V-1, E-1)$$

$$\Omega(E, N, V) \leq U(E, N, V) - 2\frac{N^{2^{V-1}}}{2^{2^V-1}} \leq U(E, N, V)$$

Now, equations are removed.

2. If a variable is bounded on one side only by M inequalities:

$$\begin{array}{rcl} \Omega(0, N, V) & \leq & \Omega(0, N-L, V-1) \\ & \leq & U(0, N-L, V-1) \\ & \leq & U(0, N, V) \end{array}$$

3. Partial modulo ending does not make the behavior of synthesis or synthesized program worse, only better, so we can ignore it for the purpose of complexity upper bound.

4. After handling equalities and inequalities of step 6, we can assume that $N \geq 2$. If L is the number of lower bounds and R the number of upper bounds, it generates $L \cdot R$ new inequalities and R equalities, where $1 \leq L \leq N-1$, $1 \leq R \leq N-1$ and of course $L + R \leq N$. If $L < R$, we would split on the L equations, so by taking R we can assume that $R \leq N/2$.

$\Omega(0, N, V) \leq \max_{L,R} \Omega(R, N-L-R+L \cdot R, V-1+R)+1$

As the next steps will be consecrated to removing the R equalities, we obtain that:

$\Omega(0, N, V) \leq \max_{L,R} \Omega(0, N-L-R+L \cdot R, V-1)+1+R$

Among the choices of L, the highest complexity is given for $L = N - R$.

$\Omega(0, N, V) \leq \max_R \Omega(0, (N-R) \cdot R, V-1) + 1 + R$

As $R \leq N/2$, we can maximize it with $N/2$

$\Omega(0, N, V) \leq \Omega(0, N^2/4, V-1) + 1 + N/2$

So by induction :

$\Omega(0, N, V) \leq 2 + 2\sum_{k=1}^{V-1} \frac{(N^2/4)^{2^{k-1}}}{2^{2^k-1}} + 1 + N/2$

$\Omega(0, N, V) \leq 2 + 2\sum_{k=1}^{V-1} \frac{N^{2^k}}{2^{2^{k+1}-1}} + 1 + N/2$

$\Omega(0, N, V) \leq 2 + 2\sum_{k=2}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} + 1 + N/2$

$\Omega(0, N, V) \leq 2 + 2\sum_{k=1}^{V} \frac{N^{2^{k-1}}}{2^{2^k-1}} + 1 + N/2 - 2(N/2)$

$\Omega(0, N, V) \leq U(0, N, V)$

QED.

Acknowledgments

We thank Barbara Jobstmann and Martin Odersky for useful discussions and suggestions.

References

E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, 1995.

U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. ISBN 0898382890.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.

C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.

A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent c. In *Conf. Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, 2009.

D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.

E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, 2007.

S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959.

J. Ferrante and C. W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.

S. Ginsburg and E. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

S. Ginsburg and E. Spanier. Semigroups, Presburger formulas and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, 2006.

B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, 2007.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. (freely available), 1993. URL `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

N. Klarlund and M. I. Schwartzbach. Graph types. In *POPL*, Charleston, SC, 1993.

J. H. Kukula and T. R. Shiple. Building circuits from relations. In *CAV*, 2000.

V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. URL `http://dx.doi.org/10.1007/s10817-006-9042-1`.

Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357084.357090.

Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

T. Nipkow. Linear quantifier elimination. In *IJCAR*, 2008.

M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

D. C. Oppen. Reasoning about recursively defined data structures. In *POPL*, pages 151–157, 1978.

N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, 2006.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/135226.135233.

A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

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

A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, 2010.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.

V. Weispfenning. Complexity and uniformity of elimination in presburger arithmetic. In *Proc. International Symposium on Symbolic and Algebraic Computation*, pages 48–53, 1997.

C. G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004.

C. G. Zarba. Combining sets with cardinals. *J. of Automated Reasoning*, 34 (1), 2005.

K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.

K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI*, 2009.