Int J Softw Tools Technol Transfer (2013) 15:455–474
DOI 10.1007/s10009-011-0217-7

SYNTHESIS

# Functional synthesis for linear arithmetic and sets

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

Published online: 22 November 2011


**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. To guarantee correctness and applicability to software (and not just hardware), these algorithms should also support unbounded data types, such as numbers and data structures. To obtain appropriate synthesis algorithms, we propose to generalize decision procedures into predictable and complete synthesis procedures. Such procedures are guaranteed to find the 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 starting from a quantifier elimination decision procedure for Boolean algebra of set with Presburger arithmetic and transforming it into a synthesis procedure. Our procedure also works in the presence of parametric coefficients. We establish 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, give up a deterministic execution model, or provide code skeletons.

R. Piskac was supported by the EPFL School of Computer and Communication Sciences and in part by the Swiss National Foundation Grant SCOPES IZ73Z0_127979. P. Suter was supported by the Swiss National Science Foundation Grant 200021_120433.

V. Kuncak (✉) · M. Mayer · R. Piskac · P. Suter
School of Computer and Communication Sciences (I&C),
Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland
e-mail: viktor.kuncak@epfl.ch

**Keywords** Software synthesis · Complete synthesis procedures · Decision procedures · Linear integer arithmetic

## 1 Introduction

Synthesis of software from specifications [42,43] promises to make programmers more productive. Despite substantial recent progress [55,58,59,62], synthesis is limited to small pieces of code. We expect that this will continue to be the case for some time in the future for two reasons: (1) synthesis is algorithmically a difficult problem, and (2) synthesis requires detailed specifications, which for large programs become difficult to write.

We therefore expect that practical applications of synthesis lie 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, just like a compilation attempt succeeds for any well-formed program. Our starting point for such synthesis algorithms are *decision procedures*.

A decision procedure for satisfiability of a class of formulas accepts a formula in its class and checks whether the formula has a solution. On top of this basic functionality, many decision procedure implementations provide the additional feature of generating a satisfying assignment (a model) whenever the given formula is satisfiable. Such a model-generation functionality has many uses, including better error reporting in verification [41] and test-case generation [1]. An important insight is that model generation facility of decision procedure could also be used as an advanced computation mechanism. Given a set of values for some of the variables, a constraint solver can at run-time find the values of the remaining variables such that a given constraint holds. Two recent examples of integrating such a mechanism into

a programming language are the quotations of the *F#* language [54] and a Scala library [30], both interfacing to the Z3 satisfiability modulo theories (SMT) solver [10]. Such mechanisms promise to bring the algorithmic improvements of SMT solvers to declarative paradigms such as constraint logic programming [29]. However, they involve a possibly unpredictable search at run-time and require the deployment of the entire decision procedure as a component of the run-time system.

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. The generated code then computes the desired values of variables at run-time. Such code is thus specific to the desired constraint and can be more efficient. It does not require the decision procedure to be present at run-time and gives the developer static feedback by checking the conditions under which the generated solution will exist and be unique. We use the term *synthesis* for our approach because it starts from an implicit specification and involves compile-time precomputation. Because it computes a function that satisfies a given input/output relation, we call our synthesis *functional*, in contrast to reactive synthesis approaches [50] (another term for the general direction of our approach is AE-paradigm or Skolem paradigm). Finally, we call our approach *complete* because it is guaranteed to work for all specification expressions from a well-specified class.

We demonstrate our approach by describing synthesis algorithms for the domains of linear arithmetic and collections of objects. We have implemented these synthesis algorithms and deployed them as a compiler extension of the Scala programming language [46]. We have found that using such constraints we were able to express a number of program fragments in a more natural way, stating the invariants that the program should satisfy as opposed to the computation details of establishing these invariants.

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 synthesis include conversions of quantities expressed in terms of multiple units of measure, coordinate transformations, as well as a substantially more general notion of pattern matching on integers, going well beyond matching on constants or $(n + k)$-patterns of the Haskell programming language [26].

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 manipulating sets of objects in high-level descriptions of algorithms, from simple operations such as choosing an element from a set or a fresh element, or splitting sets subject to size constraints. Such constructs arise in pseudo-code notations, and they provide a useful addition to the transformations previously developed for the SETL programming language [11,56]. Regarding data structures this paper focuses on sets, but the approach applies to other constraints for which decision procedures are available [35], including multisets [47,48,66] and algebraic data types [53].

*Contributions* This paper makes the following contributions:

1. 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 specification, given as an expression $\lambda x.F$ of type $\alpha \Rightarrow \mathsf{bool}$. Our compiler extension rewrites calls to choose into efficient code that finds a value $x$ of type $\alpha$ such that $F$ is true. The generated code computes $x$ as a function of the free variables (parameters) of the expression $F$. This deployment is easy to understand by programmers because it has the same semantics as invoking a constraint solver at run-time. It does not impact the semantics or efficiency of existing programming language constructs, because the execution outside choose remains unchanged.
2. Building on the choose primitive, we show how to support pattern matching expressions that are substantially more expressive than the existing ones, using the full expressive power of the term language of a decidable theory.
3. 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. Most existing procedures based on quantifier elimination are directly amenable to our approach.
4. As a first illustrative example, we describe synthesis procedures for propositional logic and rational arithmetic. We show that, compared with 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 (at compile time) a given constraint into a solved form that can be executed, avoiding most of the run-time search. The synthesized code is guaranteed to be correct by construction.
5. As our core implemented example, we present synthesis for linear arithmetic over unbounded integers. Given an integer linear arithmetic formula and a separation of variables into output variables and parameters, our procedure constructs (1) a program that computes the values of outputs given the values of inputs and (2) the weakest among the conditions on inputs that guarantees the

existence of outputs (the domain of the relation between inputs and outputs).

6. We show that the synthesis for integer arithmetic can be extended to the non-linear case where coefficients multiplying output variables are expressions over parameters that are known only at run-time. We have implemented this extension and have found that it increases the range of supported specifications. It shows that we can have complete functional synthesis at compile-time for specifications for which the satisfiability over the space of all parameters is undecidable, as long as the problem becomes decidable when the parameters are fixed at run-time.
7. We also present an implemented synthesis procedure for Boolean algebra with Presburger arithmetic (BAPA), a logic of constraints on sets and their sizes. This algorithm illustrates that complete functional synthesis applies not only to numerical computations, but also to the very important domain of data structure manipulations. This result also illustrates the idea of the *composition of synthesis procedures*. While the implementations of BAPA decision procedures work by reduction to integer arithmetic decision procedures [33,36], we here show how to build a synthesis procedure for BAPA on top of our synthesis procedure for integer linear arithmetic.
8. We describe our experience in using synthesis as a plugin for the Scala compiler. Our implementation is publicly available at http://lara.epfl.ch/dokuwiki/comfusy and can be used as a starting point for the development of further synthesis approaches.

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

Our 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 and s are not strict. After correcting the error in the specification, replacing $\mathsf{m} \leq 60$ with $\mathsf{m} < 60$ and $\mathsf{s} \leq 60$ with $\mathsf{s} < 60$, the synthesizer emits no warnings and generates code corresponding to the following:

```
val (hrs, mns, scs) = {
val loc1 = totsec div 3600
val num2 = totsec + ((−3600) ∗ loc1)
val loc2 = min(num2 div 60, 59)
val loc3 = totsec + ((−3600) ∗ loc1) + (−60 ∗ loc2)
(loc1, loc2, loc3)
}
```

The absence of warnings guarantees that the solution always exists and that it is unique. By writing the code in this style, the developer directly ensures that the condition $\mathsf{h} * 3600 + \mathsf{m} * 60 + \mathsf{s} == \mathsf{totsec}$ will be satisfied, making the program easily understood. Note that, if the developer imposes the constraint

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

our system emits the following warning:

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

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

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. Our approach supports a much richer set of patterns, as illustrated by the following fast exponentiation code that does 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 $\mathsf{fp}(m, b, i) = mb^i$, which we can prove by induction. Indeed, if we consider the case $2 * j + 1$, we observe:

$$\mathsf{fp}(m, b, i) = \mathsf{fp}(m, b, 2j + 1) = \mathsf{fp}(mb, b^2, j)$$
$$\text{(by induct. hypothesis)} = mb(b^2)^j = mb^{2j+1} = mb^i$$

Note how the pattern matching on integer arithmetic expressions exposes the equations that make the inductive proof

clearer. 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, for example, 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 $z$.

Our approach and implementation also work for parameterized integer arithmetic formulas, which become linear only once the parameters are known. For example, our synthesizer accepts the following specification that decomposes an offset of a linear representation of a three-dimensional array with statically unknown dimensions into indices for each coordinate:

```
val (x1, y1, z1) = choose((x: Int, y: Int, z: Int) ⇒
offset == x + dimX * y + dimX * dimY * z &&
0 ≤x && x < dimX &&
0 ≤y && y < dimY &&
0 ≤z && z < dimZ)
```

Here dimX, dimY, dimZ are variables whose value is unknown until runtime. Note that the satisfiability of constraints that contain multiplications of variables is in general undecidable. In such parameterized case our synthesizer is complete in the sense that it generates code that (1) always terminates, (2) detects at run-time whether a solution exists for current parameter values, and (3) computes one solution whenever a solution exists.

In addition to integer arithmetic, other 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)
```

It turns out that for the above code our synthesizer emits a warning indicating that there are cases where the constraint has no solutions. Indeed, there are no solutions when the set s is of odd size. If we weaken the specification 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)
```

Then our synthesizer can prove that the code has a solution for all possible input sets s. The synthesizer emits code that, for each input, computes one such solution. The nature of constraints on sets is that if there is one solution, then there are many solutions. Our synthesizer resolves these choices at compile time, which means that the generated code is deterministic.

## 3 From decision to synthesis procedures

We next define precisely the notion of a synthesis procedure and describe a methodology for deriving synthesis procedures from decision procedures.

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

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

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

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

We can translate the above choose construct into the following sequence of commands in a guarded command language [12]:

```
assert(∃x.F);
havoc (r);
assume (F[x := r]);
```

The simplicity of the above translation indicates that it is natural to represent choose within existing verification

systems (e.g. [16,69]) The use of choose can help verification because the desired property $F$ is explicitly assumed and can aid in proving the subsequent program assertions.

*Model-generating decision procedures* As a starting point for our synthesis algorithms for choose invocations we consider a model-generating decision procedure. Given $F \in \mathsf{Formulas}$ we expect this decision procedure 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. We write $Z(F)=\sigma$ or $Z(F)=\mathsf{unsat}$ to denote the result of applying the decision procedure 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 run-time can be considered as a baseline for our approach. In this scenario, we replace the statement (1) with the code

```
F = makeFormulaTree(makeVars(x), makeGroundTerms(a));
r= (Z(F)match {
  case σ⇒(σ(x1), . . . , σ(xn))
  case unsat ⇒throw new Exception("No solution exists")
})
```

Such dynamic invocation approach is flexible and useful. However, there are important performance and predictability advantages of an alternative *compilation* approach.

*Synthesis based on decision procedures* Our goal is therefore to explore a compilation approach where a modified decision procedure is invoked at compile time, converting the formula into a solved form.

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

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

such that the following two implications are valid:

$$(\exists \mathbf{x}.F) \;\to\; \mathsf{pre}$$
$$\mathsf{pre} \;\to\; F[\mathbf{x} := \boldsymbol{\Psi}]$$

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

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

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

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

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

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

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

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

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

The existence of a model-generating decision procedure implies the existence of a 'trivial' synthesis procedure, which satisfies Definition 1 but simply invokes the decision procedure at run-time. (In the realm of conventional programming languages, this would be analogous to 'compiling' the code by shipping its source code bundled with an interpreter.) The usefulness of the notion of synthesis procedure, therefore, comes from the fact that we can often 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 for the processing of more traditional programming language constructs, we do believe that there is space in the future for mixed approaches, such as 'just-in-time synthesis' and 'profiling-guided synthesis'.

*Efficiency of synthesis* We introduce the following measures to quantify the behavior of synthesis procedures as a function of the specification expression $F$:

- 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 for the parameters $\mathbf{a}$.

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

*From quantifier elimination to synthesis* The precondition $\mathsf{pre}$ can be viewed as a result of applying quantifier elimination (see, e.g. [23, Page 67], [44]) to remove $\mathbf{x}$ from $F$, with the following differences.

1. Synthesis procedures strengthen quantifier elimination procedures by identifying not only $\mathsf{pre}$ but also emitting the code $\boldsymbol{\Psi}$ that efficiently computes a *witness* for $\mathbf{x}$.
2. Quantifier elimination is typically applied to arbitrary quantified formulas of first-order logic and aims to successively eliminate all variables. To enable recursive application of variable elimination, $\mathsf{pre}$ must be in the same language of formulas as $F$. This condition is not required in the final step of synthesis procedure, because no further elimination is applied to the final precondition. Therefore, if the final precondition becomes a run-time check, it can contain arbitrary executable code. If the final precondition becomes a compile-time satisfiability check for the totality of the relation, then it suffices for it to be in any decidable logic.
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 $\boldsymbol{\Psi}$ or the time to evaluate $\boldsymbol{\Psi}$. For some domains, it can be computationally more difficult to compute (or even 'print') the solution than to simply check the existence of a solution.

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

## 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 that we have a function $\mathsf{witn}(x, F)$ that corresponds to constructive quantifier elimination step for one variable and produces a term $\Psi$ such that $F[x := \Psi]$ holds iff $\exists x.F$ holds. We can then lift $\mathsf{witn}(x, F)$ to synthesis for any number of variables, using the (non-tail recursive) translation scheme in Fig. 1. This translation includes the base case in which there are no variables to eliminate, so $F$ becomes the precondition, and the recursive case that applies the $\mathsf{witn}$ function.

In implementation we can use local variable definitions instead of substitutions. Given (1), we generate as $\boldsymbol{\Psi}$ a Scala code block

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

where the variables in $\Psi_n$ directly refer to variables computed in $\Psi_1, \ldots, \Psi_{n-1}$ and where $\mathsf{FV}(\Psi_i) \subseteq \mathsf{FV}(F) \backslash \{x_i, \ldots, x_n\}$. A consequence of this recursive translation pattern is that the synthesized code computes values in reverse order compared with 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

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

If the formula does not have the form $x = t \wedge F$, we can often rewrite it into this form using theory-specific transformations.

$$[\![\_\ ,\ \_]\!] : \bigcup_n \left(\mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n\right)$$

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

**Fig. 1** Successive elimination of variables for synthesis

### 4.3 Output-independent preconditions

Whenever $\mathsf{FV}(F_1) \cap \mathbf{x}_s = \emptyset$, we can apply the following synthesis rule:

$$[\![\mathbf{x}, F_1 \wedge F_2]\!] = \begin{array}{l}\texttt{let}\ \ (\mathsf{pre}, \boldsymbol{\Psi}) = [\![\mathbf{x}, F_2]\!]\ \texttt{in} \\ (\mathsf{pre} \wedge F_1, \boldsymbol{\Psi})\end{array}$$

which moves a 'constant' conjunct of the specification into the precondition. We assume that this rule is applied whenever possible and do not explicitly mention it in the sequel.

### 4.4 Propositional connectives in first-order theories

Consider a quantifier-free formula in some first-order theory. Consider the tasks of checking formula satisfiability or applying elimination of a variable. For both tasks, we can first rewrite the formula into disjunctive normal form and then process each disjunct independently. This allows us to focus on handling conjunctions of literals as opposed to arbitrary propositional combination.

We next show that we can similarly use disjunctive normal form in synthesis. Consider a formula $D_1 \vee \cdots \vee D_n$ in disjunctive normal form. We can apply synthesis to each $D_i$ yielding a precondition $\mathsf{pre}_i$ and the solved form $\boldsymbol{\Psi}_i$. We can then synthesize code with conditionals that select the first $\boldsymbol{\Psi}_i$ that applies:

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

Although the disjunctive normal form can be exponentially larger than the original formula, the transformation to disjunctive normal form is used in practice [51] 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 may need different synthesized values for variables in different disjuncts.

Other methods can have better worst-case quantifier elimination complexity [9,17,44,63] than disjunctive normal form approaches. We discuss these alternative approaches in the sequel as well, but it is the above disjunctive normal form approach that we currently use in our implementation.

### 4.5 Synthesis for propositional logic

Our paper focuses on synthesis for formulas over *unbounded* domains. Nonetheless, to illustrate the potential asymptotic gain of precomputation in synthesis, we illustrate synthesis for the case when $F$ is a propositional formula (see, e.g. [38] for a more sophisticated approach to this problem). Suppose that $\mathbf{x}$ are output variables and $\mathbf{a}$ are the remaining propositional variables (parameters) in $F$.

To synthesize a function from $\mathbf{a}$ to $\mathbf{x}$, build an ordered binary decision diagram (OBDD) [6] for $F$, treating both $\mathbf{a}$ and $\mathbf{x}$ as variables for OBDD construction, and using a variable ordering that puts all parameters $\mathbf{a}$ before all output variables $\mathbf{x}$. Then split the OBDD graph at the point where all the decisions on $\mathbf{a}$ have been made. That is, consider the set of nodes that terminate on some paths on which all decisions on $\mathbf{a}$ have been made and no decisions on $\mathbf{x}$ have been made. For each of these OBDD nodes, we precompute whether this node reaches the $\mathsf{true}$ sink node. As the result of synthesis, we emit the code that consists of nested if-then-else tests encoding the decisions on $\mathbf{a}$, followed by the code that, for each non-false node those values of $\mathbf{x}$ that trace one path to the $\mathsf{true}$ sink node.

Consider the code generated using the method above. Note that, although the size of the code is bounded by a single exponential, the code executes in time close to linear in the total number of variables $\mathbf{a}$ and $\mathbf{x}$. 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 logic synthesis (and, more generally, for NP-hard constraints over bounded domains) we can precompute solutions and generate code that computes the desired values in deterministic polynomial time in the size of inputs and outputs.

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 $c_0 + c_1 x_1 + \cdots + c_n x_n$, and the relations in the language are $<$ and $=$. Synthesis for this theory can be used to synthesize exact fractional arithmetic computations (or floating-point computations if we are willing to ignore the rounding errors). It also serves as an introduction to the more complex problem of integer arithmetic synthesis that we describe in the following sections.

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 Sect. 4.4, we consider conjunctions of literals. The method follows Fourier–Motzkin elimination [52]. 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 simply apply the one-point rule synthesis (Sect. 4.2). This step amounts to Gaussian elimination. We follow this step whenever possible, so we first eliminate 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 $a_p, b_q$. 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* We next examine the size of the generated code for linear rational arithmetic. The elimination of input variables using equalities is a polynomial-time transformation. Suppose that after this elimination we are left with $N$ inequalities and $V$ remaining input variables. The above inequality elimination step for one variable replaces $N$ inequalities with $(N/2)^2$ inequalities in the worst case. After eliminating all output variables, an upper bound on the formula increase is $(N/2)^{2^V}$. Therefore, the generated formula can be in the worst case doubly exponential in the number of output variables $V$. However, for a fixed $V$, the generated code size is a (possibly high-degree) polynomial of the size of the input formula. Also, if there are four or fewer inequalities in the original formula, the final size is polynomial, regardless of $V$. Finally, note that the synthesis time and the execution time of synthesized code are polynomial in the size of the generated formula.

### 5.2 Disjunctions for linear rational arithmetic

We next consider linear arithmetic constraints with disjunctions, which are constraints for which the satisfiability is NP-complete. 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 Sect. 4.4. We then obtain a 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 with the disjunctive normal form approach of Sect. 4.4.

The starting point of this method are quantifier elimination techniques that avoid disjunctive normal form transformation, e.g. [17,44], [5, 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 [44]. This quantifier elimination method gives us a way to compute pre.

We next present how to extend this quantifier elimination method to synthesis, namely to the computation of $\mathsf{witn}(x, F)$. Consider a substitution in quantifier elimination step that replaces variable $x_i$ with the term $m$. We then extend this step to also attach to each literal a special substitution syntactic form $(x_i \mapsto m)$. When using this process to eliminate one variable, the size of the formula can increase quadratically. After eliminating all output variables, we obtain a formula pre with additional annotations; the size of this formula is bounded by $n^{2^{O(V)}}$ where $n$ is the original formula size. (Again, although it is doubly exponential in $V$, it is not exponential in $n$.)

We can therefore 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, at synthesis time, determine whether the truth values of literals imply that pre is true. Indeed, such computation 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 the depth $n^{2^{O(V)}}$, because it tests the values of all literals

in the result of quantifier elimination. For a constant number of variables $V$, this tree represents a synthesized program whose running time is polynomial in $n$. Thus, we have shown that using basic methods of quantifier elimination (without relying on detailed geometric facts about the theory of linear rational arithmetic) we can synthesize for each specification formula a polynomial-time function that maps the parameters to the desired values of output variables.

## 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 $c_0 + c_1 x_1 + \cdots + c_n x_n$, $n \geq 0$, $c_i$ is an integer constant and $x_i$ is an integer variable. Atoms are built using the relations $\geq$, $=$ and $|$. The atom $c|t$ is interpreted as true iff the integer constant $c$ divides term $t$. We use $a < b$ as a shorthand for $a \leq b \wedge \neg(a = b)$. We describe a synthesis algorithm that works for conjunction of literals.

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

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

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

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

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

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

### 6.1 Solving equality constraints for synthesis

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

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

We instead perform synthesis based on one of the improved algorithms for solving integer equations. This algorithm avoids the multiplication of the remaining constraints by simultaneously replacing all $n$ output variables $\mathbf{y}$ in $E$ with $n - 1$ fresh output variables $\boldsymbol{\lambda}$. Using this algorithm we obtain the synthesis procedure in Fig. 2. An invocation of $\mathsf{eqSyn}(\mathbf{y}, F)$ is similar to $[\![\mathbf{y}, F]\!]$ but returns a triple $(\mathsf{pre}, \boldsymbol{\Psi}, \boldsymbol{\lambda})$, which in addition to the precondition $\mathsf{pre}$ and the witness term tuple $\boldsymbol{\Psi}$ also has the fresh variables $\boldsymbol{\lambda}$.

#### 6.1.1 *The* eqSyn *synthesis algorithm*

Consider the application of eqSyn in Fig. 2 to the equation $\Sigma_{i=1}^{m} \beta_i b_i + \Sigma_{j=1}^{n} \gamma_j y_j = 0$. If there is only one output variable, $y_1$, we directly eliminate it from the equation. Assume therefore $n > 1$. Let $d = \gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n)$. If $d > 1$ we can divide all coefficients by $d$, so assume $d = 1$.

$$
\begin{array}{l}
[\![\_\,,\,\_]\!] : \bigcup_n \left(\mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n\right) \\
[\![(\mathbf{y}, \mathbf{x}), E \wedge F]\!] = \\
\quad \texttt{let}\ (\mathsf{pre}_Y, \boldsymbol{\Psi}_Y, \boldsymbol{\lambda}) = \mathsf{eqSyn}(\mathbf{y}, E) \\
\qquad F' = \mathsf{simplify}(F[\mathbf{y} := \boldsymbol{\Psi}_Y]) \\
\qquad (\mathsf{pre}, (\boldsymbol{\Psi}_\lambda, \boldsymbol{\Psi}_X)) = [\![(\boldsymbol{\lambda}, \mathbf{x}), F']\!] \\
\qquad \mathsf{pre}_{Y0} = \mathsf{pre}_Y[\boldsymbol{\lambda} := \boldsymbol{\Psi}_\lambda, \mathbf{x} := \boldsymbol{\Psi}_X] \\
\qquad \boldsymbol{\Psi}_{Y0} = \boldsymbol{\Psi}_Y[\boldsymbol{\lambda} := \boldsymbol{\Psi}_\lambda, \mathbf{x} := \boldsymbol{\Psi}_X] \\
\quad \texttt{in} \\
\qquad (\mathsf{pre}_{Y0} \wedge \mathsf{pre}, (\boldsymbol{\Psi}_{Y0}, \boldsymbol{\Psi}_X))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{eqSyn}: \bigcup_n \mathsf{Vars}^n \times \mathsf{Formulas} \to \mathsf{Formulas} \times \mathsf{Terms}^n \times \mathsf{Vars}^{n-1} \\
\mathsf{eqSyn}(y_1, t + \gamma_1 y_1 = 0) \ = \ ((\gamma_1 | t),\ -t/\gamma_1,\ ()) \\
\mathsf{eqSyn}(y_1, \ldots, y_n, t + \Sigma_{j=1}^{n} \gamma_j y_j = 0) \ = \ (\text{for } t = \Sigma_{i=1}^{m} \beta_i b_i) \\
\quad \texttt{let} \quad d = \gcd(\beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n) \\
\quad \texttt{if} \quad (d > 1) \quad \mathsf{eqSyn}(y_1, \ldots, y_n,\ t/d + \Sigma_{j=1}^{n} (\gamma_j/d) y_j = 0) \\
\quad \texttt{else let} \quad (\mathbf{s}_1, \ldots, \mathbf{s}_{n-1}) = \mathsf{linearSet}(\gamma_1, \ldots, \gamma_n) \\
\qquad\qquad\quad (w_1, \ldots, w_n) = \mathsf{partSol}(t, \gamma_1, \ldots, \gamma_n) \\
\qquad\qquad\quad \mathsf{pre} = (\gcd(\gamma_1, \ldots, \gamma_n) | t) \\
\qquad\qquad\quad \lambda_1, \ldots, \lambda_{n-1} - \text{fresh variable names} \\
\qquad\qquad\quad \boldsymbol{\Psi} = (w_1, \ldots, w_n) + \lambda_1 \mathbf{s}_1 + \ldots + \lambda_{n-1} \mathbf{s}_{n-1} \\
\quad \texttt{in}\ (\mathsf{pre}, \boldsymbol{\Psi}, \boldsymbol{\lambda})
\end{array}
$$

**Fig. 2** Algorithm for synthesis based on integer equations

Our goal is to derive an alternative definition of the set $K = \{\mathbf{y} \mid \Sigma_{i=1}^{m} \beta_i b_i + \Sigma_{j=1}^{n} \gamma_j y_j = 0\}$ 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.

Recall that a *semilinear set* [20] is a finite union of linear sets. Given an integer vector $\mathbf{b}$ and a finite set of integer vectors $S$, a *linear set* is a set $\{\mathbf{x} \mid \mathbf{x} = \mathbf{b} + \mathbf{s}_1 + \cdots + \mathbf{s}_n; s_i \in S; n \geq 0\}$. Ginsburg and Spanier [20,21] showed that the set of all solutions of a Presburger arithmetic formula is always a semilinear set, which implies that $K$ is semilinear. However, we cannot apply this result directly because the values of parameter variables are not known until run-time. Instead, we proceed in the following steps, as shown in Fig. 2:

1. obtain a linear set representation of the set

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

of solutions for the homogeneous part using the function linearSet (defined in Sect. 6.1.2 to compute $\mathbf{s}_1, \ldots, \mathbf{s}_{n-1}$ such that

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

2. find one particular solution, that is, use the function partSol (defined in Sect. 6.1.3) to find a vector of terms $\mathbf{w}$ (containing the parameters $b_i$) such that $t + \sum_{j=1}^{n} \gamma_j w_j = 0$ for all values of parameters $b_i$.
3. return as the solution $\mathbf{w} + \sum_{i=1}^{n-1} \lambda_i \mathbf{s}_i$

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

### 6.1.2 Computing a linear set for a homogeneous equation

This section describes our version of the algorithm linearSet$(\gamma_1, \ldots, \gamma_n)$ that computes the set of solutions of an equation $\Sigma_{i=1}^{n} \gamma_i y_i = 0$. A related algorithm is a component of the Omega test [51]. We define

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

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

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

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

and find any solution. That is, compute

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

where partSol is given in Sect. 6.1.3.

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

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

We claim $S_H = S_L$.

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

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

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

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

Consider now a new vector $\mathbf{z} = \mathbf{y} - \lambda_1 \mathbf{s}_1$. Because $\mathbf{y} \in S_H$ and and $\mathbf{s}_1 \in S_H$ also $\mathbf{z} \in S_H$. Moreover, note that the first component of $\mathbf{z}$ is 0. We repeat the described procedure on $\mathbf{z}$

and $\mathbf{s}_2$. This way we derive the value for an integer $\alpha_2$ and a new vector that has 0 as the first two components.

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

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

### 6.1.3 Finding a particular solution of an equation

We finally describe the partSol function to find a solution (as a vector of terms) for an equation $t + \Sigma_{i=1}^n \gamma_i u_i = 0$. We use the extended Euclidean algorithm [8, Figure 31.1] that, given the integers $a_1$ and $a_2$, finds their greatest common divisor $d$ and two integers $w_1$ and $w_2$ such that $a_1 w_1 + a_2 w_2 = d$. Our algorithm generalizes the extended Euclidean algorithm to arbitrary number of variables and uses it to find a solution of an equation with parameters. We chose the algorithm presented here because of its simplicity. Other algorithms for finding a solution of an equation $t + \Sigma_{i=1}^n \gamma_i u_i = 0$ can be found in [3,15]. They also run in polynomial time. [3] additionally allows bounded inequality constraints, whereas [15] guarantees that the returned numbers are no larger than the largest of the input coefficients divided by 2.

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

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

$$
\begin{array}{l}
\mathsf{partSol}(t, \gamma_1, \gamma_2) = \\
\quad \texttt{let}\ \ (d, v_1, v_2) = \mathsf{ExtendedEuclid}(\gamma_1, \gamma_2) \\
\qquad\quad r = t/d \\
\quad \texttt{in}\ (-v_1 r, -v_2 r)
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{partSol}(t, \gamma_1, \ldots, \gamma_n) = \\
\texttt{let} \\
(w_1, w') = \mathsf{partSol}(t, \gamma_1, \gcd((\gamma_k)_{k\geq 2})) \\
(w_2, \ldots, w_n) = \mathsf{partSol}(-\gcd((\gamma_k)_{k\geq 2})w', \gamma_2, \ldots, \gamma_n) \\
\texttt{in}\ (w_1, \ldots, w_n)
\end{array}
$$

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

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

To eliminate an equation from the formula and to reduce a number of output variables, we first invoke $\mathsf{eqSyn}((x, y, z), 2a - b + 3x + 4y + 8z = 0)$. It works in two phases. In the first phase, it computes the linear set describing a set of solutions of the homogeneous equality $3x + 4y + 8z = 0$. Using the algorithm described in Sect. 6.1.2, it returns:

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

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

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

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

3. a list of fresh variables $(\lambda_1, \lambda_2)$.

We then replace each occurrence of $x$, $y$ and $z$ by the corresponding terms in the rest of the formula. This results in a new formula $7a - 3b + 13\lambda_1 \leq 4\lambda_2$. It has the same input variables, but the output variables are now $\lambda_1$ and $\lambda_2$. To find a solution for the initial problem, we let

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

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

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

### 6.2 Solving inequality constraints for synthesis

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

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

[Lower Bound] $A_i \leq \alpha_i x$
[Upper Bound] $\beta_j x \leq B_j$

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

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

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

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

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

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

Formula (2) is then equivalent to

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

We still cannot simply apply the synthesizer on that formula. Let $\{1, \ldots, J\}$ be a range of $j$ indices. The newly derived formula contains $J$ equalities and $2 \cdot J$ new variables. The process of eliminating equalities as described in Sect. 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 observe that the value of $k_j$ is always bounded: $k_j \in \{0, \ldots, L-1\}$. Thus, if the value of $k_j$ were known, we would have a formula with only $J$ new variables and $J$ additional equations. The equation elimination procedure described before would then result in a formula that has one variable less than the original starting formula, and that would guarantee termination of the approach.

Since the value of each $k_j$ variable is always bounded, there are finitely many $(J \cdot L)$ possible instantiations of $k_j$ variables. Therefore, we need to check for each instantiation of all $k_j$ variables whether it leads to a solution. As soon as a solution is found, 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. This leads to a translation schema that contains $J \cdot L$ conditional expression. In our implementation we generate this code as a loop with constant bounds.

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

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

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

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

### 6.3 Disjunctions in Presburger arithmetic

We can again lift synthesis for conjunctions to synthesis for arbitrary propositional combinations by applying the method of Sect. 4.4. We also obtain a complexity that is one exponential higher than the complexity of synthesis from the previous section. Approaches that avoid disjunctive normal form can be used in this case as well [17,44,63].

### 6.4 Optimizations used in the implementation

In this section we describe some optimizations and heuristics that we use in our implementation. Using some of them, we obtained a speedup of 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 an 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 choose 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* When processing inequalities, as described in Sect. 6.2, as soon as we introduce the modulo operator, we face a potentially longer processing time. This is because finding the suitable value of the remainder in equation $B'_j \bmod L \leq B'_j - A'_i$ requires invoking a loop. While searching for a witness, we might need to test all possible $L$ values. Therefore, we try not to introduce the modulo operator in the first place. This is possible in several 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 where we do not introduce the modulo operator is when $A'_i - B'_j$ evaluates to a number $N$ such that $N > L$. In that case, it is clear that $B'_j \bmod L \leq B'_j - A'_i$ is a valid formula and thus the returned formula is $\mathsf{true}$.

Finally, we describe an optimization that leads to a reduction in the number of loop executions. This 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 where $N$ exists, then $k_j$ also has to be a multiple of $N$. Putting this together, an equivalent formula of $B'_j \bmod L \leq B'_j - A'_i$ is the formula $T_j \bmod L_1 = k_j \wedge N \cdot k_j \leq B'_j - A'_i$. This reduces the number of loop iterations by at least a factor of $N$.

## 7 Synthesis algorithm for parameterized Presburger arithmetic

In addition to handling the case when the specification formula is an integer linear arithmetic formula of both parameters and output variables, we have generalized our synthesizer to the case when the coefficients of the output variables are not only integers, but can be any arithmetic expression over the input variables. This extension allows us to write, e.g. the offset decomposition program from Sect. 2 with statically unknown dimensions dimX, dimY, dimZ. As a slightly simpler example, consider the following invocation:

```
val (valueX, valueY) = choose((x: Int, y: Int) ⇒
  (offset == x + dim ∗ y && 0 ≤x && x <dim ))
```

Here offset and dim are input variables, whereas $x$ and $y$ are output variables. Note that dim$*y$ is not a linear term. However, at run-time we know the exact value of dim, so the term will become linear. Our synthesizer can handle such cases as well through a generalization of the algorithm in Sect. 6.

Given the problem above, we first eliminate the equality $\mathsf{offset} = x + \mathsf{dim} * y$ and we obtain the new problem consisting of two inequalities: $\mathsf{dim} * t \leq \mathsf{offset} \wedge \mathsf{offset} - \mathsf{dim} + 1 \leq \mathsf{dim} * t$. The variable $t$ is a freshly introduced integer variable and it is also the only output variable. At this point, the synthesizer needs to divide a term by the variable dim. In general it thus needs to generate code that distinguishes the cases when dim is positive, negative, or zero. In this particular example, due to the constraint $0 \leq x < \mathsf{dim}$, only one case applies. The synthesizer returns the following precondition:

$$\mathsf{pre} \equiv \lceil (\mathsf{offset} - \mathsf{dim} + 1)/\mathsf{dim} \rceil \leq \lfloor \mathsf{offset}/\mathsf{dim} \rfloor$$

It can easily be verified that this is a valid formula for all positive values of dim. The synthesizer also returns the code that computes the values for $x$ and $y$:

```
val t = (offset/dim).floor
val valueY = t
val valueX = offset − dim ∗ t
```

Our general algorithm for handling parametrized Presburger arithmetic follows the algorithm described in Sect. 6. The main difference is that instead of manipulating known integer coefficients, it manipulates arbitrary arithmetic expressions as coefficients. It therefore needs to postpone to run-time certain decisions that involve coefficients. The key observation that makes this algorithm possible is that many compile-time decisions depend not on the particular values

of the coefficients, but only on their sign (positive, negative, or zero). In the presence of a coefficient that depends on a parameter, the synthesizer therefore generates code with multiple branches that cover the different cases of the sign.

As an illustration, consider using synthesis to compute, when it exists, the positive integer ratio x between two integers a and b:

```
val x = choose((x: Int) ⇒a ∗ x == b && x ≥0)
```

In this example, the synthesizer needs to distinguish between the cases where a, which is used as a coefficient, is zero, negative, and positive: when a is zero, it computes as a precondition

$$\mathsf{pre}_0 \equiv b = 0$$

when a is negative, the precondition is

$$\mathsf{pre}_\ominus \equiv -b \geq 0 \wedge a|b$$

and similarly, when a is positive

$$\mathsf{pre}_\oplus \equiv b \geq 0 \wedge a|b$$

In fact, when the positive and negative cases differ only by a sign, our synthesized factors this out by using the expression $\frac{a}{|a|}$ for the sign of a (note that since the case where a is zero is treated before, there is no risk of a division by zero). The generated code for computing x is

```
if (a == 0 && b == 0) {
  0
} else if(−(a/Math.abs(a)) ∗ b ≥0 && b % a == 0) {
  b / a
} else {
  throw new Exception("No solution exists")
}
```

(Note that when both a and b are zero, any value for x is valid, 0 is just the option picked by the synthesizer.)

The coefficients of the invocation of the extended Euclidean algorithm generally also become known only at run-time, so the generated code invokes this algorithm as a library function. The situation is analogous for the gcd function. The following example illustrates this situation:

```
choose((x: Int) ⇒6∗x + a∗y = b
```

On this example, our synthesizer produces the following code:

```
if (b % {\ul gcd}(6,a) == 0) {
  val t1 = gcd(6,a)
  val t2 = −b / t1
  val (t3, t4) = coeffs(1, 6/t1, a/t1)
  (t2 ∗ t3, t2 ∗ t4)
} else {
  throw new Exception("No solution exists")
}
```

In this code, gcd computes the greatest common divisor, and (a,b) = coeffs(1,c,d) computes a and b such that a*c + b*d + 1 == 0 holds. Note that there are no tests on the signs of a and b, because the precondition and the code are the same in all cases (we define gcd(x,0) to be x).

Finally, note that the running time of the programs in this case is not uniform with respect to the values of all parameters. In particular, the upper bounds of the generated for loops in Sect. 6.2 can now be a function of parameters. Nevertheless, for each value of the parameter, the generated code terminates.

## 8 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. The logic we consider is BAPA. It supports the standard operators union, intersection, complement, subset, and equality. In addition, it supports the size operator on sets, as well as integer linear arithmetic constraints over these sizes. Its syntax is shown in Fig. 3. Decision procedures for BAPA were considered in a number of scenarios [18,33,36,67,68]. As in the previous sections, we consider the problem (1)

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

where the components of vectors $\mathbf{a}$, $\mathbf{x}$, $\mathbf{r}$ are either set or integer variables and $F$ is a BAPA formula.

Figure 4 describes our BAPA synthesis procedure that returns a precondition predicate pre($\mathbf{a}$) and a solved form $\mathbf{\Psi}$. The procedure is based on the quantifier elimination algorithm presented in [33], which reduces a BAPA formula to an equisatisfiable integer linear arithmetic formula. The algorithm eliminates set variables in two phases. In the first phase all set expressions are rewritten as unions of disjoint Venn regions. The second phase introduces a fresh integer variable for the cardinality of each Venn region. It thus reduces the entire formula to an integer linear arithmetic formula. The input variables in this integer arithmetic formula are the integer input variables from the original formula, as well as fresh integer variables denoting cardinalities of Venn regions of the input set variables. Note that all values of those input variables are 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.

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

**Fig. 3** A logic of sets and size constraints (BAPA)

**INPUT:** a formula $F(\mathbf{X}, \mathbf{Y}, \mathbf{k}, \mathbf{l})$ in the logic defined in Figure 3 with 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 the 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. We denote the resulting formula by $F_1(\mathbf{h_u}, \mathbf{k}, \mathbf{l})$.
4. Create a Presburger arithmetic formula that 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 $F_2(\mathbf{h_u}, |\mathbf{P_v}|)$ 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** $(\mathbf{h_u}_n, \mathbf{l_n}) =$

$$\mathsf{choose}((\mathbf{h_u}, \mathbf{l}) \Rightarrow F_1(\mathbf{h_u}, \mathbf{k}, \mathbf{l}) \wedge F_2(\mathbf{h_u}, |\mathbf{P_v}|))$$

   Invoking the synthesizer returns code that computes expressions for the integer output variables $\mathbf{l}_n$ and for the variables $\mathbf{h_u}_n$. 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 $Y_i$. Each $T_j$ region is contained in the bigger Venn region $U_j$ which is a Venn region of sets in $Y_i$. For each $T_j$ do: take all $R_u$ that belong to $T_j$ and let $d_j$ be the sum of all corresponding $h_{un}$. Based on the value of $d_j$, output the following code:
   - if $T_j \subseteq \cap_{S \in S_i} S^c$ 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: $Y_i = \cup_j K_j$

**Fig. 4** Algorithm for synthesizing a function $\Psi$ such that $F[\mathbf{x} := \Psi(\mathbf{a})]$ holds, where $F$ has the syntax of Fig. 3

We can, therefore, build a synthesizer for BAPA on top of the synthesizer for integer linear arithmetic described in Sect. 6. The integer arithmetic synthesizer outputs the precondition predicate pre and emits the code for computing values of the new output variables. The generated code can use the returned integer values to reconstruct a model for the original formula. 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 BAPA synthesis algorithm, when outputting code, we use functions `fresh` and `take`. The function `take` takes as arguments an integer $k$ and a set $S$, and returns a subset of $S$ of size $k$. The function `fresh`($k$) is invoked when $k$ fresh elements need to be generated. These functions are used only in the code that computes output values of set variables (the linear integer arithmetic synthesizer already produces the code to compute the values of integer output variables). The set-valued 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 the 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 $K_j = U_j$. It is a special case when $U_j = \cap_{S \in S_i} S^c$. If in this case it also holds that $d_j > 0$, we need to take $d_j$ elements that are not contained in any of the already known sets, i.e. we need to generate fresh $d_j$ elements. For this purpose we invoke the command `fresh`.

*Partitioning a set* We illustrate the BAPA synthesis algorithm through an example. Consider the following invocation of the choose function that generalizes the example in Sect. 2.

```
val (setA, setB) = choose((a: Set[O], b: Set[O]) ⇒
  (−maxDiff ≤a.size − b.size && a.size − b.size ≤maxDiff
    && a union b == big\−Set && a intersect b == empty
))
```

This example combines integer and set variables. Given a set `bigSet`, the goal is to divide it into two partitions. The previously defined integer variable `maxDiff` specifies the maximum amount by which the sizes of the two partitions may differ. We apply the algorithm from Fig. 4 step-by-step to illustrate how it works. After completing Step 3, we obtain the formula

$$F_1(\mathbf{h}_u) \equiv h_{100} = h_{110} = h_{010} = h_{001} = h_{111} = 0$$
$$\wedge\text{-maxDiff} \leq h_{101} - h_{011} \wedge h_{101} - h_{011} \leq \text{maxDiff}$$

We simplify the formula obtained in Step 4 using the constraints from Step 3 and obtain the formula

$$F_2(\mathbf{h}_u) \equiv |\text{bigSet}| = h_{101} + h_{011} \wedge |\text{bigSet}^c| = h_{000}$$

Now we call the linear arithmetic synthesizer on the formula $F_1(\mathbf{h}_u) \wedge F_2(\mathbf{h}_u)$. The only two variables whose values we need to find are $h_{101}$ and $h_{011}$. The synthesizer first eliminates the equation $|\text{bigSet}| = h_{101} + h_{011}$: a fresh new integer variable $k$ is introduced such that $h_{101} = k$ and $h_{011} = |\text{bigSet}| - k$. This way there is only one output variable: $k$. Variable $k$ has to be a solution of the following two inequalities: $|\text{bigSet}| - \text{maxDiff} \leq 2k \wedge 2k \leq |\text{bigSet}| + \text{maxDiff}$. This results in the precondition

$$\text{pre} \equiv \left\lceil \frac{|\text{bigSet}| - \text{maxDiff}}{2} \right\rceil \leq \left\lfloor \frac{|\text{bigSet}| + \text{maxDiff}}{2} \right\rfloor$$

Note that pre is defined entirely in terms of the input variables and can be easily checked at run-time. The synthesizer outputs the following code, which computes values for the output variables:

```
val k = ((bigSet.size + maxDiff)/2).floor
val h101 = k
val h011 = bigSet.size − k
val setA = take(h101, bigSet)
val setB = take(h011, bigSet −− setA)
```

In the code above, '`−−`' denotes the set difference operator. The synthesized code first computes the size $k$ of one of the partitions, as approximately one half of the size of `bigSet`. It then selects $k$ elements from `bigSet` to form `setA`, and selects `bigSet.size` $- k$ of the remaining elements for `setB`.

## 9 Implementation and experience

*Comfusy tool* We have implemented our synthesis procedures as a Scala compiler extension, which we call Comfusy.[1] 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 [13]. Moreover, the compiler supports plugins that work as additional compilation phases, so our extension is seamlessly integrated into compilation process (see Fig. 5). We used an off-the-shelf decision procedure [10] to handle the compile-time checks (we could, in principle, also use our synthesis procedure for compile-time checks because synthesis subsumes satisfiability checking).

Our plugin supports the synthesis of integer values through the choose function constrained by linear arithmetic predicates (including predicates in parameterized linear arithmetic), as well as the synthesis of set values constrained by predicates of the logic described in Sect. 8. Additionally, it can synthesize code for pattern-matching expressions on integers such as the ones presented in Sect. 2.

*Compilation times* Table 1 shows the compile times for a set of benchmarks, with and without our plugin. Without the plugin, the code is of no use (the choose function, when not rewritten, just throws an exception), but the difference between the timings indicates how much time is spent generating the synthesized code. We also measure how much time is used for the compile-time checks for satisfiability and uniqueness. The examples *SecondsToTime*, *FastExponentiation*, *SplitBalanced* and *Coordinates* were presented in Sect. 2. *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*. There is no measurement for *Coordinates* with compile-time checks, because the formulas to check are in an undecidable fragment, as the original formula is in parameterized linear arithmetic. 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. Moreover, note that the code we tested contained almost exclusively calls to the synthesizer. The increase in compilation time in practice would thus be lower for code that mixes standard Scala with selected choose construct invocations.

*Execution times of generated code* In our experience, the execution time of the synthesized code is similar to equivalent hand-written code. Our experience so far was restricted

[1] Our implementation source code and binaries are available from http://lara.epfl.ch/w/comfusy.

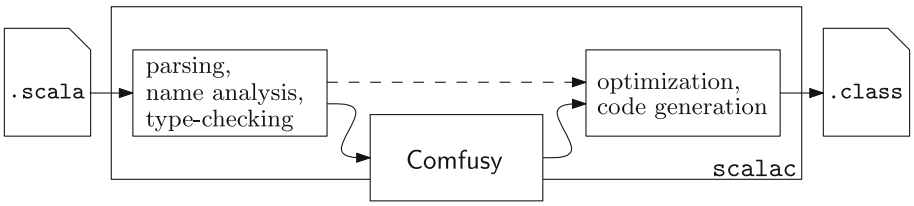


**Fig. 5** Interaction of Comfusy with `scalac`, the Scala compiler. Comfusy takes as an input the abstract syntax tree of a Scala program and rewrites calls to choose to syntax trees representing the synthesized function

**Table 1** 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)

| | 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* | 3.3 | 3.5 | 3.5 |
| *SplitBalanced* | 3.3 | 3.9 | 4.0 |
| *Coordinates* | 3.2 | 4.2 | – |
| All | 5.75 | 6.35 | 6.75 |

All times are in seconds

to small examples, not because of performance problems but rather because this is the intended way of using the tool: to synthesize code blocks as opposed to entire procedures or algorithms.

*Code size* An older version of Comfusy generated if-then-else statements that correspond to large disjunctions that appear in quantifier elimination algorithms. In certain cases, this led to formulas of large size. We have improved this by generating code that executes about as fast but uses a "for" loop instead of disjunctions. This eliminated the problems with code size and enabled synthesis for parametric coefficients, discussed above.

## 10 Related work

Early work on synthesis [42,43] 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 [55] 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. As such, it 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 focus on synthesis of program fragments with very specific control structure dictated by the nature of the decidable logical fragment.

Our work further 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 synthesized code size and running time, and (3) deployment of synthesis in well-delimited pieces of code of a general-purpose programming language.

Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [57–59]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. Search techniques have also been applied to automatically derived concurrent garbage collection algorithms [61]. In contrast, our synthesis uses the mathematical structure of a decidable theory to explore the space of all functions that satisfy the specification. This enables our approach 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 yet efficient programs. We expect that our techniques could be fruitfully integrated into search-based 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 [50] or timed systems [2]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [62]. These techniques usually take specifications in a fragment of temporal logic [49] and have resulted in tools that can synthesize useful hardware components [25,28]. 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.

Computing optimal bounds on the size and running time of the synthesized code for Presburger arithmetic is beyond the scope of this paper. Relevant results in the area of decision procedures are automata-based decision procedures [4,31], the bounds on quantifier elimination [63] and results on integer programming in fixed dimensions [14].

Automata-based decision procedures, such as those implemented in the MONA tool [32] could be used to synthesize efficient (even if large) code from expressive specifications. The work on graph types [37] proposes to synthesize fields given by definitions in monadic second-order logic. Automata have also been applied to the synthesis of efficient code for pattern-matching expressions [60].

Synthesis of constraints for rational arithmetic has been previously applied to automatically construct abstract transfer functions in abstract interpretation of linear constraints over rationals [40]. Our results apply this technique to integer linear arithmetic and constraints on sets. More generally, we observe that such synthesis is useful as a general-purpose programming construct.

Our approach can be viewed as sharing some of the goals of partial evaluation [27]. 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 with invoking the decision procedure at run time, especially when the number of variables to synthesize is bounded.

## 11 Conclusions

We have presented the general idea of turning decision procedures into synthesis procedures. We have explored in greater detail how to do this transformation for theories admitting quantifier elimination, in particular linear arithmetic. Important complexity questions arise in synthesis, such as the best possible size of synthesized code, time to perform synthesis, and the worst-case running time of the synthesized code over all inputs. We have also illustrated that synthesis procedures can be built even for cases for which the underlying parameterized satisfiability problem is undecidable (such as integer multiplication), as long as the problem becomes decidable by the time the parameters are fixed. We have also transformed a BAPA decision procedure into a synthesis procedure, illustrating in the process how to layer multiple synthesis procedures one on top of the other.

We believe that integer arithmetic and constraints on sets already make our approach interesting to programmers. The usefulness of the proposed approach can be further supported in at least two ways:

1. By developing synthesis procedures for modular (bit-vector) arithmetic, which faithfully models the machine representation of integers commonly found in programming languages. Bit-vector arithmetic by virtue of its reducibility to boolean satisfiability admits quantifier-elimination, but it is likely such a direct approach would not be the most productive one. Rather, one should look into adapting recent automata-theoretic approaches [22] or techniques for solving quantified bit-vectors formulas [64].
2. By incorporating synthesis procedures based on additional decidable constraints over data structures. For example, more control over the desired solutions for sets could be provided using decision procedures for ordered collections that we have recently identified [34]. In the example of partitioning a set, such support would allow us to specify that all elements of one partition are smaller than all elements of the second partition.

Another useful class of data structures are algebraic data types; synthesis based on algebraic data generalizes pattern matching on algebraic data types with equality and inequality constraints. The starting point for such extensions are decision procedures for algebraic data types and their extensions [7,45,53]. Our approach can also be applied to imperative data structures [37]. This idea would benefit from recent advances from more efficient decision procedures based on local theory extensions [24], including [39,65].

Given the range of logics for which we can obtain synthesis procedures, it is important to realize that we can also *combine* synthesis procedures similarly to the way in which we can combine decision procedures. We gave one example of such combination in this paper, by describing our BAPA synthesis procedure built on top of a synthesis procedure for integer arithmetic. Other combination approaches are possible building on the body of work in decision procedure combinations [19,65].

We have pointed out that synthesis can be viewed as a powerful programming language extension. Such an extension can be seamlessly introduced into popular programming languages as a new kind of expression and a new pattern matching construct. It is our hope that the availability of synthesis constructs will shift the way we think about program development. Program properties and assertions can stop being part of the dreaded "annotation overhead", but rather become a cost-effective way to build programs with the desired functionality.

## References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Tools and Algorithms for the Construction and Analysis of Systems (2008)
2. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Hybrid Systems II, pp. 1–20 (1995)

3. Banerjee, U.K.: Dependence Analysis for Supercomputing. Kluwer, Norwell (1988)
4. Boigelot, B., Jodogne, S., Wolper, P.: An effective decision procedure for linear arithmetic over the integers and reals. ACM Trans. Comput. Log. **6**(3), 614–633 (2005)
5. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Berlin (2007)
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
7. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. Electron. Notes Theor. Comput. Sci. **174**(8), 23–37 (2007)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill, Cambridge (2001)
9. Cooper, D.C.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 91–100. Edinburgh University Press, Edinburgh (1972)
10. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS (2008)
11. Dewar, R.B.K., Grand, A., Liu, S.-C., Schwartz, J.T., Schonberg, E.: Programming by refinement, as exemplified by the SETL representation sublanguage. ACM Trans. Program. Lang. Syst. (TOPLAS). **1**(1), 27–49 (1979). doi:10.1145/357062.357064
12. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Inc., Englewood Cliffs (1976)
13. Emir, B., Odersky, M., Williams, J.: Matching objects with patterns. In: ECOOP (2007)
14. Eisenbrand, F., Shmonin, G.: Parametric integer programming in fixed dimension. Math. Oper. Res. **33**(4), 839–850 (2008)
15. Ford, D., Havas, G.: A new algorithm and refined bounds for extended gcd computation. In: ANTS, pp. 145–150 (1996)
16. Flanagan, C., Leino, K.R.M., Lilibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI (2002)
17. Ferrante, J., Rackoff, C.W.: The Computational Complexity of Logical Theories. Lecture Notes in Mathematics, vol. 718. Springer, Berlin (1979)
18. Feferman, S., Vaught, R.L.: The first order properties of products of algebraic systems. Fundam. Math. **47**, 57–103 (1959)
19. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Cesare, T.: DPLL(T): fast decision procedures. In: CAV, pp. 175–188 (2004)
20. Ginsburg, S., Spanier, E.: Bounded algol-like languages. Trans. Am. Math. Soc. **113**(2), 333–368 (1964)
21. Ginsburg, S., Spanier, E.: Semigroups, Presburger formulas and languages. Pac. J. Math. **16**(2), 285–296 (1966)
22. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: FMCAD, pp. 101–109 (2010)
23. Hodges, W.: Model Theory. Encyclopedia of Mathematics and its Applications, vol. 42. Cambridge University Press, London (1993)
24. Jacobs, S.: Hierarchic decision procedures for verification. PhD thesis, Universität des Saarlandes (2010)
25. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD (2006)
26. Jones, S.P. et al.: Haskell 98 language and libraries: the revised report (2010)
27. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation (available on the Web) (1993)
28. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: a tool for property synthesis. In: CAV. LNCS, vol. 4590 (2007)
29. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Log. Program. **19/20**, 503–581 (1994)
30. Köksal, A.S., Kuncak, V., Suter, P.: Scala to the power of Z3: integrating SMT and programming. In: CADE, pp. 400–406 (2011)
31. Klaedtke, F.: On the automata size for Presburger arithmetic. Technical Report 186, Institute of Computer Science at Freiburg University (2003)
32. Klarlund, N., Møller, A.: MONA version 1.4 user manual. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus (2001)
33. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean algebra with Presburger arithmetic. J. Autom. Reason. **36**(3), 213–236 (2006)
34. Kuncak, V., Piskac, R., Suter, P.: Ordered sets in the calculus of data structures. In: CSL, pp. 34–48 (2010)
35. Kuncak, V., Piskac, R., Suter, P., Wies, T.: Building a calculus of data structures. In:VMCAI. LNCS, vol. 5944 (2010)
36. Kuncak, V., Rinard, M.: Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In: CADE-21. LNCS, vol. 4603 (2007)
37. Klarlund, N., Schwartzbach, M.I.: Graph types. In: POPL, Charleston, SC (1993)
38. Kukula, J.H., Shiple, T.R.: Building circuits from relations. In: CAV (2000)
39. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: CAV, pp. 476–490 (2005)
40. Monniaux, D.P.: Automatic modular abstractions for linear constraints. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 140–151 (2009)
41. Moskal, M.: Satisfiability modulo software. PhD thesis, University of Wrocław (2009)
42. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971)
43. Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
44. Nipkow, T.: Linear quantifier elimination. In: IJCAR (2008)
45. Oppen, D.C.: Reasoning about recursively defined data structures. In: POPL, pp. 151–157 (1978)
46. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-Step Guide. Artima Press, Walnut Creek (2008)
47. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: VMCAI. LNCS, vol. 4905 (2008)
48. Piskac, R., Kuncak, V.: Linear arithmetic with stars. In: CAV. LNCS, vol. 5123 (2008)
49. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: VMCAI (2006)
50. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL (1989)
51. Pugh, W.: A practical algorithm for exact array dependence analysis. Commun. ACM **35**(8), 102–114 (1992)
52. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, New York (1998)
53. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
54. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress, New York (2007)
55. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL (2010)
56. Sharir, M.: Some observations concerning formal differentiation of set theoretic expressions. Trans. Program. Lang. Syst. **4**(2), 196–226 (1982)
57. Solar-Lezama, A., Arnold, G., Tancau, L., Bodík, R., Saraswat, V.A., Seshia, S.A.: Sketching stencils. In: PLDI (2007)
58. Solar-Lezama, A., Jones, C.G., Bodík, R.: Sketching concurrent data structures. In: PLDI (2008)
59. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)

60. Sekar, R.C., Ramesh, R., Ramakrishnan, I.V.: Adaptive pattern matching. SIAM J. Comput. **24**, 1207–1234 (1995)
61. Vechev, M.T., Yahav, E., Bacon, D.F., Rinetzky, N.: Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In: PLDI, pp. 456–467 (2007)
62. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: TACAS (2009)
63. Weispfenning, V.: Complexity and uniformity of elimination in Presburger arithmetic. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation, pp. 48–53 (1997)
64. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: FMCAD, pp. 239–246 (2010)
65. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In: FroCoS: Frontiers in Combining Systems (2009)
66. Yessenov, K., Piskac, R., Kuncak, V.: Collections, cardinalities, and relations. In: VMCAI. LNCS, vol. 5944 (2010)
67. Zarba, C.G.: A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In: 18th International Workshop on Unification (2004)
68. Zarba, C.G.: Combining sets with cardinals. J. Autom. Reason. **34**(1), 1–29 (2005)
69. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI (2008)