Software Synthesis Procedures *

Viktor Kuncak
LARA, I&C
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland
viktor.kuncak@epfl.ch

Mikaël Mayer
LARA, I&C
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland
mikael.mayer@epfl.ch

Ruzica Piskac
LARA, I&C
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland
ruzica.piskac@epfl.ch

Philippe Suter
LARA, I&C
Swiss Federal Institute of Technology (EPFL)
Lausanne, Switzerland
philippe.suter@epfl.ch

ABSTRACT

Automated synthesis of program fragments from specifications can make programs easier to write and easier to reason about. To integrate synthesis into programming languages, software synthesis algorithms should behave in a predictable way: they should succeed for a well-defined class of specifications. We propose to systematically generalize decision procedures into *synthesis procedures*, and use them to compile implicitly specified computations embedded inside functional and imperative programs. Synthesis procedures are predictable, because they are guaranteed to find code that satisfies the specification whenever such code exists. To illustrate our method, we derive synthesis procedures by extending quantifier elimination algorithms for integer arithmetic and set data structures. We then show that an implementation of such synthesis procedures can extend a compiler to support implicit value definitions and advanced pattern matching.

1. INTRODUCTION

Synthesis of software from specifications [16, 9] promises to make programmers more productive. Despite substantial recent progress in techniques that generate short instruction sequences [12] and program fragments [22, 23], synthesis is limited to small pieces of code. We anticipate 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 may require detailed specifications, which for large programs become difficult to write.

We expect that important practical applications of synthesis lie in its integration with compilers for 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 (constraint) in its class, and checks if for some values of its variables the formula evaluates to true. 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 model-generation functionality has many uses, including better error reporting in verification and test-case generation. An important insight is that the model generation facility of decision procedures could also be used as an advanced computation mechanism. Given a set of values for some of the variables, a model-generating decision procedure can at run-time find the values of the remaining variables such that a given formula is true. Such a mechanism has the potential to bring the algorithmic improvements of decision procedures to declarative paradigms such as Constraint Logic Programming [11], which introduce search as an intrinsic aspect of program execution. Instead of changing the entire program execution platform to support search, we introduce a method to compile expressive declarative constraints while retaining the existing execution and compilation technique for the remaining parts of the program.

Our method is to transform a decision procedure into a *synthesis procedure*, which executes at compile time with a parameterized constraint as the input. The synthesis procedure generates a specialized code fragment that, at run-time, accepts the values of parameters and computes the values of variables. The computed values are guaranteed to satisfy the specified constraint. The generated 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. This approach can also give the developer static feedback, by checking the conditions under which the generated solution will exist and be unique. We use the

*The original version of this paper is entitled "Complete Functional Synthesis" and was published in (Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2010, ACM New York, NY, USA.)

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 [20]. Finally, we call our approach *complete* because it is guaranteed to work for all specification expressions from a well-defined class.

The input to a synthesis procedure is only the desired constraint, and not necessarily any bounds on values or on the structure of the synthesized code as in sketching [22] and resource-bounded synthesis [23]. This makes a synthesis procedure highly automated, but means that its implementation cannot rely only on searching an obviously finite state space; it must instead use insights from the underlying decision procedure. The use of decision procedures for particular theories also differentiates our work from the earlier work based on first-order logic [9, 12].

We demonstrate our approach by outlining synthesis algorithms for two unbounded domains: linear arithmetic and collections of objects represented as sets. We have implemented and deployed these algorithms as a compiler extension for the Scala programming language [17]. The reader can find additional details in [13]. We have found that our extension enables developers to express a number of program fragments more naturally. For example, one can state the invariants that the program should satisfy, as opposed to the computation details of establishing these invariants. In our experience, the synthesis times were acceptable and the running times were similar to equivalent hand-written code.

2. EXAMPLES

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. (The val keyword introduces a new immutable "variable" with a given value.)

```
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 a formula belonging to 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 $m \leq 60$ with $m < 60$ and $s \leq 60$ with $s < 60$, the synthesizer emits no warnings. The generated code corresponds 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 key correctness condition $h * 3600 + m * 60 + s == totsec$ will be satisfied, making program understanding and verification easier. If the developer writes the computation directly, it can be difficult for a static analyzer or a verifier to recover the key correctness condition.

Playing with our example further, suppose that 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 then 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 zero, 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)
}
```

In this example, the system uses synthesis to, e.g., compute j from the constraint $i = 2 * j + 1$. The correctness of the function follows from the observation that $\mathsf{fp}(m, b, i) = mb^i$, which we can prove by induction. For the case $i = 2 * j + 1$ 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. To support this construct, our compiler extension generates the code that selects the appropriate case and 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 (it 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 (because 42 and 5 are mutually prime), 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, a2:Set) ⇒
  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, a2:Set) ⇒
  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 such that if there is one solution, then there are many solutions. Our synthesizer resolves these choices at compile time. The resulting generated code is therefore deterministic.

3. IMPLICIT COMPUTATIONS

We next present programming language constructs that embed implicit computations into a general purpose programming language. Our constructs and algorithms are parametrized by a syntactically defined set Formulas. Formulas denote truth values, and we represent them as a well-defined subset of boolean-typed programming language expressions. Formulas are built from terms, denoted Terms, which are programming language expressions denoting values of certain types, such as integers or sets.

3.1 The "choose" Construct

As the basic form of supporting implicit computation, we integrate into a programming language a construct of the form

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

Here F is a formula (not containing choose) and $\vec{x} \;\Rightarrow\; F$ denotes the anonymous function from $\vec{x}$ to the boolean value of F (that is, $\lambda\vec{x}.F$). When we use a symbol such as $\vec{x}$ to denote a variable, we assume it could stand for a tuple of zero, one, or more actual variables. Two kinds of variables can appear within F: output variables $\vec{x}$ and (distinct from $\vec{x}$) parameters $\vec{a}$. The parameters $\vec{a}$ are program variables that are in scope at the point where choose occurs; their values will be known when the statement is executed. Output variables $\vec{x}$ denote values that need to be computed so that F becomes true, and they will be assigned to $\vec{r}$ as a result of the invocation of choose.

One of the benefits of implicit computations is that they explicitly identify the properties that the developers are allowed to assume. In particular, if $\vec{r}$ are variables distinct from $\vec{x}$ and $\vec{a}$, we can approximate the above choose invocation with the following sequence of commands in a guarded command language [5]:

assert $(\exists\vec{x}.F)$;
havoc $(\vec{r})$;
assume $(F[\vec{x} := \vec{r}])$;

Here, $F[\vec{x} := \vec{r}]$ denotes the result of simultaneously substituting variables $\vec{x}$ with the terms $\vec{r}$ in formula F, whereas havoc denotes a non-deterministic change of given variables. The simplicity of the above translation indicates that it is natural to represent choose within existing verification systems (e.g. [7, 25]). The above approximation allows different results at different invocations. A more accurate description of our implementation, which preserves determinism, can be based on Hilbert's epsilon term notation [1].

3.2 Generalized Conditionals

Our next construct enables the developer to specify how to react to the absence of solutions to a constraint. As a concrete example, consider

given $x \Rightarrow a = 2 * x + 1$ **have** $x + 10$ **else** $a + 1$

If a has value 7, then x is bound to 3 and the result is 13. If a is 10, the result is 11. The general form is the following:

given $x \Rightarrow$ F **have** T **else** E

x is a bound variable, which may appear in both F and T. The boolean expression F belongs to Formulas, whereas T and E are any programming language expressions. The construct checks if there exists a value x satisfying F, and if so, computes T with x bound to one such value. If no such value exists, it computes E (which does not refer to x). The translation to guarded commands is again straightforward.

3.3 Expressive Pattern Matching

We can use the generalized conditionals to define an advanced form of pattern matching. Consider a pattern matching expression similar in structure to the example we have seen previously.

```
i match {
  case j: 0 ⇒ m
  case j: 2∗j if j > 0 ⇒ fp(m, b∗b, j)
  case j: 2∗j+1 if j < 0 ⇒ fp(m∗b, b∗b, j)
}
```

We named the bound pattern variable j explicitly, instead of relying on syntactic conventions to infer it. Moreover, we have added guards to make the expression more interesting. We can translate the above pattern matching into:

```
given j  ⇒ i == 0 have m else
given j  ⇒ i == 2∗j && j > 0 have fp(m, b∗b, j) else
given j  ⇒ i == 2∗j+1 && j < 0 have fp(m∗b, b∗b, j) else
assert (false ) // Match failure
```

Expressions such as 2*j belong to Terms, whereas conditions such as $j > 0$ belong to Formulas. The above translation generalizes to the case of arbitrary Terms of a decidable logic used within patterns, and arbitrary Formulas in the logic as the conditions.

In the sequel we focus on the choose construct, keeping in mind that the conditional and pattern matching constructs can be synthesized similarly.

4. DERIVING SYNTHESIS PROCEDURES

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

We denote the set of variables by Vars. $\mathsf{FV}(q)$ denotes the set of free variables in a formula or a term q. If $\vec{x} = (x_1, \ldots, x_n)$ then we also use $\vec{x}$ to denote the set of variables $\{x_1, \ldots, x_n\}$. If q is a term or formula, $\vec{x} = (x_1, \ldots, x_n)$ a vector of variables and $\vec{t} = (t_1, \ldots, t_n)$ a vector of terms, then $q[\vec{x} := \vec{t}]$ denotes the result of simultaneously 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 $\sigma(x)$.

4.1 Model-Generating Decision Procedure

As a starting point for our synthesis algorithms for choose invocations we consider a model-generating decision procedure. Given $F \in$ 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,

b) or the value unsat if no such substitution exists.

We assume that the decision procedure is deterministic and behaves as a function. We write $Z(F)$=σ or $Z(F)$=unsat to denote the result of applying the decision procedure to F.

4.2 Interpretation Approach

Just like an interpreter can be considered as a baseline implementation for a compiler, as a baseline for our approach we consider deploying a model-generating decision procedure at run-time. Consider the choose statement (1). Let F_{tree} denote the syntax tree of the formula F, represented as a value within the programming language, and accepted as the input to the decision procedure Z. Similarly, let a_{name} denote the syntactic representation of the names of free parameter variables a in F.

```
F' = substitute (F_tree, a_name, a)
r = (Z(F') match {
  case σ ⇒ (σ(x_1), ..., σ(x_n))
  case unsat ⇒ assert(false) // No solution exists
}
```

The above code substitutes the known parameters by their actual values (converting the values into constants of the formula syntax tree), then invokes the decision procedure Z to obtain the values of the remaining variables. Similarly, we can replace (given x ⇒ F have T else E) with

```
F' = substitute (F_tree, a_name, a)
Z(F') match {
  case σ ⇒ Tσ      // have branch
  case unsat ⇒ E   // else branch
}
```

Here $T\sigma$ denotes evaluating the term T in the environment enriched by the returned substitution σ. Such dynamic invocation approach is flexible (because F can be computed at run-time), and immediately benefits from substantial engineering effort that was put into implementing existing decision procedures. However, there can be important performance and predictability advantages of an alternative *compilation* approach, which we explore in the rest of the paper.

4.3 Compilation: Synthesis Procedure

We consider a compilation approach where a modified decision procedure is invoked at compile time, converting the formula into a solved form.

DEFINITION 1 (SYNTHESIS PROCEDURE). *A synthesis procedure takes as the input a formula F and a vector of variables $\vec{x}$. It outputs a pair of*

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

such that the following two implications are valid:

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

We denote the fact that applying a synthesis procedure on $\vec{x}$ and F yields pre *and $\vec{\Psi}$ by writing* $(\mathsf{pre}, \vec{\Psi}) = [\![\vec{x}, F]\!]$.

Note that $F[\vec{x} := \vec{\Psi}] \to \exists \vec{x}.F$ always holds, so the above definition implies that the three formulas are all equivalent: $(\exists \vec{x}.F), \mathsf{pre}, F[\vec{x} := \vec{\Psi}]$. Consequently, if we know how to compute $\vec{\Psi}$, we can define $[\![\vec{x}, F]\!] = (F[\vec{x} := \vec{\Psi}], \vec{\Psi})$. In practice it is useful to apply simplifications to $F[\vec{x} := \vec{\Psi}]$, so we return pre explicitly.

Note that F, pre, $\vec{\Psi}$ can all refer to variables in scope at the current program point (which we denoted $\vec{a}$ in Section 3.1, but often omit for readability). The synthesizer emits the terms $\vec{\Psi}$ in compiler intermediate representation; the standard compiler then processes them along with the rest of the code. We identify the syntax tree of $\vec{\Psi}$ with its meaning as a function from the parameters $\vec{a}$ to the output variables $\vec{x}$. The overall compile-time processing of the choose statement (1) involves the following:

1. emit a non-feasibility warning if the formula ¬pre is satisfiable, reporting the counterexample for which the synthesis problem has no solution;
2. emit a non-uniqueness warning if the formula
$$F \wedge F[\vec{x} := \vec{y}] \wedge \vec{x} \neq \vec{y}$$
is satisfiable (where $\vec{y}$ are fresh variables); in such case, report the values of all free variables as a counterexample showing that there are at least two solutions;
3. as the compiled code, emit the code that behaves as
assert (pre); $\vec{r} = \vec{\Psi}$

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, without any specialization.) 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 pursues the compilation approach. We are confident that this approach has important role in implementing implicit computations. That said, we expect that there is also space for mixed interpretation-compilation solutions that explore "just-in-time synthesis" and "profiling-guided synthesis", analogously to such solutions for more conventional languages.

4.4 Quantifier Elimination versus Synthesis

The precondition computed by a synthesis procedure (pre in Definition 1) can be viewed as a result of applying quantifier elimination (see e.g. [2, Chapter 7]) to remove $\vec{x}$ from F, with the following differences.

- Synthesis procedures strengthen quantifier elimination procedures by identifying not only pre but also emitting the code $\vec{\Psi}$ that computes a *witness* for $\vec{x}$.
- Worst-case bounds on quantifier elimination algorithms measure the size of the generated formula and the time needed to generate it, but not the size of $\vec{\Psi}$ or the time to evaluate $\vec{\Psi}$. 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. Moreover, an algorithm that generates a small or simple-looking pre is not necessarily the one that generates the fastest-to-execute pre and $\vec{\Psi}$.

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

4.5 Elimination-Based Synthesis Toolbox

We next describe a basic domain-independent toolbox of techniques we found useful in converting quantifier elimination procedures into synthesis procedures. The core idea is identifying witness term functions.

DEFINITION 2. *A* witness term function *is a function*

$$\mathsf{witness} : \mathsf{Vars} \times \mathsf{Formulas} \to \mathsf{Terms}$$

such that $(\exists y.F) \to F[y := \mathsf{witness}(y, F)]$ *is a valid formula.*

Note that $\mathsf{witness}(y, F)$ may contain variables $\mathsf{FV}(F) \setminus \{y\}$.

4.5.1 Synthesis for Multiple Variables

We can lift $\mathsf{witness}(y, F)$ to synthesis for any number of variables using the following translation scheme:

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

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

The above translation has the base case, when there are no variables to eliminate, so F becomes the precondition, and the recursive case, which applies the witness function.

In an implementation we can avoid substitutions and simply use local variable definitions in the generated code and use Ψ_i instead of Ψ'_i. We generate as $\vec{\Psi}$ a code block

$$\begin{array}{l}\mathsf{val}\ x_1 = \Psi_1\\ \ldots\\ \mathsf{val}\ x_n = \Psi_n\\ \ (x_1, \ldots, x_n)\end{array}$$

where $\mathsf{FV}(\Psi_i) \subseteq \mathsf{FV}(F) \setminus \{x_i, \ldots, x_n\}$. A consequence of this recursive translation pattern is that the synthesized code computes values in reverse order compared to the steps of a quantifier elimination procedure.

4.5.2 One-Point Rule Synthesis

A widely applicable form of quantifier elimination, the one-point rule, replaces $\exists x.(x = t \wedge F)$ with $F[x := t]$, provided that $x \notin \mathsf{FV}(t)$. This rule immediately leads to a synthesis procedure step, by defining

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

If the formula does not already have the form $x = t\ \wedge\ F$, we can often rewrite it into this form using transformations that preserve equivalence, or even strengthen the formula (see "Formula Strengthening" below).

4.5.3 From Disjunctions to Conditionals

Consider a quantifier-free formula in some first-order theory where the tasks is to check formula satisfiability or apply 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 $D_1 \vee \ldots \vee D_n$ in disjunctive normal form. We can apply synthesis to each D_i yielding a precondition pre_i and the solved form $\vec{\Psi}_i$. We can then synthesize code with conditionals that select the first

$\vec{\Psi}_i$ that applies:

$$
\begin{array}{l}
[\![\vec{x}, D_1 \vee \ldots \vee D_n]\!] = \\
\quad \texttt{let}\ \ (\mathsf{pre}_1, \vec{\Psi}_1) = [\![\vec{x}, D_1]\!] \\
\qquad \ldots \\
\qquad (\mathsf{pre}_n, \vec{\Psi}_n) = [\![\vec{x}, D_n]\!] \\
\quad \texttt{in} \\
\left(\bigvee_{i=1}^{n} \mathsf{pre}_i, \left\{ \begin{array}{l} \quad \textbf{if}\ (\mathsf{pre}_1)\ \vec{\Psi}_1 \\ \textbf{else if}\ (\mathsf{pre}_2)\ \vec{\Psi}_2 \\ \ldots \\ \textbf{else if}\ (\mathsf{pre}_n)\ \vec{\Psi}_n \\ \textbf{else assert}(\textbf{false}) \end{array} \right. \right)
\end{array}
$$

Although the disjunctive normal form can be exponentially larger than the original formula, the transformation to disjunctive normal form is used in practice [21]. Other quantifier elimination methods can have better worst-case complexity [6]; these can be similarly converted into synthesis procedures. Decision tree techniques can be useful in this context. Also likely to be useful would be techniques to partially evaluate efficient constraint solving algorithms such as DPLL(T) [8].

Note that many disjunctions arising in quantifier elimination have the form of quantification over a finite set. In such cases, instead of generating a large conditional expression, it is often possible to generate a loop that searches through candidate solutions. This approach can dramatically reduce synthesized code size. We have found this technique appropriate to generate code that handles divisibility constraints in integer linear arithmetic.

4.5.4 Variable Transformations

When faced with a synthesis problem $[\![\vec{x}, F]\!]$, the variable transformation technique solves a related but simpler synthesis problem $[\![\vec{z}, G]\!]$ where $\vec{z}$ is a fresh vector of variables. The synthesized code then recovers the original values $\vec{x}$ by letting $\vec{x} = \rho(\vec{z})$ where ρ is an executable *reconstruction function.*

Note that $\vec{z}$ may have different dimension or even range over values of a different type than $\vec{x}$. We have used the variable transformation technique in a number of cases in our synthesis procedures, as Section 5 will illustrate:

- efficiently processing linear integer equations;
- representing divisibility constraints;
- reducing synthesis for sets to synthesis for integers.

In general, for *correctness* of synthesized values, we require a semantic condition corresponding to

$$G \rightarrow F[\vec{x} := \rho(\vec{z})] \tag{2}$$

That is, we require that ρ maps the values of $\vec{z}$ satisfying G to values of $\vec{x}$ satisfying F. This condition implies the validity of $(\exists \vec{z}.G) \rightarrow (\exists \vec{x}.F)$. Note that if we can express ρ as a term in our logic, then we may choose G to be identical to $F[\vec{x} := \rho(\vec{z})]$, immediately ensuring (2).

For *completeness* of synthesis, we additionally require the validity of the formula

$$(\exists \vec{x}.F) \rightarrow (\exists \vec{z}.G) \tag{3}$$

This completeness condition ensures that the new synthesis problem does not eliminate any solution.

Conditions (2) and (3) together imply that the synthesis preconditions of $[\![\vec{x}, F]\!]$ and $[\![\vec{z}, G]\!]$ are the same. Given these two conditions we apply variable transformation by defining:

$$[\![\vec{x}, F]\!] = (\mathsf{pre}, \rho(\vec{\Psi}_z)) \ \text{ where } \ (\mathsf{pre}, \vec{\Psi}_z) = [\![\vec{z}, G]\!] \tag{4}$$

Formula Strengthening.

A special case of variable transformation is *formula strengthening*, where we let ρ be the identity function. The transformation reduces to finding a formula G that entails F but where $(\exists \vec{x}.F) \rightarrow (\exists \vec{x}.G)$ is valid. Thus G may reduce the number of solutions, but not from non-zero to zero. Formula strengthening is a natural correctness condition for simple transformation of synthesis problems. It is more general than equivalence between F and G. Moreover, the even weaker condition $(\exists \vec{x}.F) \leftrightarrow (\exists \vec{x}.G)$ alone is not sufficient to guarantee that the code synthesized from G is correct with respect to F. We can use strengthening to, for example, replace a relation between variables with a stronger formula that contains a top-level equality, enabling one-point rule synthesis of Section 4.5.2, possibly after a DNF transformation of Section 4.5.3.

Pulling Out Existential Quantifiers.

Another special case is pulling out an existential quantifier. Here we assume that F is of the form $\exists \vec{y}.G$. The idea is to synthesize the value of both $\vec{x}$ and $\vec{y}$ in G and then ignore the value of $\vec{y}$. We thus let $\vec{z} = (\vec{x}, \vec{y})$ and define $\rho(\vec{x}, \vec{y}) = \vec{x}$. With this choice, both (2) and (3) are guaranteed to hold.

5. SYNTHESIS PROCEDURE FOR INTEGER ARITHMETIC AND SETS

We used the general techniques described in the previous section to design synthesis procedures for integer linear arithmetic [2, Chapter 8], as well as its extension supporting sets with cardinality constraints [14]. These techniques enabled the compilation of implicit computations such as the ones given as examples in sections 2 and 3. We next illustrate the key steps of the algorithm through a simple example, omitting many of the more subtle cases [13].

5.1 Integer Linear Arithmetic

Consider the following synthesis problem:

$$[\![(a, b, d),\ a + b = s \wedge a + 5d = 2b \wedge a \geq 0 \wedge b \geq 0 \wedge d \neq 0]\!]$$

where s is the parameter and a, b, d are the unknown variables. Rewrite the inequality $d \neq 0$ into $(d \leq -1 \vee d \geq 1)$ and transform the formula into DNF. In the sequel we illustrate synthesis for one disjunct:

$$a + b = s \ \wedge \ a + 5d = 2b \ \wedge \ a \geq 0 \wedge b \geq 0 \wedge d \geq 1$$

Variable Transformation for Linear Equations.

We first consider the integer equation $a + b = s$. After rewriting it as $a = s - b$ and eliminating a, we obtain $s - b$ as a witness term for a and, as the remaining constraint after simplification, $3b - 5d = s \ \wedge \ s - b \geq 0 \wedge b \geq 0 \wedge d \geq 1$. We then process the equation $3b - 5d = s$, where we cannot directly express one of the unknowns. We instead compute $\gcd(3, 5) = 1$ using Euclid's algorithm. In this process we find one solution for $3b - 5d = 1$, say $b = 2, d = 1$. Multiplying the last condition by s we obtain $b = 2s, d = s$ as a

solution for $3b - 5d = s$. The general solution in parametric form is then $b = 2s + 5\alpha$, $d = s + 3\alpha$, where α denotes an arbitrary integer parameter. We have thereby computed a variable transformation (Section 4.5.4) that maps α into (b, d) and preserves the set of solutions. The resulting transformed synthesis problem contains only α as the unknown and no more equations:

$$[\![\alpha,\ 5\alpha \leq -s \wedge -2s \leq 5\alpha \wedge 1 - s \leq 3\alpha]\!] \qquad (5)$$

These two equations did not produce preconditions on s. To see why a precondition can be generated, consider a different constraint, $6b - 10d = s$. Computing $\gcd(6, 10) = 2$ yields a precondition "s is even" and values $b = 2, d = 1$ that generate this gcd, with a general parametric solution $b = 2s + 10\alpha$, $d = s + 6\alpha$.

More generally, let $M\vec{x} = \vec{p}$ be a conjunction of linear equations, where $\vec{p}$ has only parameters and no variables. We can view the first part of processing equalities as finding a unimodular transformation matrix U such that $H = M \cdot U$ is the Hermite normal form of M [3]. We then use as the variable transformation $\vec{z} = \rho(\vec{x}) = U\vec{x}$ and reduce the constraint to $H\vec{z} = \vec{p} \wedge Q[\vec{x} := U\vec{x}]$. Note that $H\vec{z} = \vec{p}$ is a triangular matrix and is easier to solve than M. Solving it produces divisibility preconditions on $\vec{p}$.

Solving Inequations.

Consider the result (5) of processing equalities in our example. It remains to synthesize the value of α. We can write the constraint as a conjunction of one upper bound and two lower bounds on α: $5\alpha \leq -s$, $-2s \leq 5\alpha$, $1 - s \leq 3\alpha$. Taking into account that α is an integer, we obtain

$$\max\left(\left\lceil -\frac{2s}{5}\right\rceil, \left\lceil \frac{1-s}{3}\right\rceil\right) \leq \alpha \leq \left\lfloor -\frac{s}{5}\right\rfloor$$

where $\lceil x \rceil$ denotes x rounded towards $+\infty$, and $\lfloor x \rfloor$ denotes x rounded towards $-\infty$. As the witness value for α we can choose, for example, the upper bound and we complete the synthesis for the first disjunct. The synthesized code is

```
val alpha = − s / 5
val b = 2s + 5∗alpha
val d = s + 3∗alpha
val a = s − b
```

As usual in quantifier elimination, the precondition pre_1 is that the lower bound is less than or equal to the upper bound, i.e. $\max\left(\left\lceil -\frac{2s}{5}\right\rceil, \left\lceil \frac{1-s}{3}\right\rceil\right) \leq \left\lfloor -\frac{s}{5}\right\rfloor$. Recall from the beginning of this section that we had two disjuncts, arising from $(d \geq 1 \vee d \leq -1)$. The synthesis for the second disjunct, containing $d \leq -1$, proceeds analogously. In this case the precondition pre_2 is $\left\lceil -\frac{2s}{5}\right\rceil \leq \min\left(\left\lfloor \frac{-1-s}{3}\right\rfloor, \left\lfloor -\frac{s}{5}\right\rfloor\right)$. The overall generated code then uses an if-else statement, as in Section 4.5.3.[1]

Although the synthesis for our example stops here, all other cases, analogous to those in quantifier elimination, can arise in the general algorithm. To see this, suppose that s was not an input variable, but also needed to be synthesized. The synthesis problem would then continue with $[\![s, \mathsf{pre}_1]\!]$. We then need to represent pre_1 in linear arithmetic by eliminating division and rounding. We do so by case analysis on the reminders of s modulo the least common multiple of divisors, i.e. 15. For such case, the generated code contains a loop from 0 to 14, which can in this case be unrolled and simplified to $s = 4$ as a solution guaranteed to work.

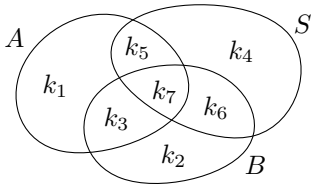
Using program specialization ideas, we also support multiplicative coefficients that are parameters instead of constants. We then emit (as opposed to statically execute) code that performs case analysis on the signs of symbolic coefficients and the appropriate computation of the least common multiple and the greatest common divisor. Note that this extension represents a departure from using witness terms alone as the generated code, because it incorporates into the synthesized code the steps of the specialized quantifier elimination procedure itself.

5.2 Sets with Cardinality Constraints

As a step towards predictable synthesis of computations on data structures, we illustrate how we can use our synthesis procedure for integers to synthesize computations on finite sets of objects. To express constraints on sets we use Boolean Algebra with Presburger Arithmetic (BAPA). This is a strict extension of integer linear arithmetic supporting sets with the usual operations $\subseteq, \cup, \cap, \setminus$ as well as the cardinality operator $|\cdot|$ computing the size of the set. A decision procedure for (quantifier-free) BAPA works by reducing the constraints over set variables to constraints over the sizes of Venn regions, which are expressible in pure linear integer arithmetic [14]. A basis of a synthesis procedure for BAPA is that a solution to these constraints can be lifted to a solution for the original problem on sets. For example, consider the synthesis problem of splitting a set S into partitions of desired sizes:

$$[\![(A, B, d),\ A \cup B = S \wedge A \cap B = \emptyset \wedge |A| + 5d = 2|B| \wedge d \neq 0]\!]$$

We start by labelling the sizes of Venn regions of the sets A, B and S by variables k_i, as displayed in the following diagram:



We rewrite our synthesis problem using these new variables:

$$\begin{aligned}
A \cup B = S &\ \rightsquigarrow\ k_1 = k_2 = k_3 = k_4 = 0 \\
A \cap B = \emptyset &\ \rightsquigarrow\ k_3 = k_7 = 0 \\
|A| + 5d = 2|B| &\ \rightsquigarrow\ \sum_{i \in \{1,3,5,7\}} k_i + 5d = 2 \cdot \sum_{i \in \{2,3,6,7\}} k_i \\
d \neq 0 &\ \rightsquigarrow\ d \neq 0
\end{aligned}$$

We additionally require $k_i \geq 0$ for all i. We apply the one-point rule for the variables equal to 0. If we compute s as $|S|$, then identify the variable a with k_5 and b with k_6, the problem reduces to our example of Section 5.1. The synthesis for integers thus yields the first part of the generated code, which computes k_5 and k_6.

It remains to show how we can reconstruct a solution for the set variables from a solution for k_5 and k_6 (in the terminology of Section 4.5.4, we need to identify ρ). For this, we rely on the existence of any computable function

[1] In this particular case, $\neg\mathsf{pre}_1$ implies $\neg\mathsf{pre}_2$, so the second case of if-else statement will never be executed—whenever we can pick a solution with a negative d, we could also pick another solution with a positive d.

$\mathsf{pickFrom}(i, T)$ that computes a subset of size i of a set T (e.g., $\mathsf{pickFrom}(i, T)$ can pick the top i elements according to some ordering [15]). We reconstruct A and B as follows:

val $A = \mathsf{pickFrom}(k_5, S)$
val $B = \mathsf{pickFrom}(k_6, S \setminus A)$

By construction, the sets from which subsets are selected are guaranteed to be of sufficient size.

5.3 Implementation

We have implemented our synthesis procedures as a Scala compiler extension called Comfusy (Complete Functional Synthesis). We used an off-the-shelf decision procedure [4] to handle the compile-time checks. We could have also used our synthesis procedure for compile-time checks because synthesis over all free variables subsumes satisfiability checking.

Our extension supports the synthesis of integer values through the choose function constrained by linear arithmetic predicates (including predicates in parametrized linear arithmetic), as well as the synthesis of set values constrained by predicates of the logic of sets with cardinality constraints. Additionally, it can synthesize code for a subset of pattern-matching expressions on integers, such as the ones presented in Section 2. Our system and examples of its use are publicly available from the web at `http://lara.epfl.ch`.

6. CONCLUSIONS

We have presented the general idea of turning decision procedures into synthesis procedures. We have explored how to do this transformation for theories admitting quantifier elimination, in particular linear arithmetic. 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.

The usefulness of the proposed approach can be further supported by incorporating synthesis procedures based on additional decidable constraints. For example, more control over the desired solutions for sets could be provided using decision procedures for ordered collections that we have recently identified [15]. 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. In addition to sets, we expect our approach to similarly apply to multisets [19]. Another relevant class are decidable constraints on algebraic data types [18, 24].

Quantifier elimination decision procedures directly support parameterized problems, so they are a particularly convenient starting point for our method. Other decision procedures are also suitable, but may require more design and implementation effort to be turned into interesting synthesis procedures. In particular, an alternative automata-theoretic approach to synthesis for integer arithmetic with bitvectors was subsequently developed [10]; this approach tends to generate larger but more efficient code.

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.

7. REFERENCES

[1] M. Abadi, G. Gonthier, and B. Werner. Choice in dynamic linking. In *FoSSaCS*, pages 12–26, 2004.

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

[3] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1993.

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

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

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

[7] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, 2002.

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[9] C. C. Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240, 1969.

[10] J. Hamza, B. Jobstmann, and V. Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, 2010.

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

[12] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28:967–989, 2006.

[13] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[14] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006.

[15] V. Kuncak, R. Piskac, and P. Suter. Ordered sets in the calculus of data structures. In *CSL*, pages 34–48, 2010.

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

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

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

[19] R. Piskac and V. Kuncak. Linear arithmetic with stars. In *CAV*, volume 5123 of *LNCS*, 2008.

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

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

1992.

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

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

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

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