Complete Program Synthesis for Linear Arithmetics

by

Mikaël Mayer

BSc., Computer Science
École Polytechnique de Paris (2008)

Submitted to the EPFL School of Computer and Communication Sciences - LARA
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

October 2010

EPFL School of Computer and Communication Sciences - LARA
January 15, 2010

Complete Program Synthesis for Linear Arithmetics

by
Mikaël Mayer

Submitted to the EPFL School of Computer and Communication Sciences - LARA
on January 15, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Program synthesis, or their fragments, is a way to write programs by providing only its meaning, without worrying about the implementation details. It avoids the drawback of writing sequential code, which might be difficult to check, error-prone or tedious.

Our contribution is to provide complete program synthesis algorithms with unbounded data types in decidable theories. We present synthesis algorithms for Linear Rational Arithmetic, Linear Integer Arithmetic and Parametrized Linear Integer Arithmetic. Our implementation and the associated Scala compiler plug-in have already been used to implement Boolean Algebra for Presburger Arithmetic synthesis.

∘ ∘ ∘

La synthèse de programmes est une manière d'écrire les programmes en spécifiant uniquement la signification du programme, sans se préoccuper des détails d'implémentation. Cela évite le désagrément d'avoir à écrire du code séquentiel, souvent difficile à vérifier, fastidieux à écrire et sujet à l'erreur.

Notre contribution est d'apporter des algorithmes de synthèse sur des types de données non bornées dans des théories décidables. Nous présentons des algorithmes de synthèse pour l'Arithmétique Linéaire Rationnelle, pour l'Arithmétique Linéaire Entière, et pour l'Arithmétique Linéaire Entière Paramétrisée. L'implémentation de notre synthétiseur et l'extension du compilateur Scala associée ont déjà été utilisées pour implémenter la synthèse de programmes sur la théorie de l'Algèbre Booléenne pour l'Arithmétique de Presburger.

∘ ∘ ∘

Thesis Supervisor: Viktor Kuncak
Title: Assistant Professor, IC, EPFL

Thesis Supervisor: Ruzica Piskac
Title: PhD Student, IC, EPFL

External Expert: Barbara Jobstmann
CNRS Researcher, VERIMAG, Grenoble, France

Acknowledgments

Special thanks to :

Viktor Kuncak, my highly experienced supervisor, also my previous advanced software analysis and verification professor, whom I consider being my best master thesis coach, for all the useful references on my subject and the help he provided me, for the continuous constructive feedback for my project and for this thesis, and much more.

Ruzica Piskac, for the help she brought me throughout my project, for supporting my thesis by writing for many chapters of it, and for her nice implementation of BAPA[1] synthesis built on my arithmetic synthesizer, etc.

Philippe Suter, for his nice Scala implementation of synthesis as a compiler plugin, for all the tips and discussions we had to make this project go up to this point, etc.

Fabien Salvi, for his highly responsive speed when I had infrastructure problems.

All the remaining people in my lab, including Hossein, Giuliano, Eva, for everything they did for me, for the discussions we had together, and for their support.

My brother Erwin and my friend Joël, for the Happy New Year 2010 project, which unexpectedly proved to be an indirect but nice synthesis application.

Barbara Jobstmann, for her great feedback for my report.

And all those who believed in the project, and/or contributed to it, whom I might have forgotten.

[1]Boolean Algebra for Presburger Arithmetic, i.e. arithmetic augmented with simple set theory

Contents

Chapter 1

Introduction

Be careful about reading health books. You may die of a misprint.

Mark Twain (1835 - 1910)

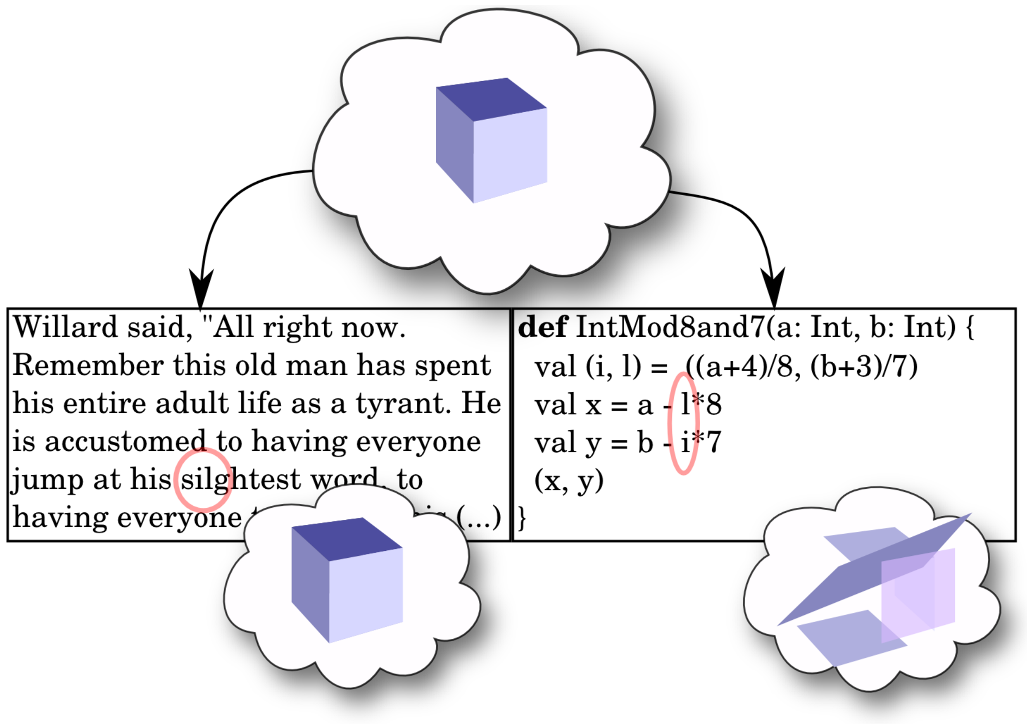
Programs and books are similar, in the sense that, depending on its author and the language style used, both are certainly more or less understood and coherent. However, although a book informally written might contain typos, a program should clearly not. Why do errors in a book not really matter? The real purpose of letters, words and sentences is to provide a meaning, an imaginary scene, a complex reasoning, much beyond the words. Even with errors, the great human capacity to grab a meaning makes this process often fruitful. Why do errors in a program really matter? The main "reader" of a program is the computer, which does not have the "global picture". For example, even though 'slightest' is misspelled in the description on the left of Fig. 1-1, it does not affect our understanding of what William said. However, the i and l inversion in the program on the right of Fig.1-1 would badly affect the behaviour of the computer, with respect to its expected meaning.

1.1 About Synthesis

Program synthesis, or their fragments, is a way to write programs by providing only its meaning, without worrying about the implementation details. It avoids the drawback of writing sequential code, which might be difficult to check, error-prone or tedious.

The goal of program synthesis is to hide low-level complexity by providing another layer of abstraction on top of the abstractions provided by the existing programming languages. Conceptually, the program would reflect its own meaning, like on the right of Figure 1-2.

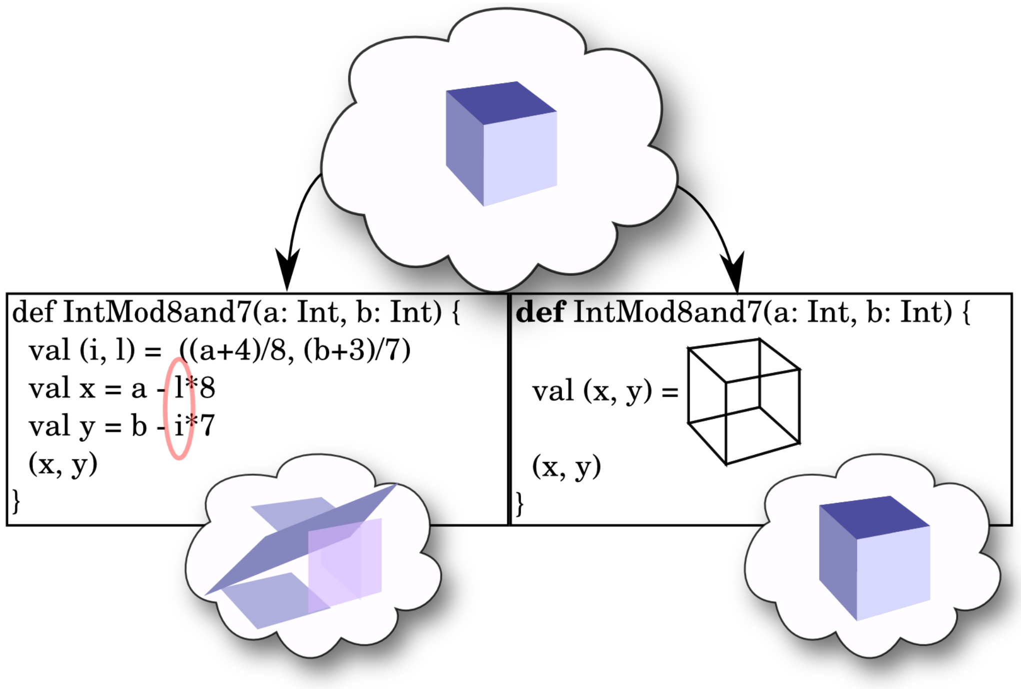
When using Hoare logic [20] for program verification, theorem provers verify that given a function f and a specification Q, the Hoare triple $\{P\}\ f\ \{Q\}$ is correct for some precondition P. If we describe synthesis with Hoare logic, it would correspond to finding, given Q, *both* a weakest precondition P and a command f such that



Willard said, "All right now. Remember this old man has spent his entire adult life as a tyrant. He is accustomed to having everyone jump at his silghtest word, to having everyone (...)

```
def IntMod8and7(a: Int, b: Int) {
  val (i, l) =  ((a+4)/8, (b+3)/7)
  val x = a - l*8
  val y = b - i*7
  (x, y)
}
```

Figure 1-1: Errors in books vs. errors in Programs



```
def IntMod8and7(a: Int, b: Int) {
  val (i, l) =  ((a+4)/8, (b+3)/7)
  val x = a - l*8
  val y = b - i*7
  (x, y)
}
```

```
def IntMod8and7(a: Int, b: Int) {

  val (x, y) =

  (x, y)
}
```

Figure 1-2: Explicit programming vs. Program Synthesis Concept

$\{P\}\ f\ \{Q\}$ is valid. For example, a solution corresponding to the Hoare triple $\{?\}\ ?\ \{2x = a + 1\}$ would be $\{2|a + 1\}\ x \leftarrow (a + 1)/2\ \{2x = a + 1\}$.

1.2 First Synthesis Steps

Let us take an example from arithmetic to understand the basics of program synthesis (see Chapter 2 for detailed explanations). Our task is to find the closest minute m to a given number of seconds T:

$$\text{“Define (m, s) such that } (T = 60 \cdot m + s,\ -30 < s \leq 30)''$$

There are two main directions in the quest for providing such high-level constructs to the programmers: run-time and compile-time resolution.

1.2.1 Run-time Resolution

One way to solve these equations is to call an external solver like Z3 [40] or CVC3 [5] to return a solution if it exists. Such command sent to a solver could look like that:

$$\textbf{val}\ (m, s) = \text{solver}(T + \text{"== 60*m + s and -30 < s and s } \leq \text{ 30"}, (\text{"m"}, \text{"s"}))$$

The approach to use a solver at run-time, has several drawbacks. First, it highly depends on the external solver, which is designed to perform much more powerful tasks and is not optimized to solve this particular problem. In some sense, this is like burning the house to fright the mouse away[1]. Second, it does not take into account that the equations are always the same and just the input values change. Third, the memory necessary to provide the solution cannot be bounded, which makes the performance of the resulting program unpredictable. Embedded systems are an example of systems, in which it is necessary to bound the amount of memory used, or even to avoid dynamic memory allocation at all. The approach to avoid dynamic memory allocation has been used by geometric libraries for robotics such as KDL [43]. In the particular class of our synthesis algorithms, mostly arithmetic synthesis, the generated programs do not have dynamic memory allocation, and thus would be suitable in general for embedded systems.

1.2.2 Compile-time Resolution - Synthesis

Let us apply our “synthesis” approach. One would write:

$$\textbf{val}\ (m, s) = \text{choose}\{(m, s) \Rightarrow T = 60 * m + s\ \textbf{\&\&}\ -30 < s\ \textbf{\&\&}\ s \leq 30\}$$

[1]French original: Tuer une mouche avec un canon

and the synthesizer would convert this code to

$$\textbf{val } m = (29 + T)/60$$
$$\textbf{val } s = T - 60 * m$$

This is the result of our synthesis procedure. Since it produces source code, the run-time of the final program will be comparable to run-time of a hand-written version, but the input code is clearer.

1.3 Contributions

This thesis makes the following contributions.

- We describe a methodology to convert decision procedures for a class of formulas into synthesis procedures that can rewrite the corresponding class of expressions into efficient executable code.
- We describe synthesis procedures for rational, integer linear arithmetic and especially for parametrized linear integer arithmetic, i.e. with coefficients unknown at compile-time. We show that, compared to invocations of constraint solvers at run-time, the synthesized code can have better worst-case complexity in the number of variables. This is because our synthesis procedure converts the given constraint (at compile time) into a solved form that can be executed while avoiding most of the search. The synthesized code is guaranteed to be correct by construction.
- We describe an approach for deploying algorithms for synthesis within programming languages. Our approach introduces a higher-order library function choose of type $(\alpha \Rightarrow \mathsf{Bool}) \Rightarrow \alpha$, which takes as an argument a function F of type $\alpha \Rightarrow \mathsf{Bool}$. Our compiler extension rewrites calls to choose into efficient code that finds a value x of type α such that $F(x)$ is true. Building on the choose primitive, we also show how to support substantially more expressive pattern matching expressions in programming languages.

1.4 Outline

The next chapters are organized as follow.

Chapter 2 (p. 17) presents concrete **examples** about how and where synthesis can be used, and section 2.2 (p. 22) describe a recent application of synthesis as a sub-program for a poem generator.

Chapter 3 (p. 29) presents the **synthesis formalism** and some common synthesis steps for the next three chapters.

Chapter 4 (p. 35) to Chapter 6 (p. 49) present the **synthesis algorithms** for Linear Rational Arithmetic, Linear Integer Arithmetic and Parametrized Linear Integer Arithmetic.

Chapter 7 (p. 57) presents the main implementation details and the performances of our synthesizer.

Chapter 8 (p. 61) presents the **related work**, among which the Boolean Algebra for Presburger Arithmetic synthesizer. Finally, Chapter 9 (p. 67) presents future enhancements for the existing synthesizer as well as for general Program Synthesis concepts.

Chapter 2

Motivating examples

Example is not the main thing in influencing others. It is the only thing.

Albert Schweitzer (1875 - 1965)

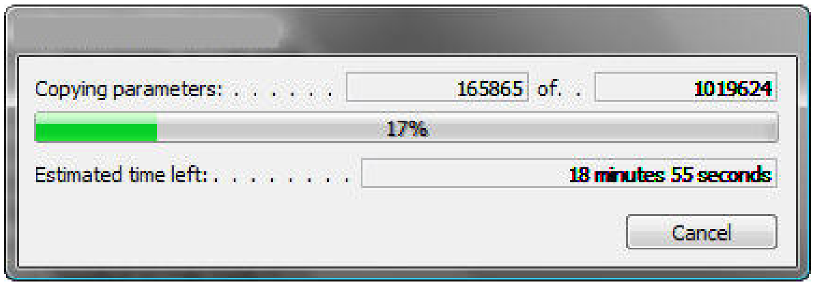
Section 2.1 contains examples of synthesis like base decomposition, advanced pattern matching, and coordinates aliasing. In Section 2.2 we explain how synthesis was used in a real unrelated public example, namely the Happy New Year Poem.

2.1 Explicit meaning vs. explicit code

In the following examples, we will present both the code that can be written to solve a problem with synthesis available, and the code our synthesizer generates. The aim is to show that programming with synthesis makes code much clearer and easier to understand than regular code, where the computation is described explicitly.

The generated code from section 2.1.1 has been found by manually applying the algorithm of Chapter 4, but the generated code from sections 2.1.2 to 2.1.5 came out of our real synthesizer.

2.1.1 Estimating time left



Copying parameters: . . . . . . 165865 of. . 1019624

17%

Estimated time left: . . . . 18 minutes 55 seconds

This is a common encountered problem in many applications. A process is going on, it is known how many data has already been processed (downloaded bytes, moved

High-level code	Generated code
val r = choose(r ⇒ p + r∗(p/t) == K)	**val** r = (K−p)/(p/t)

Table 2.1: Synthesis for elapsed time - Proportional

High-level code	Generated code
val r = choose(r ⇒ p + r∗rate == K)	**val** r = (K− p)/rate

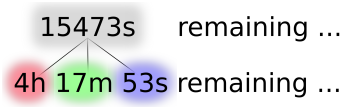
Table 2.2: Synthesis for elapsed time - Local

data, number of files already copied. . .) and how many data remains (remaining bytes, original data, number of files to copy. . .).

There are several ways to provide the user feedback about the time when the process is finished. A first way to compute the remaining time is to say that elapsed time is proportional to the amount of data processed.

Given the number of data processed p after time t, and the total amount of data K, the implicit equation defining the remaining time r is given in table 2.1. A warning would be also presented, indicating that nor p nor K should be zero. In the case it takes the current transfer instead of the ratio p/t, the solution is displayed in table 2.2. The choose method is a high-order function used to synthesize code fragments, and we explain it later in section 3.1 page 29.

2.1.2 Hours, Minutes, Seconds



Assume that we are given a remaining time in seconds and we want to display it using regular hours, minutes and seconds.

The point of this example is not the final code, which is relatively simple to write for an average programmer, but rather the way it could have been programmed with synthesis.

When someone thinks about computing hours, minutes and seconds from a given number of seconds, then one rarely starts by thinking about divisions, one would rather think about bounds and the formula defining the decomposition. This is exactly the way the program is written on the left side of Table 2.3, which is the input for the program synthesizer. The right side of table 2.3 is the code generated at compile-time. Note that the division is a floored one, otherwise we could get negative numbers for seconds and minutes. Note that the integer division `div` used here is slightly different than the usual computer division as it is floored. That is, (-3) `div` $8 = -1$ and not 0.

High-level code	Generated code

```
val (hours, minutes, seconds) =
  choose((h: Int, m: Int, s: Int) ⇒ (
      h * 3600 + m * 60 + s == totsec
   && 0 ≤ m && m < 60
   && 0 ≤ s && s < 60))
```
	```
	val (hours, minutes, seconds) = {
	  val loc1 = totsec div 3600
	  val num2 = totsec + ((−3600) * loc1)
	  val loc2 = min(num2 div 60, 59)
	  val loc3 = totsec + ((−3600) * loc1) +
	             (−60 * loc2)
	  (loc1, loc2, loc3)
	}
	```

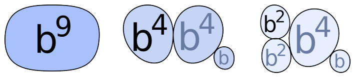
Table 2.3: Synthesis in the hour-minute-second problem

High-level code	Generated code

```
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)
}
```
	```
	def pow(base: Int, p: Int) = {
	  def fp(m: Int, b: Int, i: Int) = i match {
	    case 0 ⇒ m
	    case i if i%2 == 0 ⇒
	      val j = i/2
	      fp(m, b*b, j)
	    case i ⇒
	      val j = (i−1)/2
	      fp(m*b, b*b, j)
	  }
	  fp(1,base,p)
	}
	```

Table 2.4: Synthesis in fast exponentiation

2.1.3 Fast exponentiation



Since $b^{2i} = (b^2)^i$ and $b^{2i+1} = b * (b^2)^i$, we can compute a power of a number in logarithmic time. The general well-known algorithm looks similar to the one on the right of table 2.4. With synthesis, we can write the very clear code shown on the left of table 2.4, which compiles into the code on the right.

2.1.4 Indexing multi-dimensional arrays

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Multi-dimensional arrays are sometimes stored as a uni-dimensional array. This is actually the case in the computer memory. The drawback of using a uni-dimensional array is that accessing array elements is more complicated. In table 2.5 we simulate coordinate conversions from a 3-bytes ($0 \leq c < 3$) colored screen of size 640×480 ($0 \leq i < 640$, $0 \leq j < 480$) to a memory contains 1024-bytes blocks, where each block is indexed by x ($0 \leq x < 1024$) and the block number is indexed by y (unbounded).

High-level code	Generated code

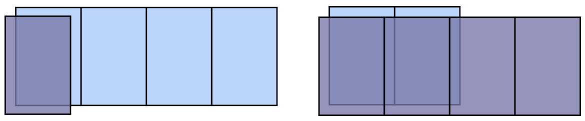
val (color, i, j) =
  choose((c: Int, i: Int, j: Int) ⇒ (
      x+1024∗y = c+3∗(i + 640∗j)
&& 0 ≤ c && c < 3
&& 0 ≤ i && i < 640
&& 0 ≤ j && j < 480
&& 0 ≤ x && x < 1024
	val (color, i, j) = {
	  val yb = Math.min(479, (x+1024∗y)/1920)
	  val ya = Math.min(639, ((x+1024∗y−1920∗yb)
	      − (3 + (x+1024∗y−1920∗yb)%3)%3)/3)
	  val j = yb
	  val i = ya
	  val c = x+1024∗y−3∗ya−1920∗yb
	  (c, i, j)
	}

val (x, y) =
  choose((x: Int, y: Int) ⇒ (
      x+1024∗y = c+3∗(i + 640∗j)
&& 0 ≤ c && c < 3
&& 0 ≤ i && i < 640
&& 0 ≤ j && j < 480
&& 0 ≤ x && x < 1024
	val (x, y) = {
	  val ya = Math.min(1023, (c+3∗i+1920∗j)/1024)
	  val y = ya
	  val x = c+3∗i+1920∗j−1024∗ya
	  (x, y)
	}

Table 2.5: Synthesis in a multi-dimensionnal conversion problem

Notice that with synthesis only one formula is needed to express the conversion for both directions (see the code on the left of table 2.5).

2.1.5 Ratio between two numbers



Given two integers b and c, we want to know the integer ratio y between them if it exists. The corresponding formula is $y * b = c \wedge y * c = b$, and the resulting generated program is displayed in table 2.6.

The *generated* precondition is the following:

$$\begin{array}{rl} & ((b = 0 \wedge c = 0) \vee (\neg(b = 0) \wedge (-c)\%|b| = 0)) \\ \vee & ((c = 0 \wedge b = 0) \vee (\neg(c = 0) \wedge (-b)\%|c| = 0)) \end{array}$$

It is interesting to note that the program is exhaustive, that is, if there is a solution, it will find it, and it is guaranteed there will not be any division by zero nor modulo by zero, if the original equation has a solution; or equivalently, if the precondition is satisfied.

This example is interesting because the original code is much more simple than the generated code, even if the latter is not fully optimized; furthermore, this example uses a linear constraint with non-constant coefficients, as it will be described in Chapter 6.

High-level code	Generated code

val y = choose(y ⇒
        y∗c == b || y∗b == c)
	val y = **if**((b == 0 && c == 0) ||
	      ((!(b == 0)) && (−c) % Math.abs(b) == 0)) {
	   val y = **if**(b == 0 && c == 0) {
	      val y = 0
	      y
	   } **else if**((!(b == 0)) && (−c) % Math.abs(b) == 0) {
	      val K1 = Math.abs(b)
	      val K0 = (−c)/K1
	      val K2 = (−1)∗(b/K1)
	      val y = K0∗K2
	      y
	   } **else** { **throw new** Error("No solution exists") }
	   y
	} **else if**((c == 0 && b == 0) ||
	         ((!(c == 0)) && (−b) % Math.abs(c) == 0)) {
	   val y = **if**(c == 0 && b == 0) {
	      val y = 0
	      y
	   } **else if**((!(c == 0)) && (−b) % Math.abs(c) == 0) {
	      val K1 = Math.abs(c)
	      val K0 = (−b)/K1
	      val K2 = (−1)∗(c/K1)
	      val y = K0∗K2
	      y
	   } **else** { **throw new** Error("No solution exists") }
	   y
	} **else** { **throw new** Error("No solution exists") }

Table 2.6: Synthesis in the ratio problem

2.2 The Happy New Year Poem

The Happy New Year Poem was a personal project in which we send our wishes in video to our contacts. The project took a lot of time, fortunately we discovered during the development that synthesis could help us.

2.2.1 Project context

My brother, a friend and I were thinking about a special surprise to send to all our Facebook contacts and to our family. The idea was the following. We would create for each one of us a video where we would sing a poem containing the names of our friends, and where the text would go from a house labelled 2009 to a house labelled 2010, with slowly falling snow, relaxing sweet music and a sun moving around (see Figure 2-1) [1].

Besides the huge challenges of the names importation, the movie soundtrack and synchronization, the 3D rendering of the 17300 pictures on several computers, the names alignment problem with a tricky change from absolute coordinates (house) to relative coordinates (camera), the recording of the voice and the automated publication on all our Facebook friends' walls, there was also the challenge of creating a correct poem with the names of our friends.

To put it in a nutshell, the specifications and relaxation points were the following.

- The input is a (huge) set of the pronounciations of our friends' names with their corresponding number of syllables. The convention used to represent the phonemes has been taken from `AJL`[44]. Example: for the two names Philippe and Philipp, we had only one entry, $\{filip, 2\}$. For Ruzica, we have the entry $\{rUjitsa, 3\}$.
- In order to be in harmony with the already composed music, each line (verse) should always contain exactly 8 syllables.
- The lines should be arranged in rhyming successions. In short, the line $2n + 1$ should rhyme with the line $2n + 2$.
- Because of the music, the lines should be arranged in a sequence of quatrains[2] and therefore, the number of lines should be divisible not only by 2 but also by 4.

$$\text{One quatrain}\left\{\begin{array}{l}\textsc{Viktor Gaëtan et Jean-Claude}\\ \textsc{Tihomir Adélaïde Maud}\\ \textsc{Marcello Belabess Lucie}\\ \textsc{Trevor Valentine Félicie}\end{array}\right. \tag{2.1}$$

[1] `http://www.youtube.com/watch?v=E2aPFdu0FNA`

[2] A quatrain is a poem in four lines.

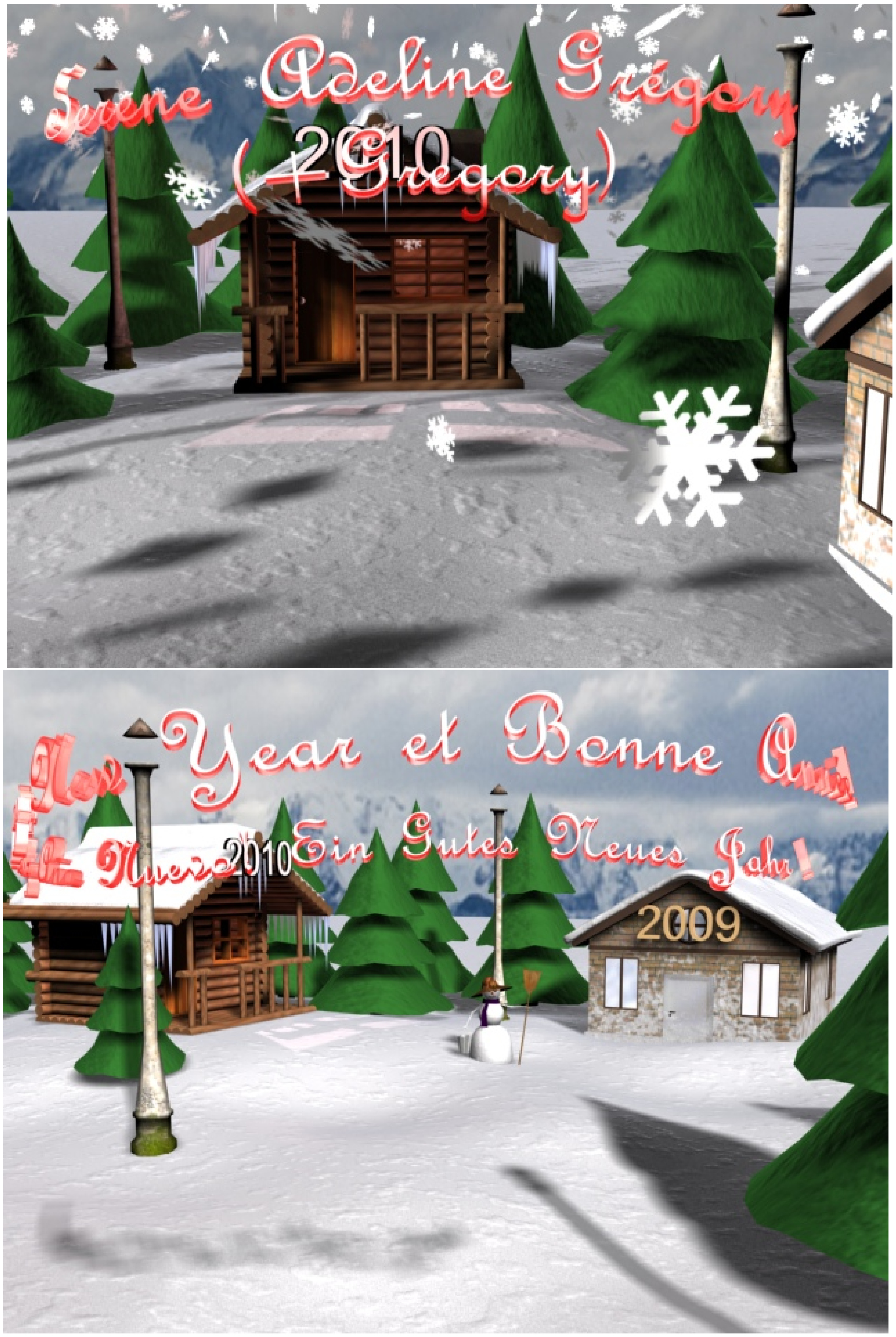


Figure 2-1: Extracts of the Happy New Year animation

- For the reasons of "family politics", specific names, e.g. names of close family members, have been placed, and there are some incomplete lines. For example, the static input of my poem was the following:

  Cynthia Mikaël et Erwin
  Cédric Arthur Sara Christine
  Laurent Grand-père et Papili
  Maman Grand-mère et Mamili

  Franck Yvann Annick et Elric
  Carole JB Thomas Eric
  Nicolas
  Papa

- A flexibility arises from the following point. There are two types of verses. Either the 8 syllables are part of names, like in "Marc Shirley Marçal Danila ". Or thanks to the french word "et" (and) which can be added, only 7 syllables are part of names, like in "Philippe Alexandre **et** Christy"

2.2.2 Arranging the poem

We will retrace the development of the program that computes some variables to determine how to globally arrange the names. For this project, we configured the synthesizer to output Python code because the main script was written in this language.

Let us consider as the first approximation a number `Nsyls` of syllables in input, and without assuming every line to be complete at the end, we wonder how many complete lines `line8` of 8 syllables it can generate, and how many syllables `remaining_syls` would remain. The corresponding code is shown in (2.2).

Using synthesis	Generated Python program
$\texttt{Nsyls} == \texttt{line8} * 8 + \texttt{remaining\_syls}$	$\texttt{tmp} = \texttt{Nsyls}/8$
$0 \leq \texttt{remaining\_syls}$	$\texttt{remaining\_syls} = \texttt{Nsyls} - 8 * \texttt{tmp}$
$\texttt{remaining\_syls} < 8$	$\texttt{line8} = \texttt{tmp}$

(2.2)

This is nice, but we don't want remaining single names at the end of the poem, this would not be interesting.

Let us now modify the code to have the flexibility to have lines of 7 syllables as well (2.3). We hide the preconditions because we expect that `Nsyls` takes large values.

Using synthesis	Generated Python program
$\texttt{Nsyls} == \texttt{line8} * 8 + \texttt{line7} * 7$	$\texttt{tmp} = (-\texttt{Nsyls})/8$
$0 \leq \texttt{line8}$	$\texttt{line8} = \texttt{Nsyls} + 7 * \texttt{tmp}$
$0 \leq \texttt{line7}$	$\texttt{line7} = -\texttt{Nsyls} - 8 * \texttt{tmp}$

(2.3)

At this point, the code on the left of (2.3) is much clearer than the code on the right. Although the code on the right is explicit in terms of programming, the left code is explicit in terms of meaning.

Now, we assume that we have existing complete lines `Elines` (see Figure 2-2 p. 27), which are not part of the input of syllables. We can introduce the total number of lines `Tlines` in (2.4).

Using synthesis	Generated Python program
$\texttt{Nsyls} = \texttt{line8} * 8 + \texttt{line7} * 7$	$\texttt{tmp} = (-\texttt{Nsyls})/8$
$\texttt{Tlines} = \texttt{Elines} + \texttt{line8} + \texttt{line7}$	$\texttt{line8} = \texttt{Nsyls} + 7 * \texttt{tmp}$
$0 \leq \texttt{line8}$	$\texttt{line7} = -\texttt{Nsyls} - 8 * \texttt{tmp}$
$0 \leq \texttt{line7}$	$\texttt{Tlines} = \texttt{Elines} - \texttt{tmp}$

(2.4)

Note that the computation of `Tlines` on the right code of (2.4) takes less additions than computing the sum of `Elines`, `line7` and `line8`. Well, let's continue the program.

We now provide the number of existing incomplete lines `Ilines`, along with the number of provided syllables `Isyls`. We ask to extract the total number of new lines `Nlines` and the number of lines created from scratch `NSLines`. Note that we apply few changes to the left code of (2.4) to obtain the left code of (2.5), while the two generated versions on the right of (2.4) and (2.5) differ quite a lot.

Using synthesis	Generated Python program
$\texttt{Nsyls} + \texttt{Isyls} = \texttt{line8} * 8 + \texttt{line7} * 7$	$\texttt{tmp} = (\texttt{Isyls} + \texttt{Nsyls})/7$
$\texttt{Tlines} = \texttt{Elines} + \texttt{Nlines}$	$\texttt{Tlines} = \texttt{Elines} + \texttt{tmp}$
$\texttt{Nlines} = \texttt{line8} + \texttt{line7}$	$\texttt{line7} = -\texttt{Isyls} - \texttt{Nsyls} + 8 * \texttt{tmp}$
$\texttt{Nlines} = \texttt{Ilines} + \texttt{NSLines}$	$\texttt{line8} = \texttt{Isyls} + \texttt{Nsyls} - 7 * \texttt{tmp}$
$0 \leq \texttt{line8}$	$\texttt{Nlines} = \texttt{tmp}$
$0 \leq \texttt{line7}$	$\texttt{NSLines} = -\texttt{Ilines} + \texttt{tmp}$

(2.5)

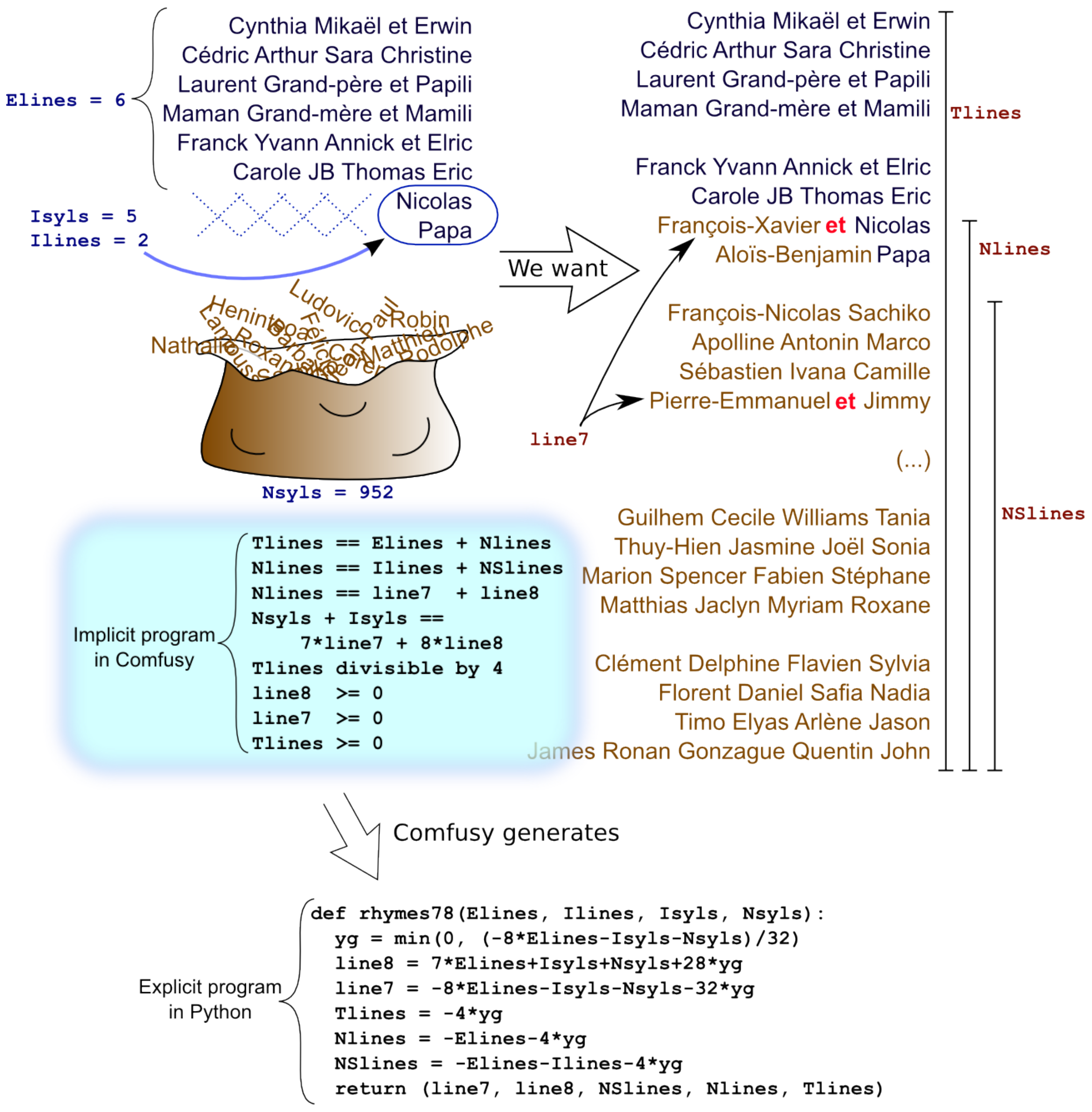
Now, we add the constraint that the number of total lines should be divisible by 4 in (2.6).

$$\begin{array}{l|l}
\text{Using synthesis} & \text{Generated Python program} \\
\hline
 & \texttt{tmp} = (-8 * \texttt{Elines} - \texttt{Isyls} - \texttt{Nsyls})/32 \\
\texttt{Nsyls} + \texttt{Isyls} = \texttt{line8} * 8 + \texttt{line7} * 7 & \texttt{Tlines} = -4 * \texttt{tmp} \\
\texttt{Tlines} = \texttt{Elines} + \texttt{Nlines} & \texttt{line7} = -8 * \texttt{Elines} - \texttt{Isyls} - \texttt{Nsyls} - 32 * \texttt{tmp} \\
\texttt{Nlines} = \texttt{line8} + \texttt{line7} & \texttt{line8} = 7 * \texttt{Elines} + \texttt{Isyls} + \texttt{Nsyls} + 28 * \texttt{tmp} \\
\texttt{Nlines} = \texttt{Ilines} + \texttt{NSLines} & \texttt{Nlines} = -\texttt{Elines} - 4 * \texttt{tmp} \\
0 \leq \texttt{line8} & \texttt{NSLines} = -\texttt{Elines} - \texttt{Ilines} - 4 * \texttt{tmp} \\
0 \leq \texttt{line7} & \\
\texttt{Tlines}\ \text{divisible\_by}\ 4 &
\end{array} \tag{2.6}$$

The right code of (2.6) is very different from the right code of (2.5) to support this new non trivial constraint.

A detailed summary of this program is available in Figure 2-2. In the closing credits of the video, the synthesizer was referred as “Comfusy”, meaning “Complete Functional Synthesis”.

As the specifications were evolving, the generated code constantly changed. Therefore we always used the generated code in the final poem generator without any manual adjustment, which would have been overwritten after a each new synthesis.


Elines = 6
Cynthia Mikaël et Erwin
Cédric Arthur Sara Christine
Laurent Grand-père et Papili
Maman Grand-mère et Mamili
Franck Yvann Annick et Elric
Carole JB Thomas Eric
Isyls = 5
Ilines = 2
Nicolas
Papa
Ludovic
Paul
Robin
Henintsoa
Nathalie
Rodolphe
Matthieu
Nsyls = 952
We want
Cynthia Mikaël et Erwin
Cédric Arthur Sara Christine
Laurent Grand-père et Papili
Maman Grand-mère et Mamili
Franck Yvann Annick et Elric
Carole JB Thomas Eric
François-Xavier et Nicolas
Aloïs-Benjamin Papa
François-Nicolas Sachiko
Apolline Antonin Marco
Sébastien Ivana Camille
Pierre-Emmanuel et Jimmy
line7
(...)
Guilhem Cecile Williams Tania
Thuy-Hien Jasmine Joël Sonia
Marion Spencer Fabien Stéphane
Matthias Jaclyn Myriam Roxane
Clément Delphine Flavien Sylvia
Florent Daniel Safia Nadia
Timo Elyas Arlène Jason
James Ronan Gonzague Quentin John
Tlines
Nlines
NSlines
Implicit program in Comfusy
Comfusy generates
Explicit program in Python

```
Tlines == Elines + Nlines
Nlines == Ilines + NSlines
Nlines == line7  + line8
Nsyls + Isyls ==
    7*line7 + 8*line8
Tlines divisible by 4
line8  >= 0
line7  >= 0
Tlines >= 0
```

```
def rhymes78(Elines, Ilines, Isyls, Nsyls):
  yg = min(0, (-8*Elines-Isyls-Nsyls)/32)
  line8 = 7*Elines+Isyls+Nsyls+28*yg
  line7 = -8*Elines-Isyls-Nsyls-32*yg
  Tlines = -4*yg
  Nlines = -Elines-4*yg
  NSlines = -Elines-Ilines-4*yg
  return (line7, line8, NSlines, Nlines, Tlines)
```

Figure 2-2: Generating the number of lines

Chapter 3

Synthesis formalism

Everything is vague to a degree you do not realize till you have tried to make it precise.

Bertrand Russell (1872 - 1970)

We are now going to explain the details on what it means to embed synthesis into an existing programming language. We implemented synthesis as a plug-in for the Scala [42] compiler, but the presented steps can be applied to any programming language.

3.1 The choose programming language construct.

We integrate into a programming language a construct of the form

$$\vec{x} = \mathsf{choose}(\vec{t} \;\Rightarrow\; F[\vec{t}, \vec{a}]) \tag{3.1}$$

Here $F[\vec{x}, \vec{a}]$[1] is a formula in a decidable logic, which has variables $\vec{x}$ and parameters $\vec{a}$. The parameters $\vec{a}$ are program variables known at the time the statement is executed, whereas $\vec{x}$ are values that need to be computed so that $F[\vec{x}, \vec{a}]$ holds[2]. We can translate the choose construct into the following sequence of commands in the guarded command languages [12]:

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

[1] $\vec{t}$ is an anonymous variable, so we prefer to use $\vec{x}$ in the remaining chapter.

[2] Notation: if F is an expression containing variables $\vec{x}$ and $\vec{a}$, then $F[\vec{x}, \vec{a}]$ is the same expression, $F[\vec{y}, \vec{b}]$ or $F[\vec{x} := \vec{y}, \vec{a} := \vec{b}]$ is the expression where the $\vec{x}$ and $\vec{y}$ have been respectively replaced by $\vec{y}$ and $\vec{x}$, and $\vec{x} \mapsto F[\vec{x}, \vec{a}]$ would be the function mapping from $\vec{x}$ to this expression, assuming that $\vec{a}$ is globally defined.

The simplicity of the translation of the choose construct also means that such construct is easier to use in verification systems such as [4, 16, 59, 60, 9] compared to the standard imperative code that would have the same effect.

3.2 Model-generating decision procedures.

As a starting point for our synthesis algorithms we consider model-generating decision procedures. We assume that a decision procedure works on a class of first-order formulas Formulas defined in terms of terms Terms. The formulas can contain free variables, and we denote $\mathsf{FV}(F)$ the set of free variables in a formula F. Given a substitution $\sigma : \mathsf{FV}(F) \to \mathsf{Terms}$, we write $F\sigma$ for the result of substituting each $x \in \mathsf{FV}(F)$ with $\sigma(x)$. Formulas are interpreted over elements of a first-order structure $\mathcal{D}$ with a countable domain D. We assume that for each $e \in D$ there exists a ground term c_e whose interpretation in $\mathcal{D}$ is e; let $C = \{c_e \mid e \in D\}$. We further assume that if $F \in \mathsf{Formulas}$ then also $F[x := c_e] \in \mathsf{Formulas}$ (the class of formulas is closed under partial grounding with constants). Given $F \in \mathsf{Formulas}$ we expect a model-generating decision procedure δ to produce either

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

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

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

3.3 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 (see also section 1.2.1 page 13). In this scenario, we replace the invocation of (3.1) with

```
F = makeFormulaTree(makeVars(x⃗), makeGroundTerms(a⃗));
r⃗ = (δ(F) match {
  case σ ⇒ (σ(x_1),...,σ(x_n))
  case unsat ⇒ throw new Exception("No solution exists")
})
```

The dynamic invocation approach is flexible and useful. It can give some advantages of constraint logic programming [23] and can also be done using e.g. the Z3 SMT solver [40] with quotations of the $F\#$ language [56]. However, there are important advantages of the compilation approach in terms of performance and predictability, as we discuss next.

3.4 Synthesis based on decision procedures.

Our goal is to explore a compilation approach where a modified decision procedure is invoked at compile time, converting the formula $F[\vec{x}, \vec{a}]$ into a "solved form" $F[f(\vec{a}), a]$. More precisely, by a synthesis procedure, we mean a procedure that takes as input a formula F with two vectors of variables $\vec{x}, \vec{a}$, and outputs:

1. a precondition predicate $\mathsf{pre}[\vec{a}]$ equivalent to $\exists \vec{x}.F[\vec{x}, \vec{a}]$.

2. a sequence of instructions $\Psi[\vec{a}]$ describing how to calculate a value of $\vec{x}$ given the values of $\vec{a}$. Mathematically speaking, $f : \vec{a} \mapsto \Psi[\vec{a}]$ defines a (maybe partial) function such that for each $\vec{a}$, if $\mathsf{pre}[\vec{a}]$ then $f(\vec{a})$ is defined and $F[f(\vec{a}), \vec{a}]$ holds.

The sequence of instructions $\Psi[\vec{a}]$ is emitted in compiler intermediate representation and compiled with the rest of the code. Furthermore, when we compile the choose statement (3.1), we added the following compilation steps:

- emit a non-feasibility warning if the formula $\neg\mathsf{pre}[\vec{a}]$ is satisfiable, reporting the model for $\vec{a}$ as a counterexample for which the given constraint has no solution;

- emit a non-uniqueness warning if the formula

$$F[\vec{x}, \vec{a}] \wedge F[\vec{y}, \vec{a}] \wedge \vec{x} \neq \vec{y}$$

  is satisfiable, reporting the model for $\vec{a}$, $\vec{x}$ and $\vec{y}$ as a counterexample showing that for the values $\vec{a}$, there are at least two solutions;

- as the compiled code, emit the code that behaves as

```
assert(pre[$\vec{a}$])
$\vec{x}$ = $\Psi[\vec{a}]$
```

Among the advantages of the compilation approach — see also section 1.2.2 page 13 — are:

- improved run-time efficiency: 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.

3.5 Efficiency of synthesis.

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

- time to synthesize the code, as a function of F;

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

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

3.6 From quantifier elimination to synthesis

$$\begin{tikzcd}
\text{Quantifier elimination} & \exists x.F[\vec{x},\vec{a}] \arrow[r] \arrow[d, Rightarrow] & \mathsf{pre}[\vec{a}] \arrow[d, Rightarrow] \\
\text{Synthesis} & \exists x.F[\vec{x},\vec{a}] \arrow[r] & \text{“}F[\Psi[\vec{a}],\vec{a}]\text{”}
\end{tikzcd}$$

The precondition pre can be viewed as a result of applying quantifier elimination (see e.g. [41]) to remove $\vec{x}$ from F. Synthesis procedures strengthen quantifier elimination procedures by identifying not only pre but also emitting the code $\Psi[\vec{a}]$ that efficiently computes a witness for $\vec{x}$. On the other hand, quantifier elimination is typically applied to arbitrary quantified formulas of first-order logic by successively eliminating quantifiers, so pre is in the same fragment of formulas as F; this condition is not required in our case (we do expect the language of pre to have a decision procedure for the purpose of warnings). Naturally, the results on quantifier elimination measure the size of the generated formula and the time needed to generate it, but not the size of $\Psi[\vec{a}]$ or the execution time for $\Psi[\vec{a}]$.

Despite the differences, we have found that we can naturally extend existing quantifier elimination procedures with explicit computation of witnesses that constitute the program $\Psi[\vec{a}]$; the general idea of eliminating variables one by one is as follows. Given (3.1), we consider the formula $\exists x_1, \ldots, x_n.F$. Suppose we have a quantifier elimination procedure that eliminates x_n from $\exists x_n.F$. This results in a formula F_1 possibly containing $x_1, \ldots, x_{n-1}, \vec{a}$. We extend the quantifier elimination to a synthesizer in a way described above and the result is the precondition predicate pre_1 (which is actually the formula $F_1[x_1, \ldots, x_{n-1}, \vec{a}]$) and the sequence of commands $\Psi_1[x_1, \ldots, x_{n-1}, \vec{a}]$ where the value of a witness for x_n is computed. We proceed with the quantifier elimination of the next output variable x_{n-1} from the formula $F_1[x_1, \ldots, x_{n-1}, \vec{a}]$ and the resulting code Ψ_2 we concatenate with Ψ_1. We repeat this procedure until all output variables are eliminated.

We formalize that as a translation scheme $[\![F, \vec{a}, \vec{x}, \Gamma]\!]$. It takes four arguments: a formula F, a list of input variables $\vec{a}$, a list of output variables $\vec{x}$ and a list of code Γ and returns a pair (pre, Ψ), where pre is a precondition predicate that takes $\vec{a}$ as an argument and Ψ is a list of commands that effectively computes values of witnesses. With ::: we denote the concatenation of two lists. Here is a formal definition of the translations scheme which also handles multiple number of output variables:

$$\begin{aligned}
&[\![F, \vec{a}, \emptyset, \Gamma]\!] = (F, \Gamma)\\
&[\![F, \vec{a}, x, \Gamma]\!] = (\mathsf{pre}[\vec{a}], \Psi[\vec{a}] ::: \Gamma)\\
&[\![F, \vec{a}, (x_1, \ldots, x_{x-1}, x_n), \Gamma]\!] = [\![\mathsf{pre}_1, \vec{a}, (x_1, \ldots, x_{x-1}), \Psi_1]\!]\,,\\
&\qquad \text{where } (\mathsf{pre}_1, \Psi_1) = [\![F, \vec{a} ::: (x_1, \ldots, x_{x-1}), x_n, \Gamma]\!]
\end{aligned}$$

Having this scheme, we are able to express one simple optimization which we will use whenever applicable, and thus we do not mention it explicitly in the rest of the paper. This optimization is used when a formula F is a conjunction of two formulas $F_1(\vec{a})$, which contains only input variables, and $F_2(\vec{a}, \vec{x})$. In that case we add $F_1(\vec{a})$ to pre and continue to synthesize only $F_2(\vec{a}, \vec{x})$.

$$\begin{gathered}
[\![F_1(\vec{a}) \wedge F_2(\vec{a}, \vec{x}), \vec{a}, \vec{x}, \Gamma]\!] = (\mathsf{pre}[\vec{a}] \wedge F_1(\vec{a}), \Psi[\vec{a}]),\\
\text{where } (\mathsf{pre}[\vec{a}], \Psi[\vec{a}]) = [\![F_2(\vec{a}, \vec{x}), \vec{a}, \vec{x}, \Gamma]\!]
\end{gathered}$$

3.7 Propositional Operations in Synthesis

The main algorithms presented in the next chapters are dealing with conjunctions of literals. We now present we deal with arbitrary formula with disjunctions.

3.7.1 Synthesis for Propositional Logic

In order to illustrate the potential gain of precomputation, first consider the following simple approach when F is a propositional formula (see e.g. [31] for a more sophisticated approach). Build an ordered binary decision diagram (OBDD) [7] for F, treating both $\vec{a}$ and $\vec{x}$ as variables, using an ordering that puts all parameters $\vec{a}$ before the variables $\vec{x}$. Then split the OBDD graph at the point where all the decisions on $\vec{a}$ have been made. For each of the OBDD nodes in this slice, we can precompute whether it reaches the true node. We emit the code that consists of nested if-then-else tests encoding the upper slice of the OBDD, followed by the code that, for nodes that reach true emits one path that reaches true. Although the size of the code can be singly exponential, the code executes in time linear in the dimension of $\vec{a}$ and $\vec{x}$. This is in contrast to NP-hardness of finding a satisfying assignment for a propositional formula F, which would occur in the baseline approach of invoking a SAT solver at run-time. In summary, for propositional synthesis we can precompute solutions to an NP-hard problem and generate code that computes unknown propositional values in polynomial time.

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

3.7.2 Propositional Connectives in First-Order Theories

Consider quantifier-free formulas in some first-order theory. To check satisfiability or apply quantifier elimination for arbitrary propositional combinations of literals, we can transform the formula to disjunctive normal form and process each disjunct independently. We can also adopt this method to synthesis. Let $D_1, \ldots, D_n$ be the disjuncts in disjunctive normal form of a formula, with variables $\vec{x}$ and parameters $\vec{a}$. Apply synthesis to each D_i yielding a precondition $\mathsf{pre}_i(\vec{a})$ and the solved form $\Psi_i(\vec{a})$. The final translation schema can be explicited as follow:

$$
\begin{aligned}
&[\![D_1 \vee \ldots \vee D_n, \vec{a}, \vec{x}, \Gamma]\!] = \\
&(\bigvee_{i=1}^{n} \mathsf{pre}_i(\vec{a}), \left\{ \begin{array}{ll}
\textbf{if } (\mathsf{pre}_1(\vec{a})) & \Psi_1(\vec{a}) \\
\textbf{else if } (\mathsf{pre}_2(\vec{a})) & \Psi_2(\vec{a}) \\
 & \vdots \\
\textbf{else if } (\mathsf{pre}_n(\vec{a})) & \Psi_n(\vec{a}) \\
\textbf{else} & \\
\ \textbf{throw new } \text{Exception(``No solution exists'')} &
\end{array} \right\}), \\
&\quad \text{where} \\
&\qquad (\mathsf{pre}_1(\vec{a}), \Psi_1(\vec{a})) = [\![D_1, \vec{a}, \vec{x}, \Gamma]\!] \\
&\qquad \ldots \\
&\qquad (\mathsf{pre}_n(\vec{a}), \Psi_n(\vec{a})) = [\![D_n, \vec{a}, \vec{x}, \Gamma]\!]
\end{aligned}
$$

While the disjunctive normal form can be exponentially larger than the original formula, the transformation to disjunctive normal form is used in practice [48] and has advantages in terms of the quality of synthesized code generated for individual disjuncts. What further justifies this approach is that we expect a small number of disjuncts in our specifications, and expect to need different synthesized values for variables in different disjuncts. Other methods can have better worst-case quantifier elimination complexity [10, 15, 58, 41].

Our solver is slightly more optimized than that. We first try to solve top-level equalities before starting to split the formula to disjunctive normal form. For example, in $x + y = 4 \wedge (x > y \vee x < y)$ we first solve $x + y = 4$ and replace the expression in the subsequent equations, yielding the new problem $4 > 2y \vee 4 < 2y$.

Chapter 4

Linear Rational Synthesis

Don't reinvent the wheel, just realign it

Anthony J. D'Angelo

In spite of its relative simplicity, the algorithm to define linear rational synthesis is a good start to understand the principles of general synthesis algorithms. Although we did not implement it, we entirely designed it and computed its complexity.

4.1 Synthesis for Linear Rational Arithmetic

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

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

4.1.1 Solving Conjunctions of Literals

Given the observations in Section 3.7.2, we consider conjunctions of literals. The method follows Fourier-Motzkin elimination [49]. Consider the elimination of a variable x.

Equalities

If x occurs in an equality constraint $t = 0$, then rewrite the constraint as $x = t'$. We eliminate x from consideration, and continue recursively with the remaining variables. The synthesized code for computing x is the term t'. This step is Gaussian elimination, and we use it whenever it is applicable. We therefore eliminate first those variables that occur in some equalities.

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 and b_p. Consider the more general case when there is both at least one lower bound a_p and at least one upper bound b_q. We then continue recursively with the formula $\bigwedge_{p,q} a_p < b_q$ and generate code for it. To compute x_n, we take the mean between the max of lower bounds and min of upper bounds.

The corresponding translation scheme is the following:

$$\begin{aligned}
&\left[\!\!\left[F[\vec{a},\vec{y}] \wedge (\textstyle\bigwedge_p a_p < x) \wedge (\bigwedge_q x < b_q), \vec{a}, \vec{y} ::: (x), \Gamma \right]\!\!\right] = \\
&\qquad (\mathsf{pre}[\vec{a}], \Psi[\vec{a}] ::: \left\{ \begin{array}{l} \textbf{val } b = \min_q\{b_q\} \\ \textbf{val } a = \max_p\{a_p\} \\ \textbf{val } x = (a+b)/2 \end{array} \right\} ::: \Gamma) \\
&\text{where} \\
&\qquad (\mathsf{pre}[\vec{a}], \Psi[\vec{a}]) = \left[\!\!\left[F[\vec{a},\vec{y}] \wedge \textstyle\bigwedge_{p,q} a_p < b_q, \vec{a}, \vec{y}, \Gamma \right]\!\!\right]
\end{aligned}$$

In case there are no lower bounds, we can compute **val** $x = b - 1$; if there are no upper bounds, we compute **val** $x = a + 1$.

Complexity of synthesis for conjunctions.

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

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

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

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

The size of the generated program is bounded by:

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

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

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

4.1.2 Disjunctions in Linear Rational Arithmetic

One way to lift synthesis for rational arithmetic from conjunctions of literals to arbitrary propositional combinations is to apply the method of Section 3.7.2. We then obtain complexity that is one exponential higher in formula size than the complexity of synthesis for conjunctions.

Chapter 5

Linear Integer Synthesis

When you can't have what you want, it's time to start wanting what you have.

Kathleen A. Sutton

We next consider synthesis for quantifier-free formulas of Presburger arithmetic [47] (integer linear arithmetic). In this theory variables range over integers. Terms are linear expressions of the form $c_0 + c_1 x_1 + \ldots + c_n x_n$, $n \geq 0$, c_i is an integer constant and x_i is an integer variable. Atoms are build using relations $\geq$, $=$ and $|$. The atom $c|t$ is interpreted as true iff an integer constant c divides term t. We also sometimes use the predicate $<$ as shorthand for $a < b$ iff $a \leq b \wedge \neg(a = b)$.

Given a quantifier-free formula F, we assume that F is in disjunctive normal form. We will describe a synthesis algorithm which works only for a formula which is a conjunction of literals. This algorithm can be further extended to be a synthesis algorithm for F as described in Section 3.7.2.

Pre-processing.

Before we can apply the algorithm, we need to do some pre-processing steps and eliminate negations and divisibility constraints. We remove negations by transforming a formula into its negation-normal form and then we eliminate negative literals by translating them into equivalent positive one: $\neg(t_1 \geq t_2)$ 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 $t \geq 0$.

Divisibility constraints of a form $K|t$ are transformed into equalities while adding a fresh variable. The obtained value of the fresh variable y is ignored in the final synthesized program.

$$\begin{gathered}[\![(K|t) \wedge F, \vec{a}, \vec{x}, \Gamma]\!] = (\mathsf{pre}_1, \Psi[\vec{a}] ::: \Gamma), \text{ where} \\ (\mathsf{pre}_1, \Psi[\vec{a}]) = [\![t = Ky \wedge F, \vec{a}, \vec{x} ::: (y), \emptyset]\!]\end{gathered}$$

$\Psi[\vec{a}]$ is a sequence of commands for computing the values of $\vec{x}$.

The negation of divisibility $\neg(K|t)$ can be handled in a similar way by introducing two fresh variables l_1 and l_2 and replacing the divisibility constraint with

$$Kl_1 = t + l_2 \wedge 0 < l_2 < K$$

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

5.1 Equality Constraints

We next present a procedure which eliminates equality constraints from a formula F. For this we will use the algorithm `eqSyn` described in Section 5.1.1. It takes as an input an equality[1] $E[\vec{b}, \vec{y}] \equiv \Sigma_{i=1}^{m} \beta_i b_i + \Sigma_{j=1}^{n} \gamma_j y_j = 0$, and returns

- an integer δ such that $\delta = \gcd_j(\gamma_j)$.
- an integer linear arithmetic formula $\mathsf{pre}_E[\vec{b}]$ describing what is required for $E[\vec{b}, \vec{y}]$ to have a solution.
- a fresh input variable d such that it should be assumed that $\delta d = \Sigma_{i=1}^{m} \beta_i b_i$
- a list of linear terms t_i to describe the solutions for y_j.

The returned terms t_i are such that the equality $E[\vec{b}, y_1 := t_1, \ldots, y_n := t_n] \equiv \delta d + \Sigma_{j=1}^{n} \gamma_j t_j$ evaluates to true. Each term t_i is a linear combination of new fresh output variables z_i and the input fresh variable d. Important is that the number of z_i variables is strictly smaller than the number of y_i variables. This way we will reduce the number of output variables for at least one. In the rest of the formula F we replace each occurrence of y_i with corresponding term t_i and we proceed with synthesis of the rest of the formula F.

$$\left[\!\!\left[E[\vec{b}, \vec{y}] \wedge F, \vec{a}, \vec{x}, \Gamma \right]\!\!\right] =$$

$$(\mathsf{pre}[\vec{a}] \wedge \mathsf{pre}_E[\vec{b}], \left\{ \textbf{ val } d = \tfrac{1}{\delta} \Sigma_{i=1}^{m} \beta_i b_i \right\} ::: \Psi[\vec{a}, d] ::: \left\{ \begin{array}{l} \textbf{val } y_1 = t_1 \\ \vdots \\ \textbf{val } y_n = t_n \end{array} \right\} ::: \Gamma), \text{ where}$$

$$(\mathsf{pre}_E[\vec{b}], d, \delta, \vec{t}) = \mathsf{eqSyn}(E[\vec{b}, \vec{y}]) \text{ and}$$
$$(\mathsf{pre}[\vec{a}], \Psi[\vec{a}]) = [\![F[y_1 := t_1, \ldots, y_n := t_n], \vec{a} ::: (d), (\vec{x} \backslash \vec{y}) ::: \vec{z}, ()]\!]$$

With $\vec{x} \backslash \vec{y}$ we denote here a vector of variables $\vec{x}$ without $\vec{y}$ variables. For example, $(x_1, x_2, x_3, x_4) \backslash (x_2, x_3) = (x_1, x_4)$.

[1] We assume that $\vec{b} \subseteq \vec{a}$ and $\vec{y} \subseteq \vec{x}$

5.1.1 Reducing the Number of Output Variables

In this section we describe the algorithm `eqSyn`. Let $\Sigma_{i=1}^{m}\beta_i b_i + \Sigma_{j=1}^{n}\gamma_j y_j = 0$ be an equality. By Bézout's theorem, this equality has a solution iff $\gcd_j(\gamma_j) | \Sigma_{i=1}^{m}\beta_i b_i$. Therefore, we define $\delta = \gcd_j(\gamma_j)$ and $\mathsf{pre}_E[\vec{b}] \equiv \delta | \Sigma_{i=1}^{m}\beta_i b_i$. As d has an implicit definition, eqSyn only has to generate a new name for it, so there remains the generation of the terms t_j.

First we consider the case when there is only one output variable in the equality. In that case the algorithm `eqSyn` returns the following result, in which we quickly verify that $E[\vec{b}, y := -d] \equiv \gamma d - \gamma d == 0$ evaluates to `true`.

$$\texttt{eqSyn}(\Sigma_{i=1}^{m}\beta_i b_i + \gamma y = 0) = \begin{cases} \delta & \mapsto \quad \gamma \\ \mathsf{pre}_E[\vec{b}] & \mapsto \quad \delta | \Sigma_{i=1}^{m}\beta_i b_i \\ d & \mapsto \quad \left[\begin{array}{l} \text{A new fresh input variable where} \\ \text{it is needed that } \delta d = \Sigma_{i=1}^{m}\beta_i b_i \end{array} \right. \\ t & \mapsto \quad -d \end{cases}$$

From now on we assume that there is more than one output variable in the equality. Out goal is to derive an alternative definition of the set $K = \{\vec{y} \mid \Sigma_{i=1}^{m}\beta_i b_i + \Sigma_{j=1}^{n}\gamma_j y_j = 0\}$ which will allow a simple and effective computation of elements in K. Note that the set K describes the set of all solutions of a Presburger arithmetic formula and following [17, 18] there is a semilinear set describing it. A *semilinear set* is finite union of linear sets. Given an integer vector $\vec{b}$ and a finite set of integer vectors S, a *linear set* is a set $\{\vec{x} \mid \vec{x} = \vec{b} + \vec{s}_1 + \ldots + \vec{s}_n; s_i \in S; n \geq 0\}$. Vector $\vec{b}$ is called a base vector while vectors in S are called step vectors. Every semilinear set is a solution of some Presburger arithmetic formula. Ginsburg and Spanier showed that converse holds as well: the set of all solutions of a Presburger arithmetic formula can be described with a semilinear set. However, we cannot apply this result immediately because there are also input variables whose values are not known until the execution time. We overcome this problem by introducing witnesses. We now explain in details three steps in defining a set describing set K.

Given the equality $\Sigma_{i=1}^{m}\beta_i b_i + \Sigma_{j=1}^{n}\gamma_j y_j = 0$ in the first step we define the set $S_H = \{\vec{y} \mid \Sigma_{j=1}^{n}\gamma_j y_j = 0\}$ which describes a solution set of a homogeneous equality. This is a linear set and it has a form $\{\vec{y} \mid \vec{y} = \alpha_1\vec{s}_1 + \ldots + \alpha_k\vec{s}_k; \alpha_i \in \mathbb{Z}\}$. Vectors $\vec{s}_i$ are known and their effective computation is described in Section 5.1.2. Important is that the number of s_i vectors is strictly smaller than n.

In the second step we compute a witness vector $\vec{w}$. For this we use generalization of Bézout's identity: for any numbers $k_1, \ldots, k_n$ with greatest common divisor d there exist integers $\alpha_1, \ldots, \alpha_n$ such that $\alpha_1 k_1 + \cdots + \alpha_n k_n = d$. A fast algorithm for computing those integers is described in Section 5.1.3.

Recall that $\delta = \gcd_j(\gamma_j)$ and let us define $I = \Sigma_{i=1}^{m}\beta_i b_i$. Let $J = I/\delta$. We apply Bézout's identity on numbers $\gamma_1, \ldots, \gamma_n$ and compute numbers $v_1, \ldots, v_n$ such that $v_1\gamma_1 + \cdots + v_n\gamma_n = \delta$. Multiplying this equality with J results in $v_1 J\gamma_1 + \cdots + v_n J\gamma_n = \delta J = I$. We define $w_i = -v_i J$ and form vector $\vec{w}$. Therefore $\Sigma_{i=1}^{m}\beta_i b_i + w_1\gamma_1 + \cdots + w_n\gamma_n = \Sigma_{i=1}^{m}\beta_i b_i - I = 0$, so the vector $\vec{w}$ belongs to the set K.

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

To conclude, the algorithm `eqSyn` returns two things: the first is precondition $\delta | \Sigma_{i=1}^{m} \beta_i b_i$ and the second is list of terms t_i. Using the computed values for generators of set S_H and a witness $\vec{w}$, terms t_i are computed as: $t_i = w_i + \lambda_1 s_{1i} + \ldots + \lambda_k s_{ki}$.

5.1.2 Efficient Computation of Linear Sets

To complete handling of equalities in our linear integer arithmetic synthesizer, the last hurdle we need to address is an efficient computation of a set describing the set of solutions of an equation $\Sigma_{i=1}^{n} \gamma_i y_i = 0$. Following the Omega test [48], we know the structure of this set. It is a linear set with $\vec{0}$ as the base vector and at most $n-1$ step vectors: $\{\alpha_1 \vec{s}_1 + \ldots + \alpha_{n-1} \vec{s}_{n-1} \mid \alpha_i \in \mathbb{Z}\}$. The Omega test is an algorithm which describes, among others, a computation of those step vectors. However, we find it too complex for our purposes, so here we propose direct computation of those step vectors without applying the Omega test.

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

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

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

- if $i < j$, $K_{ij} = 0$
- $K_{jj} = \frac{\gcd((\gamma_k)_{k \geq j+1})}{\gcd((\gamma_k)_{k \geq j})}$
- remaining values K_{ij} are computed as follows: for each index j, $1 \leq j \leq n-1$, consider the equation
$$\gamma_j K_{jj} + \sum_{i=j+1}^{n} \gamma_i u_{ij} = 0$$
and find any solution. Let k_{ij} be a value of a variable u_{ij} in the found solution. For all the remaining K_{ij} for this fixed j, output $K_{ij} = k_{ij}$. In Section 5.1.3 we describe how to find a solution using only the Euclidean algorithm.

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

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

To prove that the converse also holds, we show that a vector $\vec{x} \in S$ can be written as a linear combination of $\vec{s}_j$ vectors. Let $G_1 = \gcd((\gamma_k)_{k\geq 1})$. It follows that:

$$\begin{array}{rl} & \vec{x} \in S \\ \Leftrightarrow & \Sigma_{i=1}^{n}\gamma_i x_i = 0 \\ \Leftrightarrow & G_1 \cdot (\Sigma_{i=1}^{n}\beta_i x_i) = 0 \end{array}$$

where $\beta_i = \gamma_i / G_1$. This implies that $\beta_1 x_1 + \Sigma_{i=2}^{n}\beta_i x_i = 0$ and all β_i values are coprime, i.e. $\gcd((\beta_k)_{k\geq 1}) = 1$. Let $G_2 = \gcd((\beta_k)_{k\geq 2})$. We can then further rewrite the fact $\vec{x} \in S$ as:

$$\begin{array}{rl} & \vec{x} \in S \\ \Leftrightarrow & \Sigma_{i=1}^{n}\beta_i x_i = 0 \\ \Leftrightarrow & \beta_1 x_1 + G_2(\Sigma_{i=2}^{n}\beta_i' x_i) = 0 \\ \Leftrightarrow & x_1 = -G_2(\Sigma_{i=2}^{n}\beta_i' x_i)/\beta_1 \end{array}$$

Since β_1 and G_2 are coprime, it means that $\beta_1 | \Sigma_{i=2}^{n}\beta_i' x_i$ and x_1 can be written as $x_1 = \alpha_1 G_2$ for the integer $\alpha_1 = -\Sigma_{i=2}^{n}\beta_i' x_i / \beta_1$. Applying the definitions of G_2, β_i and G_1 results in $x_1 = \alpha_1 K_{11}$. Consider now a new vector $\vec{y} = \vec{x} - \alpha_1 \vec{s}_1$. Since $\vec{x}$ and $\vec{s}_1$ are elements of S, vector $\vec{y}$ is also an element of S. However, vector $\vec{y}$ has a special structure: its first component is 0. We repeat the described procedure on $\vec{y}$ and $\vec{s}_2$. This way we derive the value for an integer α_2 and a new vector $\vec{z}$ who has the first two components 0.

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

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

5.1.3 Finding a Solution of an Equation

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

We start with a base case when there are only two variables: $K + \gamma_1 u_1 + \gamma_2 u_2 = 0$. Because $K/\gcd(\gamma_1, \gamma_2)$ is an integer, we introduce the integer $\alpha = K/\gcd(\gamma_1, \gamma_2)$.

Following Bézout's identity there exist integers v_1 and v_2 such that $\gamma_1 v_1 + \gamma_2 v_2 = \gcd(\gamma_1, \gamma_2)$. By multiplying this last equation by $-\alpha$, we obtain $\gamma_1(-\alpha)v_1 + \gamma_2(-\alpha)v_2 + K = 0$ We define $u_i = v_i \cdot (-\alpha)$ and therefore u_1 and u_2 are correct solutions of the equation.

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

Another algorithm for finding a solution of an equation $K + \Sigma_{i=1}^{n} \gamma_i u_i = 0$ is presented in [3]. It also runs in polynomial time and allows bounded inequality constraints as well. However, we chose the algorithm presented here because it of its simplicity. It can be easily implemented. Moreover, we are only interested in finding one solution of an equation. We have no additional constraints nor we are interested in a characterization of all solutions under such additional constraints.

Here we did not describe an algorithm how to find integers integers w_1 and w_2 such that $a_1 w_1 + a_2 w_2 = \gcd(a_1, a_2)$, for given integers a_1 and a_2. It is a well-know algorithm, present in most of the textbooks on algorithms under the name Extended Euclidean algorithm. We refer to a reader to contact any of textbooks, for example [11][Figure 31.1].

5.2 Processing Inequality Constraints

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

Let x be an output variable which we are processing. Every conjunct can be rewritten in one of the two following forms, for $i \in [1, I]$ and $j \in [1, J]$:

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

As before, x should be a value which is greater than all lower bounds and smaller than all upper bounds. However, this time we also need to take into an account that x has to be an integer, so we convert upper and lower bounds to integers. We define x as the minimum of the upper bounds b, if it exists, and the maximum of the lower bounds a otherwise.

```
a = $\max_i \lceil A_i/\alpha_i \rceil$
b = $\min_j \lfloor B_j/\beta_j \rfloor$
$x$ = b **if** b defined, **else** a.
```

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

$$\bigwedge_{i,j} \lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \tag{5.1}$$

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

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

$$\begin{aligned} &\lceil A_i/\alpha_i \rceil \leq \lfloor B_j/\beta_j \rfloor \\ \Leftrightarrow &\left\lceil \frac{A'_i}{L} \right\rceil \leq \left\lfloor \frac{B'_j}{L} \right\rfloor \\ \Leftrightarrow &\frac{A'_i}{L} \leq \frac{B'_j - B'_j\%L}{L} \\ \Leftrightarrow &B'_j\%L \leq B'_j - A'_i. \end{aligned} \tag{5.2}$$

In order to convert the modulo expression to linear arithmetic expressions, we either decompose the B'_j or the A'_i modulo L. Suppose, for example, $J \leq I$. In order to generate a small formula, we prefer to decompose B'_j from (6.2) (a similar decomposition can be made on the A'_i). We introduce J special input variables $k'_1, \dots k'_J$ which will be in the range $[0, L-1]$ and J new output variables $x'_1, \dots x'_J$. We define the following new formula:

$$\bigwedge_{j\in[1,J]} \left\{ B'_j = Lx'_j + k_j \wedge \bigwedge_{i\in[1,I]} k_j \leq B'_j - A'_i \right\} \tag{5.3}$$

Then, we solve the problem by replacing all inequalities containing x by (5.3). The solution of the general problem is obtained by iterating the $(k_j)_j$ variables through all possible values in $[0, L-1]^J$. For this purpose, we introduce a for-loop structure iterating over the solution $(\mathsf{pre}'[\vec{a}, k_1 \dots k_J], \Psi'[\vec{a}, k_1 \dots k_J])$ of (5.3), which results in a program $\Psi[\vec{a}]$ of the form:

```
$[0, L]^J$ find { $(k_1 \dots k_J) \Rightarrow \mathsf{pre}'[\vec{a}, k_1 ... k_J]$ } match {
  case Some($(k_1 \dots k_J)$) $\Rightarrow$
    $\vec{r}$s = $\Psi[\vec{a}, k_1 ... k_J]$
  case None $\Rightarrow$
    throw new Exception("No solution exists")
}
```

Note that the synthesizer terminates because we introduced J new output variables but also J new equalities containing them, so by removing equalities like in Section 5.2, we will be able to remove J output variables before dealing with inequalities again.

The result is correct because for each possibility of writing $B'_j\%L$, the remaining conditions are exactly the ones mentioned above. Finally, the precondition of such a generated program is the finite existentially quantified condition:

$$\mathsf{pre}[\vec{a}] = \exists(i_1...i_J) \in [0, L]^J.\mathsf{pre}'[\vec{a}, i_1...i_J]$$

5.3 Disjunctions in Presburger Arithmetic

We can again lift synthesis for conjunctions to synthesis for arbitrary propositional combinations is to apply the method of Section 3.7.2. We also obtain complexity that is one exponential higher than the complexity of synthesis from previous section. Approaches that avoid disjunctive normal form can be used in this case as well [41, 15, 58], and we expect the lower and upper bounds on quantifier elimination [58] to apply to the size of the synthesized code.

5.4 Clarifications and Optimizations

Merging inequalities.

Two exactly opposed inequalities can be merged to produce an equality, which allows eliminating variables more efficiently.

Heuristic to choose the variable.

When removing equalities in Section 5.2, choosing the variable x_n can be done by choosing the one whose least common multiple over all the coefficients is the smallest. This reduces the number of integers to iterate over.

Partial modulo ending.

Partial modulo ending is an optimization derived from the case of base decomposition, e.g. $a = x_1 + 2048 * x_2$ where $0 \leq x_1 < 2048$. Consider the equation (6.2) page 54 for some pair of lower and upper bound (A'_i, B'_j). If $B'_j - A'_i$ is reducible to a constant $K \geq L - 1$, then we can dismiss the subsequent modulo splitting for the particular pair of equations, because $B\%L \leq L - 1 \leq K = B'_j - A'_i$.

Consider the equivalent equation (6.1) page 54 for some pair of lower and upper bound $(A_i, \alpha_i, \beta_i, B_j)$. If $\alpha_i = 1$, then we can rewrite (6.1) to

$$A_i \leq \left\lfloor \frac{B_j}{\beta_j} \right\rfloor \tag{5.4}$$

$$\Leftrightarrow \beta_j A_i \leq B_j \tag{5.5}$$

so we do not need to split the condition. A similar optimization can be made if $\beta_j = 1$.

5.5 Complexity

We next describe the complexity of our algorithms, for both the synthesis process itself and the synthesized programs. A conversion of the formula to Disjunctive Normal Form might increase by an exponential factor both the running time and the space of our synthesizer and also the size of the generated program (see 5.5.2). The execution time would also be multiplied by an exponential factor as we are checking the conditions in sequence. In the following, we consider the complexity when F is a conjunction of literals.

5.5.1 Synthesizer Time Complexity

The number of times $\Omega(E, N, V)$ our solver goes back to 5.1, given the number of equalities E, inequalities N and output variables V, is bounded from above by:

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

This result is proved in appendix A.2 page 78. Note that, the algorithm has again good efficiency in the absence of inequalities. In any case, it is also polynomial when V is constant.

5.5.2 Generated Programs Size

Each recursive call to 5.1. also means at least an assignment, so there can be at least doubly exponential assignments.

5.5.3 Generated programs Time Complexity

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

5.5.4 Bézout witness and base generation Complexity

The current code used to compute solutions can be found in appendix B page 81.

The complexity of computing `advancedEuclid` of two numbers x and y is bounded by $O(\max(\ln(x), \ln(y)))$ (Finck's theorem[13]).

The computation of the successive GCDs in `bezoutWitness` is done in

$$O(|a| \ln(\max(a)))$$

, where a is the input vector of coefficients. Then the loop body is called $O(|a|)$ times, and the execution of its body is constant, with at most one call to `advancedEuclid`. The total complexity of `bezoutWitness` is therefore in

$$O(|a| \ln(\max(a)))$$

The computation of the successive GCDs in `bezoutWitnessWithBase` is done in $O(|a|\ln(\max(a)))$, where a is the input vector of coefficients. Then the loop body is called $O(|a|)$ times, with at most one call to `bezoutWitness`.

The total complexity of `bezoutWitnessWithBase` with the list of input coefficients named a is therefore in

$$O(|a|^2 \ln(\max(a)))$$

It looks hard to do better, since the complexity matches to a log factor the size of the 2D array to be filled with coefficients.

Chapter 6

Parametrized Linear Integer Synthesis

Indecision may or may not be my problem.

Jimmy Buffett

In the previous Chapter, we explained how to generate programs satisfying linear equations and inequations with integer coefficients. The coefficients of the output variables can be *arbitrary expressions* in the input variables. This allows us to state problems like the following:

$$\left\{ax_1 + (a^2+1)x_2 = a^3 - a,\ ax_1 \geq b,\ x_2 \leq b^3\right\}$$

The main algorithm is very similar to the one of Chapter 5 but instead of manipulating known integers coefficients, it manipulates arbitrary expressions. Since the only previous compile-time decisions on coefficients were to check their sign (negative, zero, positive), the generated programs now have to do these checks at run-time.

6.1 Equality contraints

We need to extend the algorithm which deals with equalities in Chapter 5 to support arbitrary expressions as coefficients $c[\vec{a}]$ in front of output variables. For that, we integrate the coefficient transformation[1] described in section 5.1.2 into the synthesized code, as a new primitive.

[1]For the complete code of `bezoutWitnessWithBase`, see appendix B

$$[\![\ c_0[\vec{a}] + c_1[\vec{a}]\cdot x_1 + c_2[\vec{a}]\cdot x_2 + c_3[\vec{a}]\cdot x_3 = 0 \quad\wedge\quad F[\vec{a},\vec{x}]\ \ ,\vec{a},\vec{x},\Gamma]\!] =$$

$$(\mathsf{pre}[\vec{a}] \wedge \mathsf{pre}_e, \left(\begin{array}{r} \{(k_{ij})_{[1\ldots3]^2} = \texttt{bezoutWitnessWithBase}(c_0,c_1,c_2,c_3)\} ::: \\ \Psi[\vec{a}] ::: \{\textbf{val}\ \vec{x}_{1\ldots3} = (k_{ij})_{[1\ldots3]^2}\cdot(1,\vec{x}'_{1\ldots2})\} ::: \Gamma \end{array}\right)),\ \text{where}$$
$$\mathsf{pre}_e = \gcd(c_1,c_2,c_3)|c_0 \text{ and}$$
$$(\mathsf{pre}[\vec{a}],\Psi[\vec{a}]) = [\![F[\vec{a},\vec{x}_{1\ldots3} := (k_{ij})_{[1\ldots3]^2}\cdot(1,\vec{x}'_{1\ldots2}),\vec{a},(\vec{x}\backslash\vec{x}_{1\ldots3}) ::: \vec{x}'_{1\ldots2},()]\!]$$

The computation of the coefficients $(k_{ij})_{[1\ldots3]^2}$ which used to be done at compile-time, is now done at run-time. Of course, if all coefficients c_i are known, we can do the computations at compile-time. If only a part of the coefficients are known at compile-time, it is still possible to specialize the computation in order to introduce a least number of variables. We added such an optimization when a coefficient is equal to 1, independent of the other coefficients.

Furthermore, if the coefficients c_1, c_2, c_3 are all zero, keeping the precondition $\gcd(c_1, c_2, c_3)|c_0$ would lead to a division by zero exception. In this case, the precondition pre_e should be defined as $c_0[\vec{a}] == 0.$. To deal with such situation, we need to embed a new abstraction into linear combinations of output variables to describe the fact that all coefficients are zero, or not all of them are zero.

In the following code, we describe the implementation of this abstraction. Two fields record the possible values, and two methods are provided to show how to assume facts about an expression which contains an abstraction.

If `all_coefficients_can_be_zero` is false, then it means that all coefficients within this expression can be zero a the same time.

If `one_coefficient_can_be_nonzero` is false, then it means that all coefficients are zero. If both variables are true we do not have information about the current linear combination.

```
trait CoefficientAbstraction {
  // Abstraction to express that all coefficients are zero or not
  private var all_coefficients_can_be_zero: Boolean = true
  private var one_coefficient_can_be_nonzero: Boolean = true
  ...
  def allCoefficientsAreZero = (
           all_coefficients_can_be_zero &&
          !one_coefficient_can_be_nonzero)
  ...
  def assumeNotAllCoefficientsAreZero = {
    cloneWithCoefficientAbstraction(false, one_coefficient_can_be_nonzero)
  }
  def assumeAllCoefficientsAreZero = {
    cloneWithCoefficientAbstraction(all_coefficients_can_be_zero, false)
  }
}
```

The code summary to solve equalities is the following. It takes the first equality available, then it splits on the previously mentioned abstraction value. If the abstraction asserts that all coefficients are zero, the stand-alone coefficient $c_0[\vec{a}]$ is assumed to be zero, and the solver keep on dealing with the rest. If the abstraction asserts that not all coefficients are zero, then solves it naturally. Finally, if the abstraction is not defined, then it solves the two possibilities and creates a if-then-else structure.

```
sorted_equalities match {
  case (EqualZero(combination@Combination( const_part, linear_part )))::rest_equalities ⇒
    if( combination.allCoefficientsAreZero ) {
      addPrecondition( const_part===0 )
      ...solves remaining...
    } else if( combination.notAllCoefficientsAreZero ) {
      var coefficients = getCoefficients( linear_part )
      val gcd = InputGCD(coefficients)
      addPrecondition( Divides( gcd, Combination(const_part, Nil) ) )
      ...solves remaining...
    } else {
      val coefs_are_zero = ... // Formula to express that all coefficients are zero
      var (cond1, prog1) =
        solve( (combination.assumeAllCoefficientsAreZero === 0) :: rest_equalities ... )
      var (cond2, prog2) =
        solve( (combination.assumeNotAllCoefficientsAreZero === 0) :: rest_equalities ...)
      return ((cond1 || cond2),
              IfThenElse( coefs_are_zero && cond1,
                          prog1,
                          ElseIf( !coefs_are_zero && cond2,
                                  prog2) }
```

6.2 Processing Inequality Constraints

The first step of the algorithm to solve inequalities (see Section 5.2 page 44) was to take a variable, and to separate upper and lower bounds. We cannot do this now, because we might not know all signs of the coefficients.

Consider the following example, where we restrict the variables x and y using the two unknown coefficients a and b. Depending on the sign of a or b, we have different upper and lower bounds, and thus different solutions.

$$\left\{ \begin{array}{c} 1 - 2x + ay \leq 0 \\ -1 + bx + 3y \geq 0 \end{array} \right\}$$

This is why we need a way to store the sign of expressions. The complete solution is presented in appendix D page 85

6.2.1 Sign abstraction

Our first idea is to embed a sign abstraction inside each expression of input variables. In the same manner some program checkers symbolically execute programs by abstracting variables domains to $\mathcal{P}(\{]-\infty, 0[, \{0\},]0, \infty[\})$, we extend the existing input expressions classes (see appendix C page 83) with a trait containing the possible signs of the current input expression. Other possible future implementations of this abstraction are explained in Chapter 9 page 67.

```
trait SignAbstraction {
  // Simple >0, =0 and <0 abstraction
  private var can_be_positive: Boolean = true
  private var can_be_zero : Boolean = true
  private var can_be_negative: Boolean = true
  def isPositive() = can_be_positive && !can_be_negative && !can_be_zero
  def isPositiveZero()= (can_be_positive || can_be_zero) && !can_be_negative
  def isZero() = can_be_zero && !can_be_positive && !can_be_negative
  ...
  def assumePositiveZero() = {
    cloneWithSign(can_be_positive, can_be_zero, false)
  }
  ...
}
```

We also implemented sign propagation through expressions. Our algorithm is able to deduce and propagate constraints such that a sum of positive numbers is positive, a product of two non-zero numbers non-zero and vice-versa, the absolute value of a non-zero expression strictly positive, etc. This allows us to optimize cases like $bx + (b + b^3)y \geq c$, where if the algorithm assumes $b > 0$, it will deduce that $b + b^3 > 0$, to avoid the extra effort of making an assumption about the sign of this expression.

6.2.2 Splitting on the sign

When we determine the lower and upper bounds of a variable x in inequations, if not all the signs of the coefficients in front of x are determined, we repeatedly assume a sign for each one of them, and solve the resulting inequations as in section 5.2. To operate a sign split, our algorithm takes the variable which has the least undetermined coefficients.

Let us take the variable x in the previous example to apply the sign determination. The algorithm considers the three possibilities for b's sign, and splits the main problem into three sub-problems. The inequation in which x appears is first considered as an upper bound when $b > 0$, as nothing if $b = 0$, and as a lower bound if $b < 0$.

$$\text{Main problem: } \left\{ \begin{array}{c} 1 - 2x + ay \leq 0 \\ -1 + bx + 3y \geq 0 \end{array} \right\}$$

Precondition	Lower bounds		Upper bounds		Remaining
$b > 0$	$\left\{ \begin{array}{c} 1 - 3y \leq bx \\ 1 + ay \leq 2x \end{array} \right\}$	$\wedge$	$\emptyset$	$\wedge$	$\emptyset$
$b = 0$	$\{1 + ay \leq 2x\}$	$\wedge$	$\emptyset$	$\wedge$	$\{1 - 3y \leq 0\}$
$b < 0$	$\{1 + ay \leq 2x\}$	$\wedge$	$\{(-b)x \leq -1 + 3y\}$	$\wedge$	$\emptyset$

Below we show the corresponding code in which the sign is applied by calls to `assume` methods. This code outputs the stream of (upper bounds, lower bounds, remaining equations) for all the different signs that a coefficient b of a variable x can take in a given inequality. If the sign is determined, which is the case of the first three if-then-else, it returns a stream with a single element. If the sign is not determined, then it will gather all possibilities thanks to the Stream command `append`. If the coefficient b can be positive, it is assumed so and the algorithm creates a sub-problem. Same consideration when b can be zero or when it can be negative.

```
// b is a coefficient for a variable $x$, in an inequality (E) $-1 + b \cdot x + 3y \geq 0$
// This expression return a stream of [lower_bounds, upper_bounds, remaining]

if(b.isPositive) {
  /// This inequality is a lower bound, we handle it so.
  Stream(lower_bounds ++ E, upper_bounds, remaining)
} else if(b.isZero) {
  /// This is not a bound for variable $x$ so we continue without it.
  ...
  Stream(lower_bounds, upper_bounds, remaining ++ E)
} else if(b.isNegative) {
  /// This inequality is an upper bound, we handle it so.
  ...
  Stream(lower_bounds, upper_bounds ++ E, remaining)
} else {
  (if(b.can_be_positive) {
    val b_positive = b.assumeSign(1) // Then replace b by b_positive
    ...
    Stream(lower_bounds_replaced ++ E, upper_bounds_replaced, remaining_replaced)
  } else Stream()) append
  (if(b.can_be_zero) {
    val b_zero = b.assumeSign(0) // Then replace b by b_zero
    ...
    Stream(lower_bounds_replaced, upper_bounds_replaced, remaining_replaced ++ E)
  } else Stream()) append
  (if(b.can_be_negative) {
    val b_negative = b.assumeSign(−1) // Then replace b by b_negative
    ...
    Stream(lower_bounds_replaced, upper_bounds_replaced ++ E, remaining_replaced)
  } else Stream())
}
```

6.2.3 Normalizing inequations

For a given coefficients sign choice, we obtain the following upper and lower bounds for some $i \in [1, I]$ and $j \in [1, J]$.

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

The difference with section 5.2 is that now the α_i and β_i are arbitrary expressions made of input variables, not only numbers. Due to our sign determinization procedure explained in section 6.2.2, we are guarantying that the α_i and β_j are positive, so we can divide by them. Define the witness for x as the minimum of the upper bounds, if it exists, otherwise let x be the maximum of the lower bounds:

```
a = $\max_i \lceil A_i/\alpha_i \rceil$
b = $\min_j \lfloor B_j/\beta_j \rfloor$
$x$ = b if b defined, else a.
```

In the case when there are upper and lower bounds, we have to continue to solve the equations with the constraint $\max_i \lceil A_i/\alpha_i \rceil \leq \min_j \lfloor B_j/\beta_j \rfloor$. This means that for each i and j, we need to add the constraint corresponding to

$$\left\lceil \frac{A_i}{\alpha_i} \right\rceil \leq \left\lfloor \frac{B_j}{\beta_j} \right\rfloor \tag{6.1}$$

Define $L = \mathrm{LCM}_{i,j}(\alpha_i, \beta_j)$, $A'_i = \frac{L}{\alpha_i} A_i$, $B'_j = \frac{L}{\beta_j} B_j$ If the α_i and β_j are not integers, the variable L is computed at run-time, as a new non-zero input variable. This is the same for the expressions $\frac{L}{\alpha_i}$ and $\frac{L}{\beta_j}$, which are in the same case computed at run-time, and replaced with new input variables. First, we rewrite these inequalities:

$$\left\lceil \frac{A'_i}{L} \right\rceil \leq \left\lfloor \frac{B'_j}{L} \right\rfloor$$

$$\Leftrightarrow \frac{A'_i}{L} \leq \frac{B'_j - B'_j\%L}{L}$$

$$\Leftrightarrow B'_j\%L \leq B'_j - A'_i \tag{6.2}$$

Then we decompose either the B'_j or the A'_i modulo L. Suppose, for example, $J \leq I$. In this case we decompose B'_j from (6.2) (a similar decomposition can be done for the A'_i). We introduce J input variables $k_1, \ldots k_J$ and J new output variables $x'_1, \ldots x'_J$ and define the following new equations:

$$\bigwedge_{j \in [1,J]} \left\{ B'_j = Lx'_j + k_j \wedge \bigwedge_{i \in [1,I]} k_j \leq B'_j - A'_i \right\} \tag{6.3}$$

Then, we replace all inequalities containing x by (6.3) and solve the resulting problem with the remaining inequations F_1.

$$\begin{aligned}&(\mathsf{pre}'[\vec{a}, k_1, \ldots, k_j], \Psi'[\vec{a}, k_1, \ldots, k_j]) = \\ &\quad [\![(6.3) \wedge F_1, \vec{a} ::: (k_1, \ldots, k_j), \vec{x} ::: \vec{x}', \Gamma]\!]\end{aligned}$$

Since the bounds of the (k_j) variables are in $[0, L-1]^J$, and L might be computed at run-time, we need a loop to iterate over the possibilities. We introduce such a for-loop structure which finds the first value of $\vec{k}$ satisfying the precondition $\mathsf{pre}'[\vec{a}, k_1 \ldots k_J]$, and executes the corresponding program $\Psi'[\vec{a}, k_1 \ldots k_J]$ when it encounters one.

```
$[0, L]^J$ find { $(k_1 \ldots k_J) \Rightarrow \mathsf{pre}'[\vec{a}, k_1 ... k_J]$ }
match { case Some($(k_1 \ldots k_J)$) $\Rightarrow$
```

```
    $\vec{r}$s = $\Psi[\vec{a}, k_1...k_J]$
  case None ⇒
    throw new Exception("No solution exists")
}
```

Note that the synthesizer terminates because we introduced J new output variables but also J new equalities containing them, so by removing equalities like in Section 5.2, we will be able to remove J output variables before dealing with inequalities again. Note as well that the resulting program terminates, because L is determined at run-time, even if the number of times this for-loop executes is not bounded by a known constant anymore. For a full version of the generated code for the example in section 6.1 see Appendix D page 85.

Note on complexity. The size of the generated programs is larger if there are run-time decisions to take. The size increases exponentially with the number of undetermined coefficients.

Chapter 7

Implementation and Performance

In this section we describe the output format produced by our synthesizer and its performance.

7.1 Implementation

Our synthesis procedure outputs an abstract syntax tree containing the generated program with conditions. The first conversion we made was to generate a parsable string of the program, naturally in Scala. Indeed, Scala has nice idioms to implement some non-trivial features we needed for synthesis: the local variable assignment in if-conditions, bounded existentially quantified formulas, and the extraction of witnesses in these bounded existentially quantified formulas (see the mid-column of table 7.1). We also present the case split idiom after these three main features.

Then for the purpose of the example presented in section 2.2 page 22, we also implemented a converter from our abstract syntax tree to a parsable string in Python. This could be done using Python's lambda expression, non-boolean evaluation for boolean operators — like `(false or ‘‘Test’’) == ‘‘Test’’`, list comprehensions and the `reduce` function (see the right-column of table 7.1).

Third, we[1] have implemented our synthesis procedures as a Scala compiler extension. 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 [14]. Besides, the compiler supports plugins that can serve as additional phases in the compilation process.[2] We used an off-the-shelf decision procedure [40] to handle the compile-time checks.

Our plugin supports the synthesis of integer values through the `choose` function constrained by linear arithmetic predicates, as well as the synthesis of set values constrained by predicates of the logic described in Section 8.4. Additionally, it can synthesize code for pattern-matching expressions on integers such as the ones presented in Section 2.1.3.

[1]Contrary to the first two string generation that I wrote, the compiler extension and the subsequent abstract syntax tree conversion is mostly Philippe Suter's work

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

Feature to implement	Generated Scala code	Generated Python code

Variable assignment in an if-condition	**if**({**var** c = b/2;
	    c > 5 && c < 7}) {
	  ...
	}
		if (lambda c:c > 5 and
		    c < 7)(b/2):
		  ...

Bounded existential quantifiers in if-conditions.	**if**((1 to b) exists {
	    i => i % c == 0 }) {
	  ...
	}
		if(reduce((lambda a, i:a or
		    i % c == 0),
		    [i **for** i **in** xrange(1, 1 + b)],
		    False))):
		  ...

Extraction of witnesses for for-loops.	(1 to b) find { i =>
	    i % c == 0 } **match** {
	  case Some(i) => ...
	  case None => ...
	}
		kp = reduce((lambda a, i:a or
		    i % c == 0),
		    [i **for in in** xrange(1, 1 + b)],
		    False)))
		if kp:
		  i = kp
		  ...

Case split and variable assignment	**val** (x, y) = **if**(...) {
	  (5, b+c)
	} **else if** (...) {
	  (−c, b/c)
	} **else if** (...) {
	  ...
	} **else throw** ...
		if ...:
		  (x, y) = (5, b+c)
		elif ...:
		  (x, y) = (−c, b/c)
		elif ...:
		  ...
		else:
		  raise ...

Table 7.1: Comparison of Generated Scala code and Generated Python code

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

Figure 7-1: 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). All times are in seconds. There are no compile-time checks for the synthesis of set values.

7.2 Performance

Figure 7-1 shows the compile times for a set of benchmarks, with and without our plugin (in the latter case, the generated code is of course of no use). The examples *SecondsToTime*, *FastExponentiation* were presented in Section 2.1.3. *ScaleWeights* computes solutions to a puzzle, *PrimeHeuristic* contains a long pattern-matching expression where every pattern is checked for reachability. *SplitBalanced* is a set problem using the BAPA theory (see related work section 8.4) and *SetConstraints* is a variant of it. We also measured the times with all benchmarks placed in a single file, as an attempt to balance out the time taken by the Scala compiler to start up. Our numbers show that the additional time required for the code synthesis is minimal. One should also note that the code we tested contained almost exclusively calls to the synthesizer, which is clearly not representative of what we expect will be the common practice of using a selective number of invocations.

Overall, we believe that implementation is fast enough to be practical and allows us to have benefits of synthesis in Scala.

Chapter 8

Related work

I have never met a man so ignorant that I couldn't learn something from him.

Galileo Galilei (1564 - 1642)

In this section, we present related synthesis concepts, ideas, algorithms, that have influenced us or helped us to design our own synthesis algorithms.

8.1 Overview

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

Early work on synthesis [35, 34] 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 [53] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. Furthermore, the work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures. This work is more ambitious and aims to synthesize entire algorithms. By nature, it cannot be both terminating and complete over the space of all programs that satisfy an input/output specification (thus the approach of specifying program resource bounds). In contrast, we provide completeness guarantees for a given specification, but focus on synthesis of program fragments with very specific control structure dictated by the nature of the decidable logical fragment.

Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [52, 50, 51, 27, 54]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. In contrast, our synthesis uses the mathematical structure of a decidable theory to explore space of all functions that satisfy the specification. This enables our approaches to achieve completeness without putting any a priori bound on the syntax tree size. Indeed, some of the algorithms we describe can generate fairly large and efficient programs. We expect that our techniques could be fruitfully integrated into sketching frameworks.

Synthesis of reactive systems generates programs that run forever and interact with the environment. However, known complete algorithms for reactive synthesis work with finite-state systems [46] or timed systems [1]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [57]. These techniques usually take specifications in a fragment of temporal logic [45] and have resulted in tools that can synthesize useful hardware components [25, 24]. Our work examines non-reactive programs, but supports infinite data without any approximation, and incorporates the algorithms into a compiler for a general-purpose programming language.

Automata-based decision procedures, such as those implemented in the MONA tool [28] could be used to synthesize efficient (even if large) code from expressive specifications. The work on graph types [29] proposes to synthesize fields given by definitions in monadic second-order logic. The subsequent work [38] has focused on verification as opposed to synthesis.

The way we deal with inequations is partially based on a quantifier elimination procedure [48]. Several other linear inequalities solving techniques over integers and rationals have also been explored in [22, 8]. Some have even been applied to synthesis [21] from Input/Output examples, which is in contrast to synthesis from full formal specification.

Our approach can be viewed as sharing some of the goals of partial evaluation [26]. However, we do not need to employ general-purpose partial evaluation techniques (which typically provide linear speedup), because we have the knowledge of a particular decision procedure. We use this knowledge to devise a synthesis algorithm that, given formula F, generates the code corresponding to the invocation of this particular decision procedure. This synthesis process checks the uniqueness and the existence of the solutions, emitting appropriate warnings. Moreover, the synthesized code can have reduced complexity compared to invoking the decision procedure at run time, especially when the number of variables to synthesize is bounded.

8.2 Comparison of our Synthesis algorithms to other systems

Our synthesis algorithms manipulate unbounded data structures and deal with parametrized input. With this perspective, we can compare their category to other logic and arithmetic synthesis-related tools. The result is shown in Fig. 8-1. On the left

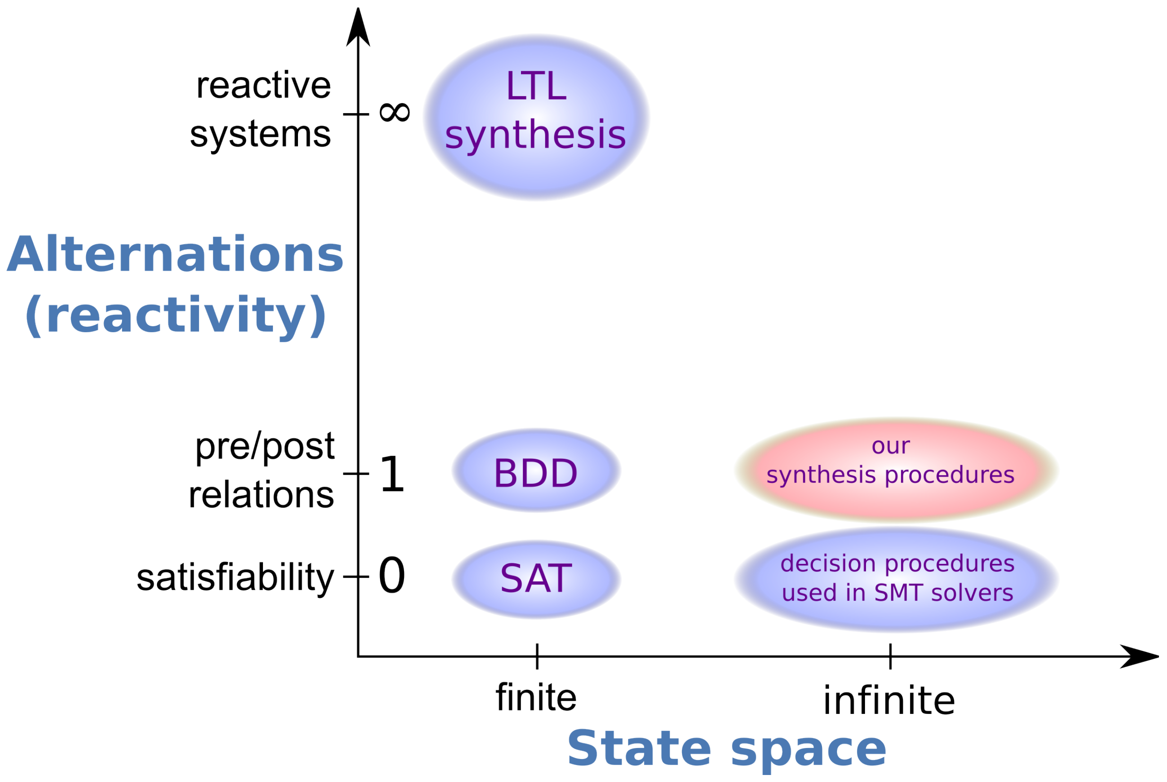


Figure 8-1: Complete algorithms for correct-by-construction synthesis

column, we show synthesis systems dealing with finite states. A simple SAT solver — Satisfiability of propositionnal formulas like $p \wedge (\neg P \vee Q)$ — produces boolean finite-state models of the formula. Boolean functions generators [30] are stronger, because they produce input-dependent models, which can be represented on BDD[1]. LTL[2] synthesis is even stronger, as it can deal with infinite and repeatitive processes in which the input can constantly change [24]. On the right column, we show synthesis systems dealing with infinite states. A integer decision procedure using quantifier elimination is one example. Finally, our synthesis algorithms produce unbounded integer functions.

8.3 Existing Synthesis Ideas

Synthesis is also an old concept about the idea of reducing the burden of the programmer. A compiler itself is nothing more than a synthesizer from one middle-level code representation (e.g. C++) to another low-level code representation (e.g. x86 assembly).

From time to time new programming constructs emerge, for which it is the compiler's burden to create a program that meets the specification. We will now discuss

[1]Binary Decision Diagrams
[2]Linear Temporal Logic

two examples of such programming constructs.

8.3.1 Regular expressions

To recognize structures defined by the regular language `{(a+b)*(b+c)*(a+c)*}` , one could write the following state machine by hand.

```
var state=0
string_to_test foreach { x =>
  if(state=0) { if(x != 'c') state = 1 else state = 2 }
  else if(state=1) { if(x == 'c') state = 3 }
  else if(state=2) { if(x == 'a') state = 4 }
  else if(state=3) { if(x == 'a') state = 4 else if(x=='b') state = 2 }
  else if(state=4) { if(x == 'b') state = 5 }
}
return (state != 5)
```

However, it is much easier to use an existing library to compile the language string `{(a+b)*(b+c)*(a+c)*}` into an automaton that does the same as the previous program. For instance, the high-level code for this example could be written as follow.

```
val Language = new Regex("""(?:a|b)*(?:b|c)*(?:a|c)*""")
string_to_test match {
  case Language() => return true
  case _ => return false
}
```

The variable `Language` represents a dynamically compiled form of the program above. It is close to the notion of synthesis, because this variable can be reused afterwards without loosing performance. This regular expression pattern matching feature is available in Scala.

8.3.2 Parser combinators

The use of high-order functions on parsers allows to write a code to parse a string and to obtain a syntax tree in a very clear way. Without high-order functions, the code would be long and difficult to write. For instance, let us consider the following basic lambda expressions.

$$\begin{array}{rcl} T := & & B\ B\ \{B\} \\ & | & B \\ B := & & \lambda x.\ T \\ & | & (T) \\ & | & x \end{array}$$

Writing a parser for this grammar is a painful task in a low-level programming languages, because it consists of a lot of functions and if-then-elses statements. Thanks to the expressivity of for example the Scala [42] programming language, it is possible to hide such complexity by special pattern matching.

```
(...)
  def T: Parser[T] = (
      B~B~rep(B) ^^ {case t1~t2~otherBs => ApplicationSequence(t1, t2, otherBss)}
    | B
    | failure("illegal start of term"))

  def B: Parser[T] = (
      "λ" ~> (ident ~ "." ~ T) ^^ { case s ~ "." ~ t => Abstraction(Variable(s), t) }
    | "(" ~> T <~ ")" ^^ { case t1 => t1 }
    | ident ^^ { case s => Variable(s) }
    | failure("illegal start of BasicTerm")
  )
```

It does not only reduce the amount of code but it is almost as fast as a hand-written parser. The code is much clearer and represents better what the programmer has in mind.

8.3.3 Dealing with unknown coefficients

The idea to introduce for-loops when coefficients are unknown, explained in Chapter 6, has been tackled in [58, 33] by introducing the notion of "bounded quantifiers", in the case of decision procedures.

8.4 BAPA Synthesis

BAPA [6] stands for Boolean Algebra for Presburger Arithmetic. It contains arithmetic, set operations, and a function symbol to represent the cardinality of a set, thus making the link between both.

As a separate work, we[3] discovered that the algorithms described in this thesis could be used to obtain synthesis for BAPA expressions, namely to produce functions which output not only integers, but also sets. For that, we took an existing decision procedure described in [32] and updated it to obtain a BAPA synthesizer [36].

[3]This is mostly Ruzica Piskac's work, both for theory and implementation, and Philippe Suter made the Scala plug-in work with this theory.

Chapter 9

Future work

The best way to predict the future
is to invent it.

Alan Kay

Anyone looking for ideas on how to make it better might be interested in this section. We will also present others interesting synthesis problems.

9.1 Improvement for the current synthesizer

Although our current synthesizer is now complete, there are several ways to improve both the quality of the synthesizer and the quality of the synthesized programs.

Modular Arithmetic. Most programming languages do not use unbounded integers. Usually, they use 32 or 64 bits integers. Our synthesizer is correct if the integers are implemented as BigInts, i.e. unbounded integers. With bounded integers, neither functions like `LCM` or `Bézout` would work for large numbers, nor would the generated programs. Whenever it is needed, for robustness against failure, for industrial applications, we could extend the synthesizer to handle modular arithmetic.

External abstraction storing. The current abstraction implementation is to extend the existing InputTerm classes (see appendix C page 83) with an abstraction trait (see section 6.2.1 page 51).

Although this allows us to write nice sign checks like `coef.isPositive`, it is not optimized for propagating assumptions about the sign of an expression. To improve the implementation, we could store the sign abstraction in an external map, along with a dependency graph between values in order to quickly propagate signs. The resulting simplifications would enhance both the program generation speed and the quality of the generated programs.

Better abstractions. Although it is not entirely obvious, we encountered lack of optimization on the code 2.5 page 20. For the code shown on the right, the current abstractions cannot detect that the expression $x + 1024y - 1920yb$ is positive. Therefore, the synthesizer forces the computation of a floored division:

$$\min(639, ((x + 1024y - 1920yb) - (3 + (x + 1024y - 1920yb)\%3)\%3)/3)$$

In the equivalent optimized program, the synthesizer would have discovered that the expression $x + 1024y - 1920yb$ is positive and the computation of the previous expression would only use a simple division by 3 :

$$\min(639, (x + 1024y - 1920yb)/3)$$

With even better abstractions, the synthesizer could detect that the expression to the right of the min is always less or equal to 639, so it would reduce this expression to

$$(x + 1024y - 1920yb)/3$$

Furthermore there are many abstractions available for symbolic program execution and verification, e.g. the octagon abstraction [37] or the congruence abstraction [19]. Such abstractions could be implemented in our synthesizer, if the external storage for abstractions is available. It could even be possible to use different abstractions at the same time, even if the sign abstraction is always a minimal requirement for the synthesizer. Finally, it would result in more optimized generated programs.

Scalability and robustness. To verify that a theorem prover is sound, i.e. the theorems it proves are true, and the conjectures it disproves are false, it has to provide proofs, which can be checked by other independent formal proof checkers. For synthesis, we have the same objective. The synthesizer is written by humans, thus it might contain undiscovered bugs. To overcome this problem, there are two solutions. One possibility is to prove the synthesizer correct once and for all, which would be a great challenge but then nothing else would be needed. Alternatively, we could enhance the synthesizer to output a verifiable proof tree to explain why the generated programs are correct.

Independent post-optimization. Even though we already put some effort to generate short programs, and the final compiler optimizes further, it could be interesting to automatically optimize the resulting code by some independent ways like symbolic execution, or partially evaluating conditionals using validity checkers.

9.2 Synthesis ideas

In this section, we now present other synthesis ideas that we explored in parallel, but we did not focus on. They are always about reusing existing formal knowledge to automatically produce code that is otherwise painful to write.

9.2.1 Induction axioms for recursive programs

One of the first synthesis ideas we explored was to use a constructive version of the induction axiom. In contrast to the usual induction axiom used by theorem provers, it produces code. Furthermore, it can be used to generate recursive code given some properties about a recurring function f, an initial value a, a decreasing function w, and a property P. In the indexes, we show the signatures of the variables, where o represents a Boolean and i an integer or equivalent.

$$\begin{array}{l}
\forall P_{:\, i \to i \to o} \forall w_{:\, i \to i} \forall f_{:\, i \to i \to i} \forall a_{:\, i} \\
[P(0,a),\ \forall n_{:\, i} \neq 0.\ w(n) < n,\ \forall n_{:\, i} \forall u_{:\, i} P(w(n+1),u) \to P(n+1, f(n+1,u))] \\
\longrightarrow \forall n_{:\, i}.P(n, \\
\quad \text{val } g = (\_ \text{ match } \{ \text{ case } 0 \Rightarrow a; \text{case } n+1 \Rightarrow f(n+1, g(w(n+1)))) \\
\quad g(n))
\end{array}$$

This high-order logic rule describes a skeleton for programs. With i representing integer pairs, such kind of axiom could be refined and used for example to synthesize a Greater Common Divisor recursive algorithm, where $w(x,y)$:= if $(y < x)$ (y,x) else $(x, y-x)$", $f((x,y),g) := g$.

```
def gcd(x: Int, y: Int): Int = [
  if(x == 0)
    y
  else
    if(y < x)
      gcd(y, x)
    else
      gcd(x, y−x)
}
```

However, the main difficulty is to find inductive proofs, which was one of the first concerns of early program synthesis [35].

9.2.2 Other theories

Here are some small examples illustrating additional domains for synthesis.

String theory. To express the function `StringEndsWith`(t,s), a function determining if s is a suffix of t, one could use the following implicit definition and apply a synthesis algorithm on it:

Input	`StringEndsWith`$(t,s) \equiv \exists w.t = w \oplus s$
Output	`StringEndsWith`$(t,s) = t$`.substring`$(t.size - s.size, s.size) == s$

Examples	`StringEndsWith`(“*dogiscute*″, “*iscute*″) == `true`
	`StringEndsWith`(“*dogiscute*″, “*mydogiscute*″) == `false`

With synthesis, one would not bother about computing the length, doing the subtraction, etc. Even better, we could return the corresponding prefix w by an implicit function like

Input	$\texttt{StringGetPrefix}(t, s) \equiv w$ `where` $t = w \oplus s$
Output	$\texttt{StringGetPrefix}(t, s) = t.\texttt{substring}(0, t.size - s.size)$
Examples	$\texttt{StringGetPrefix}(\text{“}dogiscute\text{''}, \text{“}iscute\text{''}) == \text{‘}dog\text{‘''}$ $\texttt{StringGetPrefix}(\text{“}dogiscute\text{''}, \text{“}mydogiscute\text{''}) == \text{“''}$

Trigonometry. Trigonometry conversion can be painful, because of multiple cases. To convert Cartesian (x, y) coordinates to polar (r, θ) coordinates, it would be convenient to write the following code and let a synthesizer do the dirty job.

Input	**val** (r, θ) where x = r*cos(θ) && y = r*sin(θ) && 0 < $\theta \leq 2 * \pi$

Output	**val** (r, θ) = (sqrt(x^2 + y^2), {**val** t = 4*arctan(y/(x+|z|));
	 if (t $\leq$ 0) t + 2*π **else** t
	 })

Binary trees When working in binary trees, it could be convenient to implicitly express some properties, like to compute the minimum. Let us suppose that we have the following abstract syntax tree, where binary trees are represented either as a Leaf or a Node with two binary trees and an integer.

```
sealed abstract class Tree
case class Node(l: Tree, i: Int, r: Tree) extends Tree
case class Leaf extends Tree
```

A minimum might not be defined on such a tree. Therefore, an implicit minimum function would not return an `Int`, but an `Option[Int]` [1]. Considering the two cases, it would be convenient to supply the following code, and let a synthesizer infer the real code, with all the necessary pattern matching.

```
  def minimum(t: Tree[Int]): (n: Option[Int]) = {
    if(t==Leaf) {
      n where n == None
    } else {
      n where n ∈ content(t) && ∀m ∈ content(t). n ≤ m
    }
  }
```

where the function `content` is a trivial catamorphism on such binary trees mapping the tree to the set of all its elements.

```
def content(t: Tree[T]): Set[T] =
  t match {
    case Leaf =>
    case Node(l, i, r) => content(l) ∪ {i} ∪ content(r)
  }
```

Decision procedures for such recursive algebraic data types have been recently described [55], and it would be interesting to extend these procedures to synthesis.

[1] If t is of type `Option[A]`, then either t=`Some(a)` where `a` is of type `A`, or t = `None`

Chapter 10

Conclusion

Let us retrace the main steps we went through. We explained our concept of program synthesis in general arithmetic, and we illustrated it through various examples. After having identified three theories, namely linear rational arithmetic, linear integer arithmetic and parametrized linear integer arithmetic, we explained for each of them a corresponding optimized program synthesis algorithm in detail. Last but not least, we showed the power of such synthesis as part of a Scala compiler plug-in to demonstrate the future power of such synthesis.

Years of programming taught me that the more you program, the more you introduce bugs, and you learn from them. However, I am not convinced by the omnipresent program-debug method, it becomes exceedingly frustrating when the programs are huge. At some point, one might know what he wants, but if the displayed program does not correspond to her mental model of the problem anymore, it is harder and error-prone to try to maintain it.

My main motivation to begin and to keep working on this project was to move toward my personal goal of reaching the so dreamed zero-bug software development language. It would looks like an semi-automated multi-users context-aware and full of knowledge programming platform, where one would formulate her desires—not programs—in high-level natural language, refining it using more advanced vocabulary or more sentences, to obtain the desired program.

Appendix A

Derivation of Complexities

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

A.1 Linear Rational complexity

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

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

We will prove that:

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

where

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

where

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

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

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

A.1.1 Removing 1 equality

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

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

Therefore, we have the following relation:

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

A.1.2 Removing E equalities

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

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

Let us simplify the inner term:

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

Therefore, we have the following relation:

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

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

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

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

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

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

Complexity :

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

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

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

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

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

The expression $(\max(\ldots) + \min(\ldots))/2$ of section 4.1.1 is constructed, not computed, so this counts as $O(1)$.

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

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

Therefore, we have the following relation:

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

A.1.6 Merging and upper bound

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

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

A.1.7 Proof by induction

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

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

(4) The complete induction hypothesis let us assume that

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

.

Therefore, for $v = V - E$:

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

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

Using this result in (4), we obtain:

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

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

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

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

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

So by simplification, we obtain:

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

(3) The complete induction hypothesis let us assume that

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

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

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

Assuming that $K_5 \geq K_4$:

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

By bounding from above :

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

QED.

A.1.8 Size and execution time

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

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

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

This can be bounded from above by

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

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

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

A.2 Linear Integer complexity

To prove the complexity result presented in section 5.5.1 page 47, let us examine the number of times $\Omega(E,N,V)$ our solver goes back to 5.2, given the number of equalities E, inequalities N and output variables V.

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

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

where

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

By arithmetic properties, it implies the following expected result

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

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

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

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

By induction hypothesis, we obtain:

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

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

Now, equations are removed.

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

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

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

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

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

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

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

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

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

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

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

   So by induction :

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

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

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

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

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

   QED.

Appendix B

Bézout witnesses and base vectors

The following Scala functions are used to find witness to equations of the form $a_0 + a_1 \cdot x_1 + \ldots + a_n \cdot x_n = 0$, and to find $n-1$ base vectors to equations of the form $a_1 \cdot x_1 + \ldots + a_n \cdot x_n = 0$.

```
// Finds (x1, x2, k) such that x1.a + x2.b + gcd(a,b) = 0 and k = gcd(a ,b)
def advancedEuclid($a_{in}$: Int, $b_{in}$: Int):(Int, Int, Int) = {
  var (x, lastx) = (0, 1)
  var (y, lasty) = (1, 0)
  var (a, b) = ($a_{in}$, $b_{in}$)
  var (quotient, temp) = (0, 0)
  while(b != 0) {
      val amodb = (Math.abs(b) + a%b)%b
      quotient = (a − amodb)/b
      a = b
      b = amodb
      temp = x
      x = lastx−quotient∗x
      lastx = temp
      temp = y
      y = lasty−quotient∗y
      lasty = temp
  }
  if(a < 0)
    return (lastx, lasty, −a)
  else
    return (−lastx, −lasty, a)
}

// Finds coefficients x such that k∗gcd($a_{in}$) + x.$a_{in}$ = 0
def bezoutWitness(a: List[Int], k: Int):List[Int] = {
  var coefs = a
  var a_in_gcds = a.foldRight(Nil:List[Int]){
    case (i, Nil) => List(i)
    case (i, p::q) => gcd(p, i)::p::q
  }
  var result:List[Int] = Nil
  var last_coef = −1
  while(coefs != Nil) {
    coefs match {
      case Nil =>
      case 0::Nil =>
        result = 0::result
        coefs = Nil
      case el::Nil =>
        // Solution is −el/abs(el)
        result = (k∗(−last_coef ∗ (−el/Math.abs(el))))::result
        coefs = Nil
      case (el1::el2::Nil) =>
        val (u, v, _) = advancedEuclid(el1, el2)
        result = (−v∗k∗last_coef)::(−u∗k∗last_coef)::result
        coefs = Nil
      case (el1::q) =>
        val el2 = a_in_gcds.tail.head
        val (u, v, _) = advancedEuclid(el1, el2)
        result = (−u∗k∗last_coef)::result
        last_coef = −v∗last_coef
        coefs = q
        a_in_gcds = a_in_gcds.tail
    }
  }
  result.reverse
}

def bezoutWitnessWithBase(e: Int, a: List[Int]): (List[List[Int]]) = {
  var coefs = a
  var coefs_gcd = coefs.foldRight(Nil:List[Int]){
    case (i, Nil) => List(Math.abs(i))
    case (i, a::q) => gcd(a, i)::a::q
  }
  var n = a.length
  var result = List(bezoutWitness(a, e/coefs_gcd.head)) // The gcd of all coefs divides e.
  var i = 1
  var zeros:List[Int] = Nil
  while(i <= n−1) {
    val kii = coefs_gcd.tail.head / coefs_gcd.head
    val kijs = bezoutWitness(coefs.tail, coefs.head/coefs_gcd.head)
    result = (zeros ::: (kii :: kijs))::result
    coefs = coefs.tail
    coefs_gcd = coefs_gcd.tail
    zeros = 0::zeros
    i += 1
  }
  result.reverse
}
```

Appendix C

Abstract syntax tree

We present the abstract syntax tree for Parametrized Integer Linear Arithmetic. Note that our synthesizer accepts `Formula` expressions for input, which can contains only up to linear combinations of output variables. `Division` is only available in output, not as input. The other non-linearities available from the `InputTerm` expressions are authorized because `InputTerm` expressions form the coefficients of the output variables.

```
abstract class Expression
  abstract class Formula
    case class Conjunction(Formula∗)    // $f_1 \wedge f_2 \wedge ...f_n$
    case class Disjunction(Formula∗)    // $f_1 \vee f_2 \vee ...f_n$
    case class Negation(Formula)    // $\neg f_1$
    abstract class Equation
      case class EqualZero(Combination)    // $c = 0$
      case class GreaterEqZero(Combination)    // $c \geq 0$
      case class GreaterZero(Combination)    // $c > 0$
      case class False
      case class True
      case class Divides((InputTerm, Combination)    // $i|c$
  abstract class Term    // $t$
    case class Combination(InputTerm, (InputTerm, OutputVariable)∗)
            // $i_0 + i_1x_1 + ... + i_kx_k$
    case class Maximum(Term∗)    // $\max(t_1, ..., t_n)$
    case class Minimum(Term∗)    // $\min(t_1, ..., t_n)$
    case class Division(Term, InputTerm)    // $\frac{t}{i}$

abstract class InputTerm    // $i$
  case class InputCombination(Int, (Int, InputVariable)∗)    // $K_0 + K_1a_1 + ... + K_ka_k$
  case class InputAbs(InputTerm)    // abs($i$)
  case class InputAddition(InputTerm∗)    // $i_1 + i_2 + ... + i_k$
  case class InputDivision(InputTerm∗, InputTerm∗)    // $\frac{i_1i_2...i_k}{i'_1i'_2...i'_q}$
  case class InputMultiplication(InputTerm∗)    // $i_1i_2...i_k$
  case class InputGCD(InputTerm∗)    // $\gcd(i_1, i_2, ...i_k)$
  case class InputLCM(InputTerm∗)    // lcm($i_1, i_2, ...i_k$)
  case class InputMod(InputTerm∗)    // $\mod (i_1, i_2, ...i_k)$
```

Appendix D

Parametrized Linear Integer Synthesis Full Example

The input corresponding to the example in section 6.2 page 51 is the following, where x and y are the output variables.

$$\begin{aligned} 1 - 2x + ay \leq 0 \\ -1 + bx + 3y \geq 0 \end{aligned}$$

When we give our synthesizer this input, it outputs the following code. This code is not intented to be understood, because its meaning is represented exactly by the two equations above. This example shows some of the potential of writing code with synthesis.

Although this code is supposed to be correct by construction, it has also been tested in real. It also assumes that the bezoutWithBase, lcmlist and gcdlist functions are defined.

```
def PLISExample(a : Int, b : Int):(Int, Int) = {
  val (x, y) = if(−a > 0) {
    val x = 0
    val y = Math.max((−a)/(−a), 1)
    (x, y)
  } else if(a == 0) {
    val x = 1
    val y = ((3+(−b)∗x) − (3 + (3+(−b)∗x)%3)%3)/3
    (x, y)
  } else if({val k0 = Common.lcmlist(List(3,a))
      (a > 0) && ((0 to (−1) + Common.lcmlist(List(3,a))) exists {
        case k1 => { val k3 = Common.gcdlist(List(2∗(k0/a),
            (−1)∗Common.lcmlist(List(3,a))))
          val k2 = ((−k1) + ((−1)∗(k0/a)))/k3
          val List(List(k4,k5),List(k6,k7)) =
            Common.bezoutWithBase(1, 2∗(k0/a)/k3, ((−1)∗Common.lcmlist(List(3,a)))/k3)
            (((−k1) + ((−1)∗(k0/a))) % Common.gcdlist(List(2∗(k0/a),
              (−1)∗Common.lcmlist(List(3,a)))) == 0) &&
            ((k6∗(b∗(k0/3) + 2∗(k0/a)) > 0) || ((−k1) + ((−1)∗(k0/a)) +
            ((−1)∗(k0/3)) + k2∗k4∗(b∗(k0/3) + 2∗(k0/a)) >= 0 && k6∗(b∗(k0/3) +
          2*(k0/a)) == 0) || ((−k6)*(b*(k0/3) + 2*(k0/a)) > 0))} })}) {
    val k0 = Common.lcmlist(List(3,a))
    val (x, ya) = ((0) to ((−1) + Common.lcmlist(List(3,a)))) find { case k1 =>
      {val k3 = Common.gcdlist(List(2*(k0/a),(−1)*Common.lcmlist(List(3,a))))
       val k2 = ((−k1) + ((−1)*(k0/a)))/k3
       val List(List(k4,k5),List(k6,k7)) = Common.bezoutWithBase(1,
         2*(k0/a)/k3, ((−1)*Common.lcmlist(List(3,a)))/k3)
       (((−k1) + ((−1)*(k0/a))) % Common.gcdlist(List(2*(k0/a),
        (−1)*Common.lcmlist(List(3,a)))) == 0) && ((k6*(b*(k0/3) + 2*(k0/a)) > 0)
        || ((−k1) + ((−1)*(k0/a)) + ((−1)*(k0/3)) + k2*k4*(b*(k0/3) + 2*(k0/a)) >= 0
        && k6*(b*(k0/3) + 2*(k0/a)) == 0) || ((−k6)*(b*(k0/3) + 2*(k0/a)) > 0))}
    } match {
      case Some(k1) =>
      val k3 = Common.gcdlist(List(2*(k0/a),(−1)*Common.lcmlist(List(3,a))))
      val k2 = ((−k1) + ((−1)*(k0/a)))/k3
      val List(List(k4,k5),List(k6,k7)) = Common.bezoutWithBase(1, 2*(k0/a)/k3,
          ((−1)*Common.lcmlist(List(3,a)))/k3)
      val yb = if(k6*(b*(k0/3) + 2*(k0/a)) > 0) {
        val yb = (((−1) + ((−1)*((−k1) + ((−1)*(k0/a)) + ((−1)*(k0/3)) +
          k2*k4*(b*(k0/3) + 2*(k0/a)))) + k6*(b*(k0/3) + 2*(k0/a))) −
          ((k6*(b*(k0/3) + 2*(k0/a))) + ((−1) + ((−1)*((−k1) + ((−1)*(k0/a)) +
          ((−1)*(k0/3)) + k2*k4*(b*(k0/3) + 2*(k0/a)))) + k6*(b*(k0/3) +
          2*(k0/a)))%(k6*(b*(k0/3) + 2*(k0/a))))%(k6*(b*(k0/3) +
          2*(k0/a))))/(k6*(b*(k0/3) + 2*(k0/a)))
        yb
      } else if((−k1) + ((−1)*(k0/a)) + ((−1)*(k0/3)) + k2*k4*(b*(k0/3) +
        2*(k0/a)) >= 0 && k6*(b*(k0/3) + 2*(k0/a)) == 0) {
        val yb = 0
        yb
      } else if((−k6)*(b*(k0/3) + 2*(k0/a)) > 0) {
        val yb = (((−k1) + ((−1)*(k0/a)) + ((−1)*(k0/3)) + k2*k4*(b*(k0/3) +
          2*(k0/a))) − (((−k6)*(b*(k0/3) + 2*(k0/a))) + ((−k1) + ((−1)*(k0/a)) +
          ((−1)*(k0/3)) + k2*k4*(b*(k0/3) + 2*(k0/a)))%((−k6)*(b*(k0/3) +
          2*(k0/a))))%((−k6)*(b*(k0/3) + 2*(k0/a))))/((−k6)*(b*(k0/3) + 2*(k0/a)))
        yb
      } else { throw new Error("No solution exists") }
      val ya = k2*k5+(k7)*yb
      val x = k2*k4+(k6)*yb
      (x, ya)
      case None => throw new Error("No solution exists")
    }
    val y = ((−1+2*x) − (a + (−1+2*x)%a)%a)/a
    (x, y)
  } else { throw new Error("No solution exists") }
  (x, y)
}
```

Bibliography

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

[2] I. Asimov. *Gold: The Final Science Fiction Collection.* Eos paperback edition, 2003.

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

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

[5] C. Barrett and C. Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.

[6] A. Bes. BAPA was shown decidable in 1959 paper by Feferman and Vaught. Personal communication, November 2004.

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

[8] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Automated Reasoning*, 41:33–59, 2008.

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

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

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

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

[13] J. S. E. Bach. Algorithmic number theory.

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

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

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

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

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

[19] P. Granger. Static analysis of linear congruence equalities among variables of a program, 1991.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[21] M. Hofmann, A. Hirschberger, E. Kitzelmannn, and U. Schmid. Inductive synthesis of recursive functional programs. In *KI '07: Proceedings of the 30th annual German conference on Advances in Artificial Intelligence*, pages 468–472, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] A. A. Isil Dillig, Thomas Dillig. Cuts from proofs a complete and practical technique for solving linear inequalities over integers, 2009.

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

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

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

[26] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* 1993.

[27] E. Kitzelmann. Data-driven learning of functions over algebraic datatypes from input/output-examples. In P. Geibel and B. J. Jain, editors, *KI-2007 Workshop on Learning from Non-Vectorial Data*, volume 6 of *Publications of the Institute of Cognitive Science*, pages 36–45. Institute of Cognitive Science, Universität Osnabrück, 2007.

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

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

[30] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In *ICLP'93: Proceedings of the tenth international conference on logic programming on Logic programming*, pages 441–455, Cambridge, MA, USA, 1993. MIT Press.

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

[32] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 215–230, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] A. Lasaruk and T. Sturm. Weak quantifier elimination for the full linear theory of the integers: A uniform generalization of presburger arithmetic. *Appl. Algebra Eng., Commun. Comput.*, 18(6):545–574, 2007.

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

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

[36] M. Mayer, P. Suter, R. Piskac, and V. Kuncak. On Complete Functional Synthesis. Technical report, 2009.

[37] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.

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

[39] M. Moncur. The quotations page. `http://www.quotationspage.com`.

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

[41] T. Nipkow. Linear quantifier elimination. In *IJCAR*, 2008.

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

[43] Orocos. KDL. `http://www.orocos.org/kdl`.

[44] J.-L. Piedanna. AJL. `http://jlpfractware.free.fr/ajl.htm`.

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

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

[47] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Aritmethik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves, Warsawa*, pages 92–101, 1929.

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

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

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

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

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

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

[54] P. D. Summers. A methodology for lisp program construction from examples. New York, NY, USA, 1976. ACM.

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

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

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

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

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

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

Printed at EPFL - Lausanne - Switzerland

Mikaël Mayer
17 rue des champs Moré
90150 Menoncourt, France

`mikael.mayer.2005@polytechnique.org`