Interactive Programming by Example

THÈSE N^{O} 7956 (2017)

PRÉSENTÉE LE 28 SEPTEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ANALYSE ET DE RAISONNEMENT AUTOMATISÉS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Mikaël MAYER

acceptée sur proposition du jury:

Prof. P. Dillenbourg, président du jury
Prof. V. Kunčak, directeur de thèse
Prof. M. Vechev, rapporteur
Prof. R. Chugh, rapporteur
Prof. M. Odersky, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

Celui qui pose une question
aura l'air stupide cinq minutes.
Celui qui ne pose pas de question
restera stupide toute sa vie.
Proverbe chinois

He who asks a question
may look stupid for five minutes.
He who asks no questions
will remain stupid all his life.
Chinese proverb

To my lovely wife Marion who listened to me and gave me encouraging advice,
To my son Leoban for whom I want to be an interactive example,
To my parents Laurent and Christine who encouraged me to start this thesis,
To my brothers and sister – with whom I always share good times,
To my grandparents, my extended family, nephews, uncles, aunts, sisters-in-law,
To humanity, and even all living beings...

Acknowledgements

I thank the members of the jury for their time in preparing the defense of this thesis.
For the people I indirectly worked with, and whose conversations undoubtedly had an influence on my research, I would like to thank Philippe Suter, Eva Darulova, Giuliano Losa, Regis Blanc, Etienne Kneuss, Manos Koukoutos, Georg Schmid, Romain Edelmann, Daniel Lupei, Romain Rütschi and Ted Hart.
Three students took their time to develop these new technologies with me, so I would like to thank them there: Lomig Mégard, for the amazing work he did at enhancing Pong Designer; Thomas Dupriez, for his work on Web designer, which I had the opportunity to demo each time there was an EPFL open doors; and Elric Milet, for this work on abstracting websites.
My research has been made fruitful thanks to the collaboration of my co-authors, and I would like to extend a special thanks to them: Gustavo Soares, for our work on the Microsoft Prose Interface; Alexander Polozov, for his mentoring role in both my integration at Microsoft and for writing; Maxim Grechkin, for helping me with the statistical approach; Ravichandran Madhavan, for so many great projects on enhancing teaching; Jad Hamza, for the incredible time we spent together, even during breaks; Ruzica Piskac, for her pugnacity at defending what is true; Mark Santolucito, for his help on writing English; and for their support I would also like to thank Rishabh Singh, Filip Niksic, Mark Marron, Benjamin Zorn and Vu Le.
I also extend my gratitude to our administrative staff, Yvette Gallais, Sylvie Jankow and Fabien Salvi, who were always there to help me. I would like also to thank Holly Cogliati-Bauereis, for her proofreading of this thesis which was very valuable.
I would like to thank Sumit Gulwani, my manager and mentor for a full, intensive and productive year at Microsoft Research, which changed my life. He helped me to develop a real vision about customers of these kinds of technologies, mentored me so well that I knew how to continue my thesis.
Last but not least, this thesis would have never probably see the day without the tremendous daily collaboration and encouragement provided by my thesis director, Viktor Kunčak. I am grateful to him for sharing his visions and continuously providing feedback and pointers to advance this thesis. Viktor is very passionate and knowledgeable, and it is always delightful to converse with him.
I acknowledge the financial support of the European Research Council.

Lausanne, 26 April 2017 M. M.

Abstract

As of today, programming has never been so accessible. Yet, it remains a challenge for end-users: students, non-technical employees, experts in their domains outside of computer science, and so on.

With its forecast potential for solving problems by only observing inputs and outputs, programming-by-example was supposed to alleviate complex tasks requiring programming for end-users. The initial ideas of macro-based editors paved the way to subsequent practical solutions, such as spreadsheet transformations from examples.

Finding the right program is the core of the programming-by-example systems. However, users find it difficult to trust such generated programs. In this thesis, we contribute to proving that some forms of *interaction* alleviate, by having users provide *examples*, the problem of finding correct and reliable *programs*.

We first report on two experiments that enable us to conjecture what kind of *interaction* brings benefits to programming-by-example. First, we present a new kind of game engine, Pong Designer. In this game engine, by using their finger, users program rules on the fly, by modifying the game state. We analyze its potential, and its eventual downsides that have probably prevented its wide adoption. Second, we present StriSynth, an interactive command-line tool that uses programming-by-example to transform string and collections. The resulting programs can also rename or otherwise manage files. We obtained the result that confirms that many users preferred StriSynth over usual programming languages, but would appreciate to have both.

We then report on two new exciting experiments with verified results, using two forms of interaction truly benefiting programming-by-example. Third, on top of a programming-by-example-based engine for extracting structured data out of text files, in this thesis we study two interaction models implemented in a tool named FlashProg: a view of the program with notification about ambiguities, and the asking of clarification questions. In this thesis, we prove that these two interaction models enable users to perform tasks with less errors and to be more confident with the results. Last, for learning recursive tree-to-string functions (e.g., pretty-printers), in this thesis we prove that questioning reduces the learning complexity from a cubic to a linear number of questions, in practice making programming-by-example even more accessible than regular programming. The implementation, named Prosy, could be easily added to integrated development environments.

Acknowledgements

Key words: Programming by example, programming by demonstration, program synthesis, accessibility, domain-specific languages, active learning

Résumé

Aujourd'hui, la programmation n'a jamais été autant à portée de tous. Cependant, de nombreux utilisateurs rechignent encore à l'utiliser, comme des étudiants, des employés de domaines non techniques, des experts en dehors de l'informatique, et ainsi de suite.

Avec sa vocation à résoudre les problèmes rien qu'en recevant des exemples de ce qui rentre et ce qui doit sortir, la programmation par l'exemple était censée mettre des tâches complexes demandant de la programmation à portée de main tous les utilisateurs. Les idées initiales des éditeurs de texte basés sur les macros ont ouvert la voie à des solutions pratiques ultérieures, telles que les transformations de tableurs à partir d'exemples.

Le cœur de la programmation par l'exemple, c'est de trouver le programme correct. Cependant, les utilisateurs peinent à se fier à de tels programmes générés. Cette thèse contribue à prouver que certaines formes d'*interaction* simplifient le problème de trouver des *programmes* corrects et fiables quand il s'agit de tirer profit d'*exemples*.

Tout d'abord, nous rendons compte de deux expériences qui nous ont permis de conjecturer quelles étaient les formes d'*interaction* qui pourraient améliorer la programmation par l'exemple. En premier lieu, nous présentons un nouveau genre de moteur de jeu, Pong Designer. Dans ce moteur de jeu, les utilisateurs programment des règles à la volée, en modifiant l'état du jeu avec leur doigt. Nous analysons son potentiel et finalement ses inconvénients qui l'ont probablement emêché d'être populaire. Deuxièmement, nous présentons un outil interactif en ligne de commande, StriSynth, qui utilise la programmation par l'exemple pour transformer des chaînes de caractères et des collections. Entre autres, ces transformations peuvent aussi renommer ou autrement gérer des fichiers. Nous avons obtenu le résultat que beaucoup d'utilisateurs ont préféré StriSynth à des langages de programmation conventionels, mais qu'ils apprécieraient d'avoir les deux.

Pour la suite, nous rendons compte de deux expériences innovantes aux résultats éprouvés, montrant à quel point deux nouvelles formes d'interaction peuvent améliorer quantitativement la programmation par l'exemple. Troisièmement, par-dessus un extracteur de données structurées sur un texte brut, extracteur fonctionnant en programmation par l'exemple, nous étudions dans cette thèse deux modèles d'interaction implantés dans un logiciel nommé FlashProg : une vue du programme avec des notifications à propos des ambiguités, et une fenêtre où l'ordinateur peut poser des questions pour clarifier certaines extractions. Dans cette thèse, nous prouvons que ces deux nouveaux modèles d'interaction permettent aux utilisateurs d'accomplir des tâches avec moins d'erreurs, et d'exprimer une plus grande confiance envers les résultats que s'ils n'avaient pas ces

modèles d'interaction. En dernier lieu, quand il s'agit à l'ordinateur d'apprendre des fonctions récursives transformant des arbres en chaînes de caractères, par exemple pour afficher une représentation claire de données lors d'un déboguage, cette thèse prouve que l'interaction sous la forme de questions casse la complexité d'un nombre de question passant de cubique à linéaire, ce qui fait que la programmation par l'exemple devient même plus accessible que la programmation habituelle pour ce genre de tâches. L'implantation, nommée Prosy, pourrait facilement s'ajouter à des environnements de développement intégrés.

Mots clefs : Programmation par l'exemple, programmation par la démonstration, synthèse de programmes, accessibilité, langages spécifiques à un domaine, apprentissage actif

Contents

List of Figures

List of Tables

1 Introduction

"Your wish is my command"
Anonymous traditional arabo-persic genie

"Vos désirs sont mes ordres"
Génie traditionel arabo-perse anonyme

Programming is the translation from abstract wishes to concrete instructions executable by the computer. In other words, programming is about closing the *gap* between what users want and what computers execute.

High-level programming languages partially fill the gap from abstract wishes to concrete instructions, building on top of low-level instructions. Functional programming, object-oriented programming, library management, model-based system design, and many other constructs empower users to think at a higher level. Furthermore, programming languages are increasingly accessible for those who want to learn. There are plenty of free online tutorials, open classrooms, massive online open courses (MOOCs) and social communities dedicated to the learning of a language, e.g. [Resnick et al., 2009]. Specific programming languages are also very rewarding job skills[1].

Specifications are another step to building the bridge between abstract wishes and high-level programs, often building on top of programming languages. By formally specifying the expected behavior of a program, users might eliminate crashes and bugs; and in many cases verify that programs perform as intended. Without specifications, costly crashes can occur. In 1985, people died from the radiotherapy machine Therac 25 [Atomic Energy of Canada Limited, 1985]. In 1996, millions of dollars were lost in the launching of Ariane 5 [European Space Agency, 1996]. In 2011, a lot of soldiers were not correctly paid with Louvois [Ministère de la Défense Française, 2011]. Programming with specifications [Suter, 2012] has gained in popularity and is at the heart of many successful businesses [Vigyan Singhal, 1999, Cousot et al., 2001, Distefano and Calcagno, 2009, Synopsys, 2016]. Specifications often reflect the abstract wishes more than the

[1] https://techbeacon.com/best-paying-programming-languages

code. For example, we could specify that a list sorting algorithm is correct by

```
def sort(in: List) = ... ensuring { out ⇒ isSorted(out) && content(in) == content(out) }
```

which is much closer to the wish of "a sorted list" than any implementation.

As users are not yet intimately linked to computers, there is no formal known intermediate step between abstract wishes and specifications. Users fill this gap by a form of organizational and algorithmic thinking, identifying how to decompose the needs into constraints for the computer. We admittedly suppose that, to some extent, abstract wishes can serve to decide if the specifications are coherent or not.

For example, let us consider a usual workflow from abstract wishes to computer instructions. First, we start with an abstract wish:

```
"My university wishes to select the fifty best candidates for a prize.
```

then after thinking about how to transform this wish to specifications, provided a list of students named 'in' we might write:

```
size(out)=50, (∀ s ∈ out, ∀ student ∈ in \ out, s.score ≥ student.score) and out ⊂ in
```

At this point, some algorithms might help to derive the following high-level program, or we could find it manually:

```
val sorted = in.sortBy(_.score)
sorted.take(50)
```

and then we compile it to some executable code understandable by the computer. Below is the beginning of such code produced for the Java virtual machine (JVM), in three aligned formats: binary code, assembly, paraphrased version.

```
0010 1011 1011 1011 0000 0000 0000 1100 0101 1001 0010 1010 1011 0111...
[aload_1] [  new  ] [       12        ] [  dup  ] [aload_0] [invokesp...
  load      create a score extractor    duplicate   load     initialize
   in       referenced at address 12       it      context    extractor
```

The remaining steps are: "create an ordering for integers, call the sort method, push 50, call the take method, return the result". Note that calling methods consume their arguments.

We can trust a computer for two out of these three steps. The compilation from programs to executable code rarely produces bugs [Le et al., 2015], and there is no error at all when using certified compilers [Leroy, 2006]. Furthermore, modern theorem provers can automatically prove that programs meet their specifications [Kuncak, 2007, Piskac et al., 2014, Kuncak, 2015] formally prove that programs have an absence of run-time errors [Cousot et al., 2001, Jourdan et al., 2015], or even suggest corrections [Chandra et al., 2011]. Even better, recent research in program synthesis suggests that a computer can itself translate specifications to programs [Solar-Lezama et al., 2007, Srivastava et al., 2010, Kuncak et al., 2010b, Kneuss et al., 2013, Alur et al., 2013, Kneuss et al., 2015, Reynolds et al., 2015, Koukoutos et al., 2016]. It appears that only the translation of abstract wishes to specifications is error-prone. If the latter are not complete enough,

or if they do not match the abstract wishes, there is almost no hope for the remaining pipeline to eventually produce a correct program.

Although we cannot measure the internal thinking of users to see if specifications match their abstract wishes, we believe that we can still alleviate the translation of abstract wishes to specifications, by letting computers interactively provide feedback to users.

1.1 Examples as Ambiguous Specifications

In this thesis, we focus on specifications that have the form of input/output examples [Myers, 1987, Cypher, 1991, Lieberman, 2000, Lieberman, 2001, Ruvini, 2004, Gulwani, 2011, Zhang and Sun, 2013, Kini and Gulwani, 2015, Feser et al., 2015, Frankle et al., 2016], rather than mathematical formulae that entirely describe a problem. Examples are naturally under-specified mathematical formulae, and are representative of the whole set of inputs. Furthermore, as examples are tied to the kind of data programs operate on, and not from the program structure itself, they are often easier to reason about. Note the difference between programming-by-example and programming-by-demonstration. In the state of the art, programming-by-demonstration usually consists in providing detailed steps on *how* to produce the output, whereas programming-by-example usually consists in providing only the output. However, the difference is thin, and for the remaining of this chapter, we are using only the wording of programming-by-example.

For example, it is natural to specify how to rename files using examples, rather than using formal specifications. Compare a formal complete specification:

```
def rename(file: String, index: Int) = {
  ...
} ensuring { result ⇒
  result == "%03d".format(index+1) +
            " " +
            "^(\\w)".r.findFirstIn(file).get +
            "^\\w+ (\\w)".findFirstIn(file).get +
            ".png"
}
```

with an example-based specification:

```
def rename(file: String, index: Int) = {
  ...
} ensuring { result ⇒
  file → index → result passes {
  case ("Mikael Mayer.PNG", 0) ⇒ "001 MM.png"
}}
```

The second one feels more natural than the first. The first specification is merely computational, and might be even wrong, for example, if we wanted to extract all initials,

not just the first two.

Most programming-by-example engines rely on prior knowledge about the shape of programs [Alur et al., 2013], if not all engines. A (context-free) grammar usually determines the shape of programs, its rules dictate how to combine them. Let us take as example the task of renaming files. The grammar might define a program as being either (1) a constant, or (2) an extraction of the input that uses regular expressions, or (3) a concatenation of two programs. The programs satisfying a given input/output example pair are thus either (1) the constant program returning the output, or (2) if the output is present in the input using a regular expression, an extraction of the input using this regular expression, or (3) a combination obtained after recursively learning programs producing complementary parts of the output. For (3), we split the output string O into two non-empty strings A and B such that $O = A + B$, and we recursively solve the sub-problems of the input/A and input/B example pairs (this procedure is sometimes called witness function). Due to the grammar, we can ensure that the first program cannot be a concatenation, to avoid duplicates. Having a set of candidates for the first program, and a set of candidates for the second program, we combine them efficiently using a "join set" computing the cartesian product lazily. Similarly, using different decompositions of the output string, we obtain different join sets which we can put in a "union set". Join and union sets form a version space algebra (VSA) [Lau et al., 2000, Lau et al., 2003a, Singh, 2016]. By caching the computation of some of these steps [Polozov and Gulwani, 2015], this version spaces can efficiently represent sets of billions of programs without enumerating them all. Given multiple input/output examples, we can intersect the resulting program sets from each of the examples. Intersections of version spaces sets can be computed faster than the intersection of plain sets, up to an exponential ratio. Finally, to obtain only one program, we select the best one according to heuristics that provide a score to all programs.

Software offering programming-by-example mostly failed until recently [Lau, 2009]. One of the reasons is that a learning algorithm might fail to learn the correct program because of ambiguous specifications. Failing to translate specifications to correct programs means that the former are either incomplete, or that the translation itself should be improved. Improving the translation of example specifications to programs using existing knowledge [Gulwani, 2013] and machine learning [Singh and Gulwani, 2015] is possible up to a certain level. However, such models cannot account for all possible scenarios. It remains that the user is responsible for correctly translating abstract wishes to specifications. It is hence challenging to ensure that the examples accurately reflect the wishes. To avoid errors, the computer can provide feedback to "encourage trust by presenting a model users can understand" [Lau and others, 2008], so that users can act on it. We identify three kinds of orthogonal feedback that can help users to better perform a translation from their abstract wishes to specifications: the output feedback, the program feedback and the specification feedback.

- The output feedback is arguably the simplest feedback for programming-by-example, but it is still useful. It consists in showing the output of the generated programs, so that if these outputs are not correct, the user might infer the need of more specifications [Smith et al., 2000, Yessenov et al., 2013].

- The program feedback consists in showing the generated programs to users, so that if users detect a wrong piece of code, they can refine the specifications [AgentSheets, Inc., 2017].

- The specification feedback consists in giving feedback on the specifications written by the user, by suggesting to add, remove or modify specifications. In [Gulwani, 2011], Gulwani shows that we can spot potential errors in the specifications of users, based on the surrounding specifications.

1.2 Output Feedback

The output feedback is an indirect feedback. The computer runs the program on several other or subsequent inputs, so that the user can detect wrong outputs.

For example, suppose that we have an engine enabling us to program a game by demonstration. At some point, we add an integer of value 0 to the game. After selecting a collision between the ball and a block, we change the value of the integer to 1, because it represents a score to increase each time there is a collision between the ball and the block. Then we resume the game. After a second collision of the ball with this block, the score is still at 1. This might indicate that the specification was not understood correctly because it conflicted with the abstract wish that the score should be 2. By selecting the collision event and changing the value of the score to 2, the system can correctly infer the rule of incrementation, and successive collisions can increment the score. Viewing the result of the rules, or the output feedback, enabled us to correct the behavior.

Another possibility consists of renaming a file. By showing the user how the subsequent files are to be renamed, this output feedback provides the user a quick feedback, and eventually confidence – or not ! – about the precision of the specification.

1.3 Program Feedback

The program feedback is a more direct kind of feedback. The computer exhibits its findings so that users might inspect it. If some piece of code does not reflect what users had in mind, it supposedly helps them to refine their specifications by further providing suitable input/output examples.

The program feedback might be useful for larger sets of inputs, because it might not

always be possible to present all outputs. Furthermore, even if a program is valid on a visible subset of the inputs, it will not always be correct on all inputs. We might also want to reuse the program, customize or maintain it. A specialized kind of program feedback also shows what the subparts of the program are doing. For example, if a regular expression appears in a generated program, the computer could display where this regular expression matches the input.

Another way of providing feedback on programs is to paraphrase it. Without paraphrasing, engineer-designed languages are often cryptic [2]. Paraphrasing helps to understand the program at a glance, and should not be too low-level [Lau, 2009]. Instead of

```
Concat(Counter(1,3), Concat(Const(" "),
Concat(Substring(Position("","[A-Z]",1),Position("[A-Z]","",1)),
Concat(Substring(Position(" ","[A-Z]",-1),Position(" [A-Z]","",-1)),
  Const(".png")))))
```

we would arguably prefer the following more readable paraphrased representation:

> A 3-digit counter starting at 1 + the string " " + the first uppercase letter + the last uppercase letter after space + the string ".png"

Some frameworks allow for the creation of programs by using natural language [Le et al., 2013, Gulwani and Marron, 2014, Gao et al., 2015, Gvero and Kuncak, 2015] or quasi-natural language [Androutsopoulos et al., 1995, Popescu et al., 2004, Myers et al., 2004, Wang et al., 2017]. However we are currently not aware of any tool that enables both paraphrasing and the edition of the paraphrasing. In the loop of programming-by-example, program feedback is useful for the user merely to indicate what kind of examples he needs to add.

1.4 Specification Feedback

"Programming is hard" [Suter, 2012], especially reading programs that a person did not write himself. Indeed, the feedback of reading programs is not always sufficient for users to recognize and to compare against their abstract wishes "Yes, that is the right thing to do". Not only might the program be difficult to read, despite that it is paraphrased, but when there are many similar programs matching the input/output examples, the user might be uncertain about which one is correct.

From the previous example, the following program is valid and equivalent for the same input (bold added for the difference). Choosing which of the two programs is correct, or even which parts of them are, highly depends on the context.

[2] powershellmagazine.com/2014/09/09/using-the-convertfrom-string-cmdlet-to-parse-structured-text

> A 3-digit counter starting at 1 + the string **between the first two words** + the first uppercase letter + the **second uppercase letter** + the **lowercased string starting at the first dot**

Users usually need programs because they have many inputs to process, some of which are sometimes already available. In any case, users might be willing to manually provide outputs for only a few of the inputs. Exploiting other inputs, the computer can start to provide feedback at the specification level, by suggesting to add or modify examples.

Let us suppose that there is a second file to rename, for example "Leonardo Da Vinci.JPG". The original program in the previous section will produce "002 LV.png". In order to be able to suggest new specifications to add, the computer can take the programs found in the previous step (which all have the same behavior on the first file), and compare their output when it runs them on this new file. As there can be billions of programs, the computer might select a subset of the programs, according to some heuristics. From the original program, we present some variants that produce the same output for the first example, but differ for the second file:

The first output "001 MM.png" is produced from the first input "Mikael Mayer.PNG" by:	Output for second input "Leonardo Da Vinci.JPG"
the original program: A 3-digit counter starting at 1 + the string " " + the first uppercase letter + the last uppercase letter after space + the string ".png"	"002 LV.png"
a variant where the digits ("[**001**] MM.png") could be...	
...a constant string	"00**1** LV.png"
...the index concatenated with "01"	"**101** LV.png"
a variant where the space ("001[ ]MM.png") could be...	
...the string between the end of the first word and the start of the penultimate word	"002 **Da** LV.png"
a variant where the first letter ("001 [**M**]M.png") could be...	
...a constant string "M"	"002 **M**V.png"
...the second uppercase letter	"002 **V**V.png"
...the fifth uppercase letter from the end	"002 **D**V.png"
...the string between the end of the first space and the start of the last lowercase word	"002 **Da V**V.png"
a variant where the second letter ("001 M[**M**].png") could be...	
...a constant string "M"	"002 L**M**.png"
...the first uppercase letter	"002 L**L**.png"
...the second uppercase letter	"002 L**D**.png"

... the string between the end of the first space and the start of the last lowercase word	"002 L**Da** V.png"
a variant where the two letters ("001 [**MM**].png") could be...	
... the sequence of all capitals letters which are followed by a lowercase word	"002 L**D**V.png"
a variant where the extension ("001 MM[**.png**]") could be...	
... the lowercased string appearing after the first dot	"002 LV**.jpg**"

Note that with only the variants above, there are $3 \times 2 \times 5 \times 5 \times 2 + 3 \times 2 \times 2 = 312$ valid programs, each one likely to return a different result compared to any other program. By offering only the suggestions of the second column, users might choose with one click which one is correct. With the right heuristics, there is a high probability that the correct output is among them. Using this specification feedback from yet unhandled inputs, users can thus expect to find the correct program much faster and with better reliability.

1.5 Outline of the Dissertation

This thesis focuses on interaction techniques that make programming-by-example more trustful and more reliable than it was before. More precisely, we define what the most efficient interactions are. The remaining of this thesis is organized as follows:

Chapter 2 presents a Game Engine (Pong Designer) offering the ability to program games by demonstration using touch. Its novel architecture enable users to go back in time up to five seconds in the game, to edit the game state on-the-fly and to add rules when they select an event before editing the game. Furthermore, the rules thus produced are part of the game itself, and users can modify its constants. If the rule the system inferred is not the correct one, the user can select the corret alternative. We discuss the benefits of such a game engine.

Chapter 3 introduces StriSynth, a tool that can synthesize code from examples for file management tasks. We design algorithms for learning how to rename, filter and partition files by example. Because StriSynth is embedded in a programming language Scala, it offers the possibility of composing programs and applying them to collections. StriSynth can synthesize code on a variety of benchmarks in reasonable time, often faster than a programmer could do. We also asked users to compare StriSynth with a 'conventional' programming language (PowerShell) for some specific tasks, and we show that there are some tasks for which StriSynth is highly preferable.

Chapter 4 presents FLashProg, a tool for interactively extracting structured data from raw text using examples. The interface offers to highlight the text to be extracted

with the mouse, which is intuitive. From a user study, we draw the statistically significant result that two novel interaction models cause users to gain trust in the results and make less errors than without them. The first interaction model is to display a rich version of the program, whereas the second interaction model is to ask clarification questions, We further demonstrate that users prefer clarification questions over program investigation.

Chapter 5 constructively proves several theoretical results that make questioning a first-class citizen in Prosy, a programming-by-example-based engine to learning recursive tree-to-string functions. As we progressively enhance an algorithm for learning these functions, we prove that, without interaction, finding such a recursive function is an NP-complete problem. With one step of interaction, where the computer can select the examples to provide, we demonstrate that an upper bound on the number of examples is of the order of $O(n^3)$, matching for the first time the lower bound found 20 years ago. Furthermore, the learning algorithm obtains a polynomial-time complexity. Finally, with an unbounded number of interaction steps, we demonstrate that the computer requires only a linear number of questions, making this sort of interactive programming-by-example technically as precise and as fast, if not faster, as normal programming.

Chapter 6 presents a qualitative comparison of the contributions of Chapters 2 to 5 on two scales. The first scale compares how each of the four previous chapters address the needs of users, according to the answers they provide to the following six observation questions:

- Are tasks conducted faster and/or easier using Programming by Example than just programming?
- Do users need to see and understand the generated programs?
- Do users need to tweak, modify or reuse the generated programs?
- Can displaying the generated program in a paraphrased form simplify its understanding?
- Do users trust more the results of programming-by-example when the computer provides the generated program than when it does not?
- Is programming by example is more reliable/faster when the computer asks questions?

We present partial answers to these questions, along with a short analysis at the end of each chapter (pages 44, 76, 106, 161). We also discuss and compare the utility of the tools of the four previous chapters by using three qualitative criteria related to the title of this thesis:

- Interactivity: Is there feedback, or is there an effort from the computer to understand the user's goal?

- Programmability: Is the program modifiable? Is the programming language expressive?
- Exemplarity: Are examples well suited for the task? Do they reflect the goal?

We also discuss the future work for each of the previous chapters, and in general.

Chapter 7 concludes this thesis.

In Chapters 2 to 5, the alphabetical numbering of sections (A., B., C.) denote sections not part of any published papers.

2 Game Programming by Demonstration

"Une chose intelligente, c'est une chose que n'importe quel imbécile peut comprendre."

"A clever thing is something that any fool can understand"
Georges Wolinski

A. The Ideal Goal of Self-Reconfiguring Interfaces

The main problem in developing most desktop software is that it requires a cycle of writing code / compiling / testing. What can we do if we want to modify the interface on-the-fly, while using it? Can we eliminate the cycle and do programmatic editing while using the interface?

After thinking about it for a while, we decided to implement this idea as the form of a game engine, where the user can modify the game itself on-the-fly, while playing, and using programming-by-example editing.

For this, we first thought about the process of demonstrating game rules, such as selecting an event and demonstrating the result. The game engine would run the program that describes the rules of the game. After receiving rule demonstrations, the game engine would record ambiguities. We also wanted to be able to recreate all possible classic games easily. Pong Designer was born.

Game Programming by Demonstration

The following pages 12-42 are the content of the peer-reviewed paper named "Game Programming by Demonstration" (DOI: 10.1145/2509578.2509583) that appeared in Onward! 2013 and published by ACM. I reproduced it with the permission of the authors, who are myself and Viktor Kuncak. In addition to the original appendix, the appendix B.1 page 187 contains a tutorial for creating a space-invaders like game.

Contributions: *All the software presented in this paper is my own contribution.*

The increasing adoption of smartphones and tablets has provided tens of millions of users with substantial resources for computation, communication and sensing. The availability of these resources has a huge potential to positively transform our society and empower individuals. Unfortunately, although the number of users has increased dramatically, the number of developers is still limited by the high barrier that existing programming environments impose.

To understand possible directions for helping end users to program, we present Pong Designer, an environment for developing 2D physics games through direct manipulation of object behaviors. Pong Designer is built using Scala and runs on Android tablets with the multi-touch screen as the main input. We show that Pong Designer can create simple games in a few steps. This includes (multi-player and multi-screen) Pong, Brick Breaker, Pacman, Tilting maze. We make available Pong Designer as well as several editable games that we created using it. This paper describes the main principles behind Pong Designer, and illustrates the process of developing and customizing behavior in this approach.

2.1 Introduction

Smartphone and tablet devices have dramatically increased the number of individuals with access to computing resources. The availability of these resources has an enormous potential to positively transform our society. Unfortunately, using traditional development methods for such devices is at least as difficult than for desktops, and possibly more difficult due to new constraints on device input size, energy consumption, and the complexity of the software stack. Furthermore, the benefits of these platforms can be fully realized only by specialization of applications to particular domains, or even particular users. We would like to enable domain experts that are not software developers by training to develop applications that support well their domain activities. Many educational, scientific, engineering and artistic domains would benefit from such end-user programming. For tasks such as scripting and home automation, we would like to deliver personalized applications at the low price that makes current phone applications accessible. To achieve this level of specialization, the number of developers needs to be much closer to the number of users. A promising approach to realize these goals is to empower users themselves to program, blurring the traditional divide between professional software developers and end users. This direction is especially appealing as the increasing computing capabilities of these ubiquitous devices enable more advanced run-times and software development tools, and as new algorithmic techniques enable automated programming assistance and synthesis [Lau et al., 2003a, Gvero et al., 2013, Shacham et al., 2009, Gulwani, 2012a, Singh and Gulwani, 2012b].

One of the challenges when programming using conventional text-oriented editors is the disconnect between program representation and its effect during execution. Several recent approaches support understanding the effect of a line of code with an enhanced editor [Nasilowski, 2011] or with very high-level constructs such as behaviors and constraints [Scirra, 2013], [Stolee and Fristoe, 2011]. Others prefer to guide the programmer by providing code structures that can fit together, either modifiable during the game simulation [Resnick et al., 2009] or in a special structured editor [Scirra, 2013], [Abelson and McKinney, 2010]. These approaches reduce the burden of syntax checking. However, few of them provide a way to directly modify the running application by demonstrating the desired behavior using examples.

Recently, programming by demonstration has been revisited and shown very successful in domains such as spreadsheets [Gulwani, 2012a, Singh and Gulwani, 2012b], which map inputs to outputs. We wish to understand the potential of programming by example on a new generation of touch-enabled devices and apply it to more complex domains, containing interactive behavior. This led us to the domain of graphical games running on tablets.

2.1.1 Contributions

This paper presents a development approach for graphical games where developers use demonstration to describe not only application state, but also its behavior. The developer can pause the game and directly manipulate objects to demonstrate the desired effects through examples. The key aspects of our approach are the following:

- An on-the-fly editing principle: the users can pause, rewind, and modify a running game using a time progress bar, with graphical access to events from the past.
- Rule demonstration: a rule-based execution model, where developers dynamically create and update rules using concrete demonstrations on objects. The system automatically infers candidate rule conditions and actions from such demonstrations.
- A freely available working Scala/Android implementation that leverages these principles on runs on devices enabled with multitouch and accelerometer input. The system and several examples of (editable) games are freely available as “Pong Designer” app from Play Store, as well as at http://lara.epfl.ch/w/pong/ .

2.2 Building Games by Demonstration

We illustrate the flavor of game development in Pong Designer by showing how to develop several games using a remarkably small number of steps on a tablet computer (we used Asus Eee Pad Transformer for our experiments).

We first show how to build a brick breaker (Breakout-style) game. We then present the process of building a variant of Pacman with moving camera and accelerometer input.

The Appendix B.2 presents an additional example and screen shots.

2.2.1 Breakout-Style Game

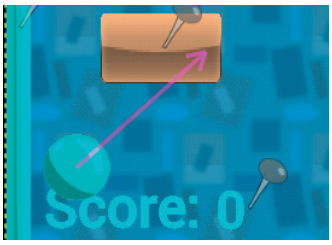
This example illustrates the use of time slider and demonstration of behavior in response of actions.

Suppose that we wish to program a classic Breakout-style game, where the goal is to drive the bouncing ball, using a sliding paddle to ensure it does not escape the screen at the bottom, and aiming to ensure that the ball breaks all the bricks on the screen.

Describing the initial state. We begin by creating the game objects, using predefined shapes. This process resembles drawing in a simple vector graphic design application, such as the ones used to create conference presentation slides. To introduce shapes into the playing field or modify their properties we use buttons from an on-screen toolbar:

The buttons provide a way to increment or decrement a number, set up the velocity and angle value of any shape, whether the shape can move or not as well as its visibility, size and color. Setting up the game layout plays the role of defining data structures and objects in a conventional process. Figure 2.1 shows a possible result for our example, which includes the walls, bricks, the ball, and the paddle.

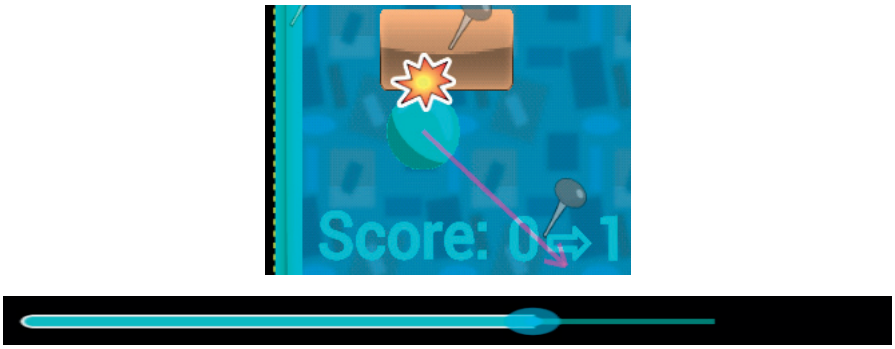
Setting dynamic properties such as velocity. In contrast to a drawing program, the objects we created are in fact models of physical objects with associated behavior in a two-dimensional physics world. Objects can be either non-movable (pinned to the ground), or movable according to Newton's inertial laws with friction. By default, all objects have zero initial velocity. If we select the ball object, we can change its speed by moving the endpoint of the velocity vector displayed on screen. In this example, we set the initial velocity of the ball to move towards a nearby brick:



Starting the game. Pressing the "play" button starts the game, running the physics simulation from the current state and displaying the outcome in real time on the screen (in this case, the ball moves towards the brick). As the game runs, we observe that the time bar at the bottom of the screen makes progress, indicating the passage of time, much like when playing a music or a video stream.

During this time the system silently records internal events, such as object collisions, as well as user input, including user's movements on the multi-touch screen.

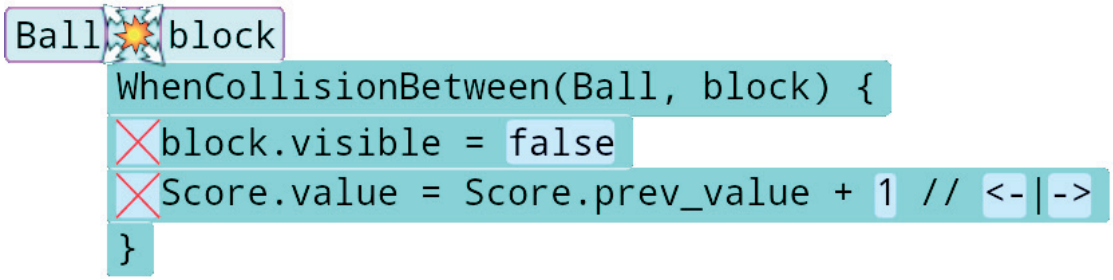
Identifying events of interest. In our example we observe the ball hitting the brick and bouncing off it, with the brick staying intact. We would like to change this behavior to ensure that that, when the ball hits a brick, the score increments and the brick disappears. We press the "Pause" button to stop the simulation, which allows us to again edit objects in the last simulated state. This time, however, the game also contains the history of past events. Pressing the events button displays the events using appropriate graphical metaphors. The following representation shows that there was a recent collision between the brick and the ball:



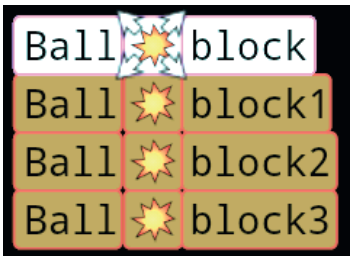
Navigation in time and space to select events. Note that the number of events recorded can be large, but the user can navigate them using the fact that an event is indicated near the relevant objects, and at the relevant time. The user uses the time bar to go back in past to the approximate point in time when the event of interest occurred and then selects the event. In our example, the user chooses the collision event between the ball and the brick.

Describing actions: breaking a brick. The selected event represents a condition for triggering a rule. The entire game is governed by such event-triggered rules. To specify

the action that should take place in case of a given event, the user simply performs the desired action in the game editor. This approach subsumes macro recording present in editors, but, unlike simple macro recording, it is followed by a crucial *generalization* step. In our current example, we would like to indicate that the brick should disappear in case of a collision with the ball. To do this, after selecting the collision event, we simply move the brick outside the visible screen, or set its visibility to *invisible*. We also increment the score counter. After pressing the OK button, the system automatically generates the corresponding rule.



The user can edit and delete some of its parts, if desired, or simply accept it as it is. The rule will now be applied whenever the ball hits this brick. If we copy any object, the system copies the rule along with the objects to which they apply. In this case we first create one brick and the rule. When duplicating the brick, the system will duplicate the rule. (This corresponds to a prototype-based object system; we are currently adding support for classes of objects.)



Second rule: moving the paddle. We next wish to specify that the paddle follows the horizontal movements on the touch screen when they occur on the paddle. To do this, we run the game and attempt to move the paddle. The paddle does not respond, but the movement is recorded in the event history. We can then use the recorded movement to correct the existing behavior as follows. We pause the game and move the time slider towards the point where we made the movement on the screen. The movements are displayed using curves that describe the path traced on the screen, as in the following movement to the right:

We select the move event, which acts as the condition of our rule. As the action for the rule, we demonstrate the corresponding change in the position of the object, moving it from the initial position

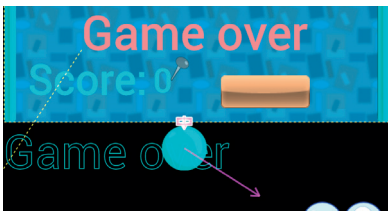
into the final position:

Because of a "snapping-to-position" feature, this movement is recorded as a perfectly horizontal movement. At this point the system performs a *generalization* from the concrete demonstration and generates a rule that applies to these two objects whenever rule conditions are met. Note that, even if the two movements demonstrated were not perfectly identical, the system finds that they are sufficiently close. It therefore derives the following rule, which matches the location of touch event to the horizontal movement of the paddle:

```
paddle
        WhenFingerMovesOn(paddle) { (xFrom, yFrom, xTo, yTo) =>
        paddle.x = paddle.prev_x + xTo - xFrom // <-|->
        }
```

In general, our system uses a set of parametrized templates to compute a set of possible actions that could explain the demonstrated state change. If the user expands the generated rule, they are able to delete lines and select the intended result from the templates by pressing arrows.

Describing the losing condition. We would like to specify that a text displaying "Game over" appears when the ball is out of screen. For that, we first create the label and make it invisible by default. To specify when this text should become visible again, we follow these steps. We set the velocity of the ball towards the bottom of the game and launch the simulation. When the ball goes out of the screen, we pause the game. The engine detected the event of the ball going out of screen.



We select the out-of-screen event, make the "Game over" text box visible, and confirm this behavior by pressing OK. The system then creates the corresponding rule automatically.

Describing the winning condition. Finally, we would like to specify that a label "Victory" should appear when the score reaches 19, which is the number of blocks in the game. For this purpose, we make a text displaying "Victory" in the middle, initially invisible. We then change the score to 18 to indicate an interesting starting scenario for simulation. We then play the game until the score reaches 19. When we press the event menu, the system detected the previous change as a *number change event*. We select it and accept the offered condition "When **score** == 19".

We then make the "Victory" text visible and confirm the rule.

This completes the description of the basic functionality of a break out style game.

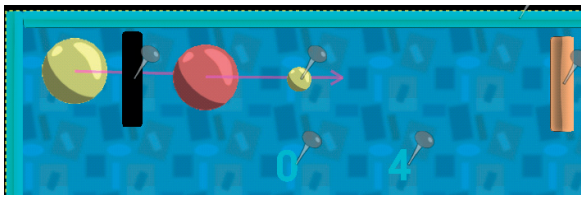
2.2.2 Pacman with Accelerometer Input

We next show how to build a Pacman-like game controlled using the accelerometer (the angle of the device). Only one part of the game field is visible on the screen, so the camera (viewpoint) needs to follow the player. The player moves in the labyrinth, and needs to eat all food chunks without touching any of the enemies. We will introduce three enemies; if they touch the player, the player loses a life. After losing four lives, the game is lost. The player wins by eating all food chunks.

We create the game logic from scratch, specifying that the player should be able to pass through black walls, eat the chunks, and indicating when the enemy should pass through the chunks and when they make the player lose a life. The steps are as follows.

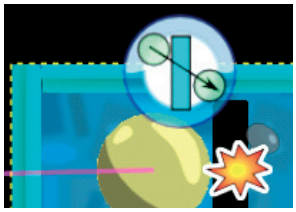
1. Create a yellow circle for a player, a red circle for an enemy, a yellow circle for a

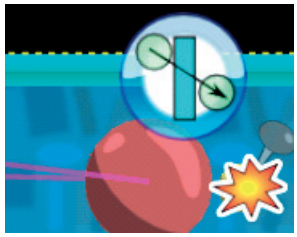
food chunk, a brown rectangle for a wall and a black rectangle for "home" (a wall through which the player can go, but not the enemies). Add two numbers, set one to 0 to count the number of chunks eaten, and the other one to 4 to count the number of lives. Set the initial speed of the player and the enemy.



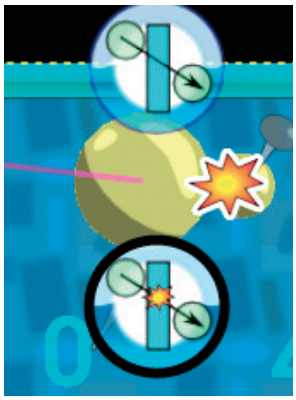
2. Start the game. After a few seconds, pause it.

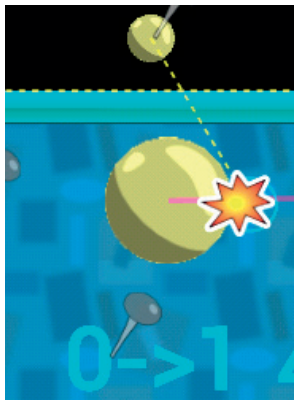
3. Suppose the player, undesirably, bounces against the black rectangle. Press the event menu, select the collision, and press on the button to remove the collision. Now the player passes through the black wall. Follow similar steps to remove the collision between the enemy and the food chunk.





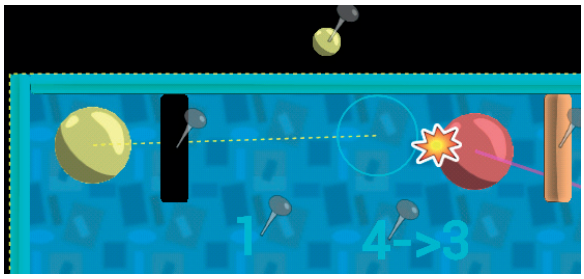
4. Next, observe that the player collides with the food. Select an instance of this collision, choose an action that the objects should go through each other but the collision is recorded, augment the score from 0 to 1, and move the food away from the visible part of the screen.





5. Move the time forward to observe the enemy bouncing against the wall and colliding with the player. Select the collision, select the option that the object should go

through each other but the collision is recorded (as for the food), move the player back to the base, set its speed to zero, and decrease the number of lives.

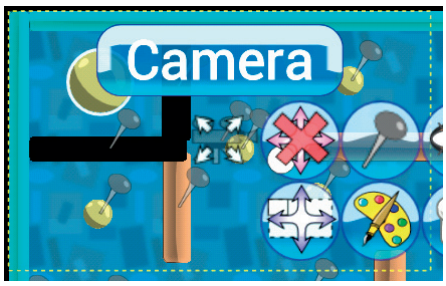


6. Create the remaining part of the game field, including a maze, by duplicating the walls. Replicate food and enemies, as needed.

In the second phase of constructing the game, we would like to have the player to be affected by a notion of gravity, and we would also like the camera to follow the player.

1.  Press the accelerometer button to be able to select the shapes affected by this sensor, and select the player. Press the accelerometer button to stop selecting objects.

2.  Press the camera button and then select the player. As a result, the camera will follow the player. To adjust the view screen, resize the camera.



3. To add “Victory” and “Game over” labels, follow the steps mentioned in the previous game.

In addition to its immediate use when following the player in a large game field, the camera can be used to switch between several screens in case of a multi-level game, or a game with menus and options.

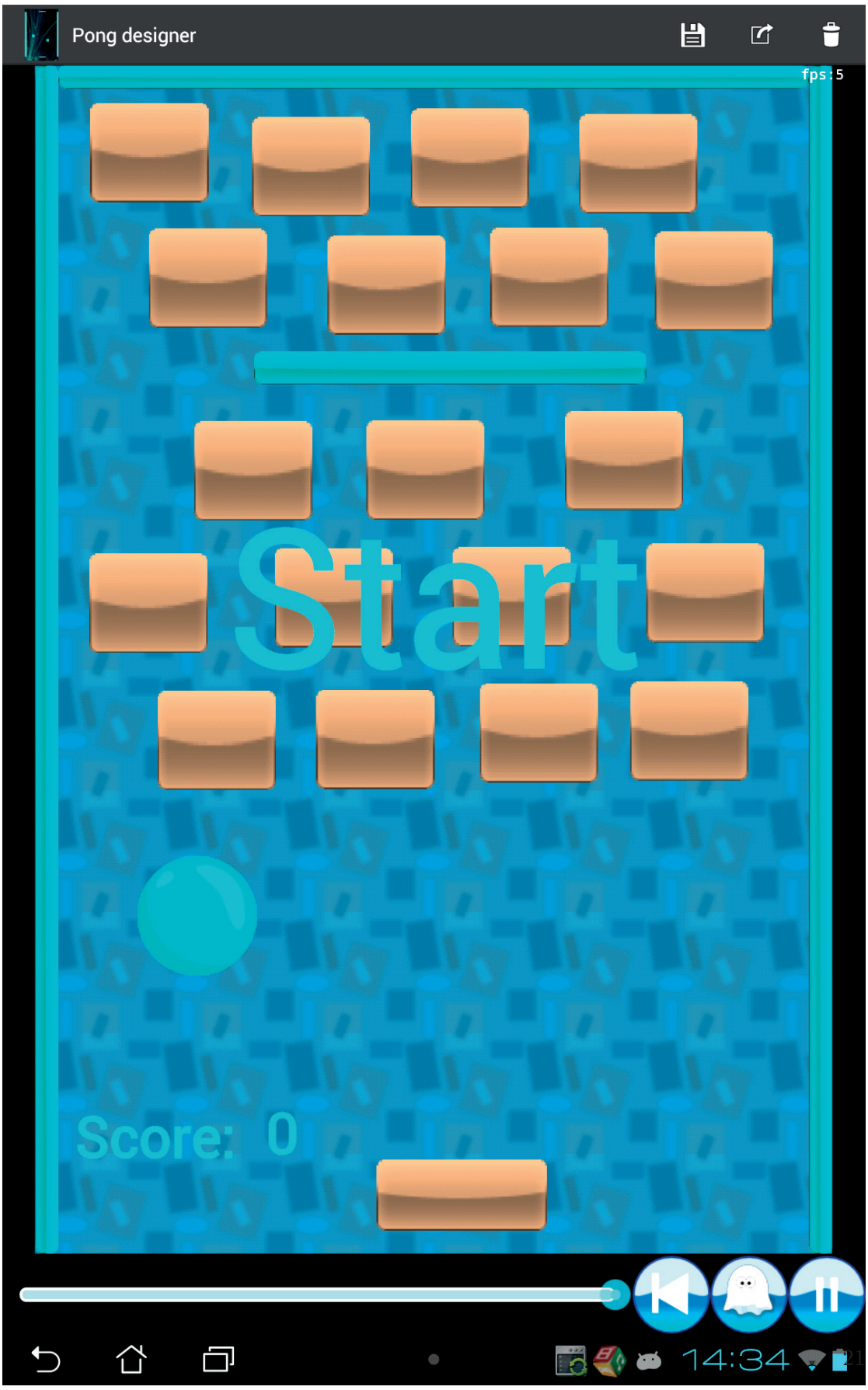


Figure 2.1 – Static Initial State of a Breakout-Style Game

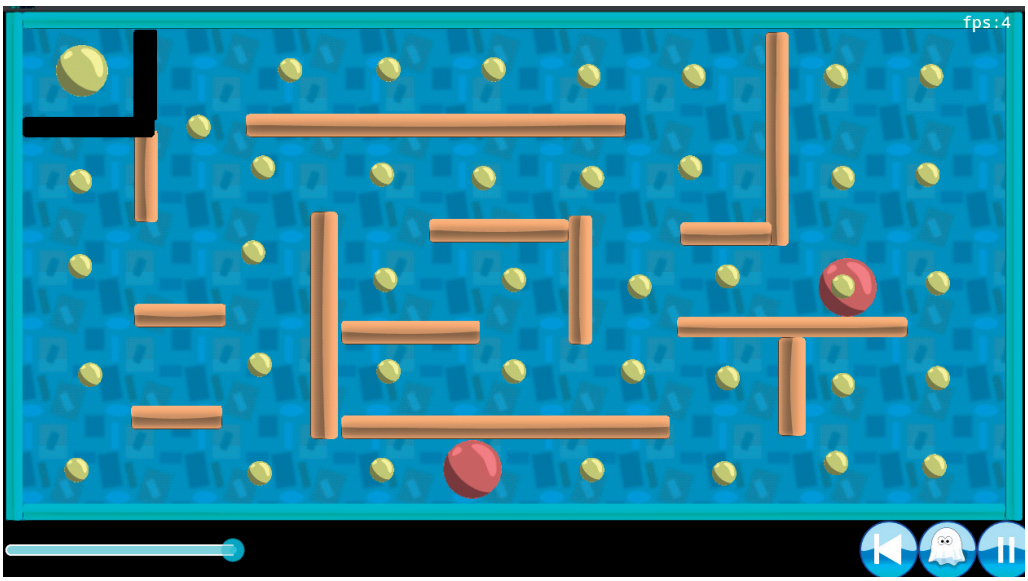


Figure 2.2 – Screenshot of a Pacman-like game build in Pong Designer

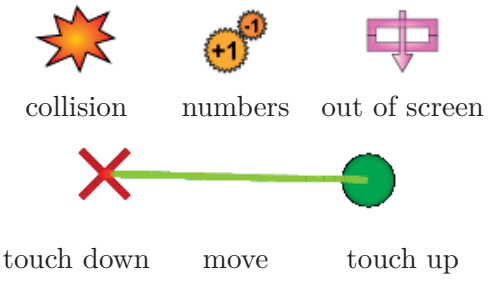
2.3 Pong Designer Principles

The key novelty of Pong Designer is the rule-based model with the ability to dynamically change rules through concrete demonstrations in a desired context.

Starting from one or more concrete demonstration (which can be introduced incrementally at different points in time), the system performs a generalization to obtain a rule that applies beyond the concrete state in which it was demonstrated. Rules contain a condition i.e. an *event*, and the *action* (state change).

2.3.1 Events

Pong Designer currently supports six kinds of events. Each kind of event may include several variations. For example, the user can specialize collision rules as "objects go through each other and no rule is executed", "objects go through each other and the rule is executed" or "objects bounce against each other and the rule is executed".



2.3.2 Specifying Actions

Selecting an event enables the user to specify an action using an example of an input and the corresponding output, which the system generalizes.

During the modification of the game state, the system draws two versions of objects being changed. The first version shows the original state of objects (input); the second state is the transformed state of the objects (output). It is possible to modify either the input or the output. However, the user mostly only changes the output state, as the input state is often the current state before adding a rule. To switch between these two, the developer toggles the input/output button.

When the user changes the output, moving a shape moves it as usual normally (the first picture below). This is the default mode. When the user changes the input state, moving

a shape will move a shadow version of it.

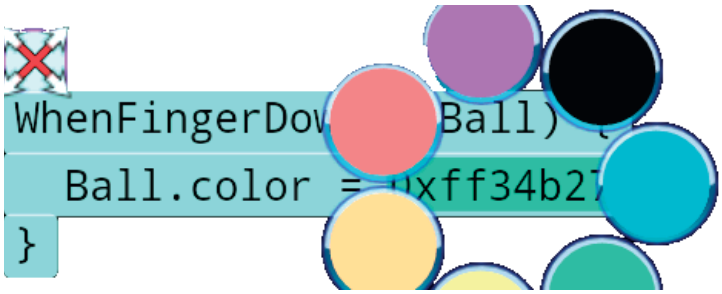
2.3.3 Editing Created Rules

Once created through demonstration, it is possible to directly adjust the rules. This functionality is provided as a fallback for the cases in which the demonstration does not achieve the desired effect or the user prefers to examine a more textual version of the rules. Note that, even in this representation we use graphical notation for events. Moreover, it is possible to edit constants in rules using selection or increment and decrement actions that requires no keyboard input and is therefore convenient for touch-based input. Finally, a preview feature makes it possible to quickly preview the effect of a single rule invocation on the current state.

Pressing the OK button creates a new rule based on the modifications of the game, or refines the existing open rule by providing a new demonstration.

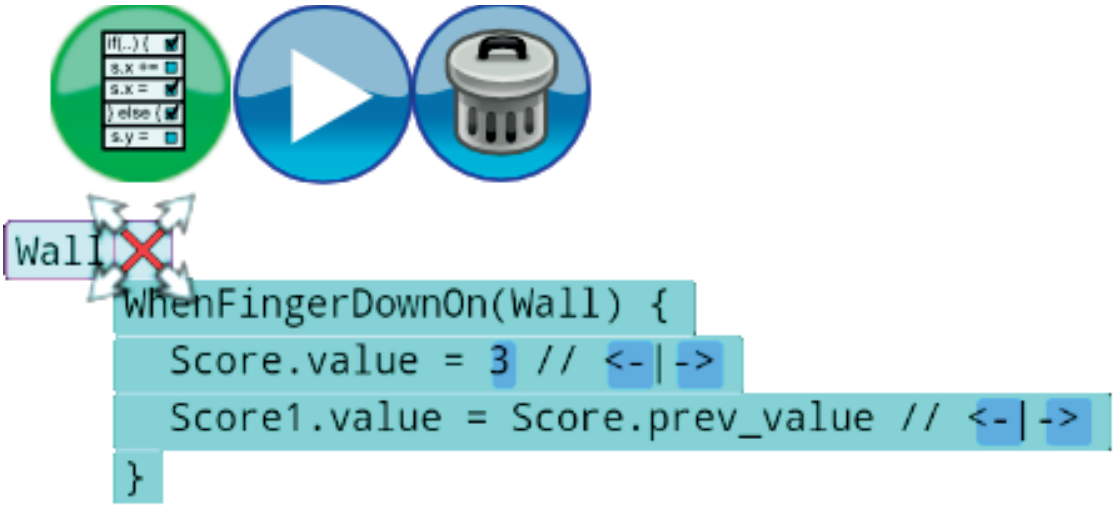
After a rule is created, one can display its content in a form that mixes text and graphics for further review and editing.

Drag movement on the touch screen over a constant in dark blue modifies the value of the constant. Depending on the type of the number, an appropriate input method is invoked. For example, changing a color constant opens a color palette.



Pressing the arrows enables choosing different code possibilities that were generated by

the system. For example, we can switch between three candidate code fragments that transform 1 into 3, which are $= \ldots \times 3$, $= \ldots + 2$ and $= 3$.




To apply the code of the rule to the game, press the play button. This can be used for example to modify parameters and to see how the rule behaves for them, in order to correct them. If the rule is opened, the game shows what the results of the rule would be.

2.3.4 Underlying Domain-Specific Language

Each game in Pong Designer can be described by its initial state and the set of rules. Although rules can be modified and displayed graphically, they also have a textual representation as a domain-specific language embedded into Scala. We use this domain-specific language as reference semantics, but also as a way to emit a compiled version of the game. Figure 2.3 summarizes this domain-specific language.

The game engine considers rules similar to logic gates. This means that the order in which the game executes their inner lines of code is not important. The game modifies all the parameters at the same time, through the use of a stacking mechanism illustrated in Figure 2.3. In all the assignments, the parameters names to the right of the equality start with "prev", which means that their value is the one before the rule started. In the sequel, whenever $v \mathrel{+}= C$ is written, it means in reality $v = prev\_v + C$ so that the previous value of v is left unchanged for other lines of code.

When the user provides an input/output example of how the game state should change, the code generator compares the example against the list of actions in Figure 2.4. If several actions are suitable for the same change, the code generator wraps all of them in a parallel instruction `PARALLEL_EXPR`, in the same order as in Figure 2.4. The meaning of such an

expression is "Execute the first action, but keep the others in the case the first action is wrong." For example, if the code has to change a score value v from 1 to 2 when a certain event occurs, the system will create the line `PARALLEL_EXPR`$(v = 2, v \mathrel{+}= 1, v \mathrel{*}= 2)$. Now, if the same event occurs and the user specifies that the score should increase up to 3, the system will drop the first $v = 2$ and the last $v \mathrel{*}= 2$ and will keep only the second $v \mathrel{+}= 1$.

Furthermore, if the user enters a new input/output example which is contradictory regarding an existing piece of code, the code generator selects the most recent one. For example, if the code for a rule is `PARALLEL_EXPR`$(v = 2, v \mathrel{+}= 1)$, and the player asserts that the value should increase from 2 to 4, this is not consistent. Therefore, the code generator will overwrite the previous rule by producing the following code `PARALLEL_EXPR`$(v \mathrel{*}= 2, v \mathrel{+}= 2, v = 4)$.

If the event occurs when there is a finger move on a shape, then the variables describing the move can be used in the code. The system accepts relative coordinates imprecisions up to 40%. For example, if the user moved his finger from $xFrom = 90$ to $xTo = 140$, and he also moved the shape x from 80 to 128, the system outputs the code `PARALLEL_EXPR`$(x \mathrel{+}= xTo - xFrom, x \mathrel{+}= 50, x = 128)$ because $128 - 80$ is approximately equal to $140 - 90$. Enriching such expressions by considering an arithmetic interval solver to accomodate imprecisions is a problem we might investigate in the future.

Although each atomic modification `MODIF_LINE` could be, in principle, arbitrarily generated by the grammar, the generator may only use patterns from Figure 2.4, especially for conditional if-then-else statements. Whenever other shapes are involved in those patterns, it means that the generator loops over all the shapes having the desired property, such as a "width". For each shape so that the pattern works, it outputs a code snippet. These patterns ensure coherent code and might be extended in the future as we add other behaviors.

2.3.5 Rule Creation Algorithm

Whenever the user performs a new demonstration, Pong Designer uses it to adjust the existing set of rules.

Creating and updating rules are similar activities. If the rule does not exist yet, the system creates it according to the type of the selected event, and its action is initially empty (see Figure 2.8)

The system extracts the code from the game state. It uses templates to generate the code (see Figure 2.6). Templates have access to the game state, so they can, for example, provide a code to align shapes, or set up a number as a combination of two other numbers. The template system in Figure 2.5 gives an idea of our template matching process.

When the system recovers the code from the game, it merges them with the existing code from the rule (see Figure 2.7). If there are number conditions for the new code, the system generates corresponding if-then-else statements or refines the existing ones. Generated if-then-else statements currently only check whether the number is less than constant, so the code is easily maintainable.

When the user duplicates a shape, if the condition of a rule contains the shape, the system duplicates the entire rule by replacing all occurrences of the old shape with the new shape. This can lead to an substantial increase in the size of the code, which we hope to reduce in the future by abstracting collections. If the condition of a rule does not contain the shape, the system duplicates every line of code modifying one of the shape's properties for the new shape property.

In the future we expect to deploy more sophisticated algorithms for learning from examples, and, more broadly, machine learning techniques to infer the intended behavior.

```
GAME := class NAME extends Game
        '{' GAME_CONTENT '}'

GAME_CONTENT := layout_width = constant
                layout_height = constant
                {SHAPE_DEF}+ {CATEGORY_DEF}+
                {RULE_DEF}+

SHAPE_DEF :=
  val NAME = new (Rectangle | Circle | IntegerBox | TextBox)
  '{' {property = value}+ '}'

CATEGORY_DEF := Accelerometer({NAME}+)
                | Gravity2D({NAME}+)

RULE_DEF :=
    WhenFingerDownOn(NAME) '{' CODE '}'
  | WhenFingerUpOn(NAME) '{' CODE '}'
  | WhenFingerMovesFrom(NAME) '{'
      (xFrom: Float, yFrom: Float,
       xTo: Float, yTo: Float) =>
      CODE
    '}'
  | WhenNumberChanges(NAME) '{'
      (newValue: Int) =>
      CODE
    '}'
  | WhenEver(BOOL_EXPR) '{' CODE '}'
  | WhenCollisionBetween(NAME, NAME) '{'
      CODE
    '}'
  | NoCollisionBetween(NAME, NAME)
  | NoCollisionEffectBetween(NAME, NAME)

CODE := SIMPLE_CODE
      | if(BOOL_EXPR, SIMPLE_CODE, CODE)
SIMPLE_CODE := {PARALLEL_EXPR}+
PARALLEL_EXPR := Parallel(MODIF_LINE+)
MODIF_LINE := NAME.property = FORMULA
FORMULA := FORMULA (+|−|∗|%|/) FORMULA
FORMULA := NAME.prev_property
FORMULA := constant
FORMULA := newValue|xFrom|yFrom|xTo|yTo
BOOL_EXPR := FORMULA (≤|≥|<|>|==) FORMULA
BOOL_EXPR := BOOL_EXPR (|| | &&) BOOL_EXPR
BOOL_EXPR := !BOOL_EXPR
```

Figure 2.3 – An overiew of the language and grammar used by our system to generate code

```
cx = x1,      cx = cx1,      cx = x1+w1
x = x1−w,    x = cx1−w,    x = x1+q1−w
x = x1, x = cx1,            x = x1+w1
x += xTo−xFrom,             x += xFrom−xTo
x += C_N,                   x = C_N
x += yTo−yFrom,             x += yFrom−yTo
(cx, cy) = (2∗cx1−cx2, 2∗cx2−cx1),
```

Templates for generic shapes position. Templates for y are simply obtained by replacing x by y. The last line describes the detection of mirrored shapes. Identifiers xFrom, yFrom, xTo, yTo representing finger movements are available only if the code is inside a WhenFingerMovesOn rule.

```
angle = C_N,                angle += C_N
angle = angle(x1, y1, xTo, yTo)
velocity *= C_F,            velocity = C_F
color = C_N
visible = C_B
```

Templates for generic shapes properties. The function angle describes the angle of the center of a shape to the finger.

```
width += xTo−xFrom,     width += C_N
width = C_N             width *= C_F
height += yTo−yFrom     height = C_N
height *= C_F           height += C_N
radius += C_N,          radius *= C_F
radius = C_N            radius += xTo−xFrom ...
```

Templates for rectangular and circular shapes.

```
v = nv
v = nv / i if nv % i == 0
v = nv ∗ i
v += C_N if C_N == 1 or −1
v = v1 + v2,                v = v1 − v2
v = v1 ∗ v2,                v = v1 / v2
v = v1 ∗ C_N,    v = v1
v += C_N if |C_N| > 1
v = C_N
text = text1          text = text1 + text2
text = Constant
```

Templates for integer and text boxes. We write "v" instead of "value". The identifier "nv" represents the new value if the rule is triggered by a changing number.

Figure 2.4 – The language of actions that the game engine generates by decreasing priority for each property. For each property, we write "property" instead of NAME.property. "other" and "other2" are identifiers to describe other shapes. We abreviate "other.property" to "property1" and "other2.property" to "property2". cx is "center_x", w is "width" and h is "height". C_N is an integer constant, C_F a float constant and C_B a boolean constant.

```
trait TemplateShape {
  var shape: Shape
  def variants(s: Shape): List[Expression] = {
    shape = s
    if(condition) List(result) else Nil }
  def condition: Boolean
  def result: Expression
}
```

Generic template definition as a trait. A template needs to define its condition and its result.

```
trait TemplateOtherShape extends TemplateShape {
  var other_shape: Shape
  override def variants(s: Shape) = {
    shape = s
    var expressions = Nil
    for(o ← game.shapes) {
      other_shape = o
      if(other_shape ≠ shape && condition)
        expressions += result
} } }
```

Generic template to compare against other shapes. Such template can be use to detect alignments, same color, number equality, etc.

```
trait TemplateParallel extends Template {
  def templates: Traversable[Template[T]]
  def result: Expression = {
    var expressions = Nil
    for(template ← templates)
      expressions += template.variants(shape)
    Parallel(expressions)
} }
trait TemplateBlock ...
```

Special templates regrouping other templates in parallel or in block.

```
object TX_DX2 extends Template[Shape] {
  def condition = ofType(TOUCHMOVE_EVENT) &&
      approx(shape.x−shape.prev_x, xTo−xFrom) &&
      !movementIsVertical
  def result = "shape.x = shape.prev_x+(xTo−xFrom)") }
object TX_AlignLeft1 extends TemplateOtherShape {
  def condition = approx(shape.x, other_shape.prev_x, 20)
  def result =   "shape.x = other_shape.prev_x" }
object TX extends TemplateParallel {
  def condition = shape.prev_x ≠ shape.x
  val templates = List(TX_DX2, TX_AlignLeft1) }
object TShape extends TemplateBlock {
  def condition = true
  val templates = List(TX, TY, TColor ...) }
```

Basic templates to match horizontal finger tracking, alignments on x, and mutliple variants for changes observed on x. The last template TShape regroups other templates to create the body of a rule.

Figure 2.5 – Templates are named structures producing code for a given shape. Figure 2.4 describes the possible results.

```
def codeGeneration(game, event,
                   actionsCondition,  existingActions)  = {
  actions ← ()
  initialize templates
  for{shape ∈ game} {
    variants ← TemplateShape.variants(shape)
    if{variants ≠ ()} {
      actions += ParallelExpr{variants}
    }
  }
  mergeCode{game, event, actions,
            actionsCondition,  existingActions}
}
```

Figure 2.6 – codeGeneration: Algorithm which takes a "game", an "event", an optional condition "actionsCondition" under which actions in the game should be performed, a list "existingActions" of actions that are currently performed when this event is triggered. Outputs a sequence of actions describing the intended action merged with the previous ones.

```
def mergeCode(game, event, actions,
                      actionsCondition, existingActions) = {
(actionsCondition, existingActions) match {
  case (true, ()) ⇒
    actions
  case (true, "if("cond")" code_T "else" code_F) ⇒
    "if("cond")" mergeCode(..., actions, true, code_T)
    "else" mergeCode(..., actions, true, code_F)
  case (true, _) =>
    - Group by assigned property existingActions and actions.
    - Intersect expressions parallelExpr for the same property
    - If intersection is empty, take the new code.
    - Return the resulting block code.
  case ("newValue ≤" B,
        "if(newValue ≤" A")" code_T "else" code_F) ⇒
    if( B < A ) {
      "if(newValue ≤" B")" mergeCode(..., actions, true, code_T)
       "else (if(newValue ≤" A")" code_T "else" code_F")"
    }
    ...
}
```

Figure 2.7 – mergeCode: Algorithm which takes a "game", an "event", an optional condition "actionsCqondition" under which actions in the game should be performed, a list "existingActions" of actions that are currently performed when this event is triggered. Outputs a sequence of actions describing the intended action merged with the previous ones. Notice how conditionals are merged: The structure of the program remains consistent. Texts in quotes are the representation of a program.

```
def ruleMerge(game, event,
                        existingRule,   actionsCondition)  = {
if{existingRule   is not defined} {
  existingRule  ← emptyRuleFrom(event)
  //existingRule.code is empty
}
actionsCondition  ← ''true"
if(event  is  a ''Number change event'')  {
  if(event  is  a ''Number equal event'')  {
    actionsCondition  ← ''newValue == event.shape.value''
  }
  else  if(event  is  a ''Number greater  event''){
    actionsCondition  ← ''newValue ≥ event.value''
  }
  else  if(event  is  a ''Number less  event''){
    actionsCondition  ← ''newValue ≤ event.value''
  }
  else  if(event  is  a ''Number positive  event''){
    actionsCondition  ← ''newValue ≥ 0 ''
  }
  else  if(event  is  a ''Number negative event''){
    actionsCondition  ← ''newValue ≤ 0 ''
  }
}
rule.code  = codeGeneration(game, event,
                              actionsCondition,   existingRule.code)
game.rules += rule
```

Figure 2.8 – ruleMerge: Algorithm which takes a game with input and output state available, an event and an optional existing rule. Outputs a rule to describe the complete event handling.

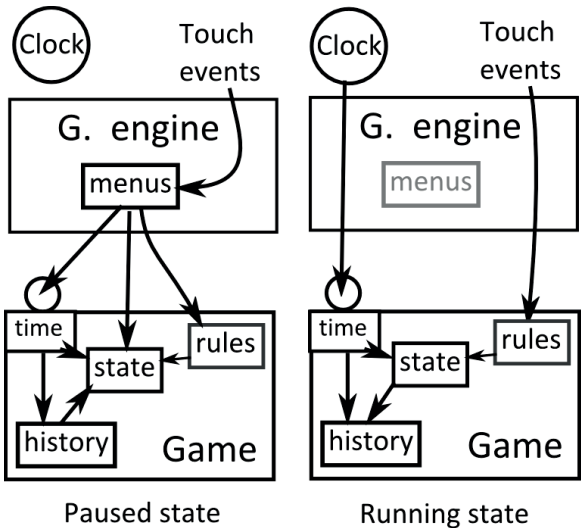


Figure 2.9 – The two states of the game engine

2.4 Implementation Aspects

We next describe the architecture of Pong Designer that enables the modification of run-time behaviors.

2.4.1 Role of the game engine

Time plays an important role in the game engine. The game itself is not aware of the real time, only the game engine is. To manage time, Pong Designer can be in two different states. In the running state, the game runs naturally, whereas in the editing state, the game is paused and everything, including current time, is editable. Because of these two states, we made sure that the time of the game is tightly controlled by the game engine.

To control the time for the current game, the game engine performs the following actions synchronously and forever:

1. When in the running state, it updates the time of the game from a clock.
2. When in the editing state and if modified, it updates the time of the game from the time slider.
3. Displays the game.
4. When in the editing state, displays more information about the game state, such as if objects are static or if an object has been modified.
5. When in the editing state and activated, displays selectable events from the last 5 seconds in order to let the user create or modify rules.
6. When in the editing state, displays the game menu on top of it.

Concerning touch events, the game engine behaves differently. Because of the platform, touch events are received asynchronously. The game engine deals with them in two different ways:

1. When in the editing state, the game engine dispatches touch events to menus which in return modify the game state
2. When in the running state, the game engine dispatches touch events directly to the game

Figure 2.9 gives a summary of the interface between the game engine and the game in the two states The slider controlling time is included in the box “menus”.

2.4.2 Time management by games

Games running with this game engine need to have the possibility to go back in time for at least a short period. Therefore, a game needs to store internally a 5-second history of all of its parameters and its shapes' parameters. Events are a particular case (see below) but are still recorded in a 5-second history. If the user wishes to start from the beginning, or if the game has been wrong and not corrected for the last five seconds, it is still possible to reset it to the initial state.

The data structure storing the history of parameters is a double linked list of values ordered by timestamps. This structure is parametrized in the type of the values, which can be anything among integers, floats, string and events. Except for events, we optimized the history by storing values only when they change, so that if the parameters of an object does not change, its history holds only one value.

When time elapses (forwards) to a new value set by the game engine, the following actions occur:

1. The game goes through all touch events that were stored asynchronously, and executes them according to its rules.
2. The game updates the physics by moving shapes and handling collisions and at the same time triggers synchronous events, like those from collisions, out-of-screen events and number change events.

The time can elapse backwards only when the game engine is in the editing state. In this case, if the user goes backwards in time with the slider (up to 5 seconds) the game reverts all its parameters and all its shapes' parameters to their value at the given time. If the user moves the time slider back to the right, the game executes forwards as if it was running.

2.4.3 Two kinds of events

How events are generated and stored is the key point to understanding how games are executed in Pong Designer. We distinguish two kinds of events.

Firstly, asynchronous events, such as touch gestures and accelerometer changes, are stored in a buffer. When the time increases to another value, the game flushes all these events. Because they are external to the game, they provide the necessary input to play it. After having triggered corresponding rules, these events are recorded in a 5-second history. When the game engine is in the editing state, they can therefore be selected by the user to create new rules or modify existing ones. When the user then lets the time elapse forwards in the editing state, the game replays these events from the edited

history. It is thus possible to change the rules and to see their different outcome for the same touch input immediately, making it convenient to determine constants, for instance if rules are changing speed, position, etc.

The second types of events, synchronous events, such as collisions, out-of-screen events or number change detection, are detected after the physics is updated. When they are detected, these events might also trigger rules that are part of the game. They are also stored in a 5-second history. Although the user can still select them to create new rules and to modify existing ones, they will always be recomputed when the time elapses forwards, both in the editing state or in the running state.

2.4.4 Deployment on Android

We are compiling against the latest Android API version 17 (Jelly Bean) by using the SDK that Android provides. Our application is also compatible until the API 10 (Gingerbread). Because the SDK is written in Java, and because the Android virtual machine only deals with Java-like classes, we are using two different plug-ins to be able to program in scala: sbt and AndroidProguardScala. Because Scala libraries are not available on Android by default, the two plug-ins embed Scala libraries to provide a final stand-alone application. Our prototype application is available from the Android Play store as Pong Designer.

2.5 Discussion

Our purpose is twofold. The first objective is to reduce the gap between coding and testing, and the second is to allow the user to learn faster how to program by providing him a comfortable environment.

First, let us remark that there is an inherent duality between the code and the interface. This dual paradigm is representative of a major duality in the software development: compilation vs. testing, programmer vs. designer, engineering vs. marketing, developer vs. user, etc.

Because of too simple design decisions, for many systems the interface maps the code implementation, and do not meet the goals of the users [Cooper et al., 2012]. For example, a program would like to ask if the user wants to save the changes, which in most cases should be done without asking. This happens because it reflects more or less the way the file system internally works.

The approach of self-reconfiguring interfaces is to try to reduce as much as possible the gap between the configuration and the execution. Reactive customization is at the core of self-reconfiguring interfaces. The purpose is to empower the user with programming

capacities, by specifying a desired behaviour on-the-fly. Coding should be done by the interface itself, so that the programmer would not spend too much time learning an API.

Bret Victor investigated the way courses currently teach programming [Victor, 2012b] and depicted its bottlenecks. By comparing the program output to the code, B. Victor found principles for programming environments, if implemented correctly, would lead to a better understanding and a better learning curve for users and programmers.

> "Traditional visual environments visualize the code. They visualize static structure. But that's not what we need to understand. We need to understand what the code is doing."

To understand what the code is doing, we use visualization, debugging and verification systems. However, there are few programming environments that allow to directly manipulate what the code is acting on. Usually, any interaction of this kind only provides backwards pointers, such as retrieving the original position in the source of a compiled TeX file. We aim at providing more code What You See Is What You Get (WYSIWYG) customization features based on the manipulation of the outcome of the code, in order to demonstrate intended behaviors.

B. Victor identified the following list that users need for an enjoyable coding experience. We add a comment after each principle to suggest how we may be contributing to these guidelines:

- **Show the data** - we show the physical world;
- **Show comparisons** - we use the time slider to compare the initial and final states of actions;
- **Get something on the screen as soon as possible** - users can insert basic shapes easily;
- **Create by reacting** - to create the rules, users select events that occurred in the past;
- **Create by abstracting** - the system automatically abstracts demonstrations into general rules.

Looking into future, the main challenge will likely be to find the right tradeoff between the complexity of the code we want to generate by demonstration and the visual simplicity. For example, it might be difficult to design multiple winning conditions in our current game engine if they are described by a complex boolean formula. Generalizing conditions in certain ways is currently impossible, for example, checking if all numbers within a

Figure 2.10 – Code required in Scratch to describe consequence of a collision.

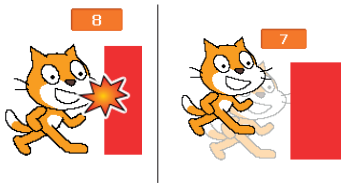


Figure 2.11 – Hypothetical illustration of applying Pong Designer approach to the Scratch example. The developer pinpoints to a visually represented event, then changes the state into the desired one. The system infers the state transformation from the example demonstration by finding a function that maps the input to the output state.

certain range are greater than 10, and adding a new number to the range. We believe, however, that we are close to a system that can be used productively for a range of 2D games. We hope that, in the future, we can carry such directly manipulation to other domains, using the idea of physical manipulation as a metaphor with which most individuals have deep and early experience.

2.6 Related Work

Most existing approach for game programming do not support inference of rules from demonstrations. We make an overview of some of those systems, as well as some of the approaches for inferring code from examples.

Scratch. Scratch is a game engine that helps to teach programming to children aged 8+. It provides all the typical structures of programming, loops, threads, if-constructs, tests, intersection detection, stylus, etc. There are two main screen areas in Scratch: one with programming blocks, and the other with the canvas. The objects on the canvas can be directly moved and rotated with the mouse, and programming has been made easier through assembling of compatible predefined blocks, which prevents the construction of programs that would not correct according to types and the syntax. The system is simple and seem appealing for teaching conventional programming through a graphical variation of the usual textual rendering of program text. While the code is running, it is possible to grasp objects, move and rotate them. However, we found that writing coordinates by hand, which is the only way to introduce specific coordinates into a program, can be cumbersome. Consider a situation where we wish to move the cat when a given line of code is executed to a position for which we do not know the coordinates. Currently,

one needs to first move the cat round, note down the coordinates, and then enter them into the source code, which is much less immediate than in our system. In general, the programmer needs to define all constants by hand. Another important feature that we found lacking is the ability to go back in time to identify the desired behaviors. A major part of programming is to define the behavior of the interaction between two objects, but such behaviors cannot be defined on-the-fly in Scratch. If an object, let us say a cat, should loose a life when it touches a red enemy, the programmer normally creates the code such as in Figure 2.11. The creation of the analogous code is easier if we graphically set up the effects of the collision. To do so, in our system we just select the event on the screen, then change the position and the number of lives. The desired code is generated automatically. we believe that educational systems such as Scratch would also benefit from our demonstration-based approach

When it comes to program a single stand-alone behavior, e.g. for an enemy to try to reach a player, Scratch allows the user to program any looping constructions, branching conditions, lists and variables to achieve the desired result. Because of its programming paradigm, Pong Designer still lacks such explicit programming features. However, we could imagine in the future to program an AI by specifying a score which needs to be incremented, and the rules that increment the score. With some learning algorithms and input restrictions, the AI could learn to move a paddle towards a ball, or to go away from the player if the latter is in an invincible mode in order not to lose lives. For more sophisticated AI strategies and descriptions. we still need to enrich the interface and our language.

Programming by example. [McDaniel, 2001] reports that programming-by-demonstration paradigms are often Turing-complete, so is our engine. However, this completeness usually does not lower the complexity of programming non trivial tasks. Such paradigms become useful in the presence of a library, which directs the purpose of the programming. This is a reason why we choose to provide a direct support for physics, so that providing examples allows the programmer to quickly create games without having to worry about details. For example, he does not need to create the bouncing code when a collision occurs.

[Gulwani, 2012a] and [Singh and Gulwani, 2012b] reports that programming constraints can be learned and generalized by their system through a given set of input/output examples. By using inductive synthesis with a DSL, they were able to find all the expressions that could match the inputs to the outputs. Their examples included text editing macros and spreadsheets. Our engine follows a similar algorithm on graphical input, although it is much less complex for now.

The way rules are refined according to multiple input-output examples is similar to the Version Space Algebra method which automatically learns programs from traces [Lau

et al., 2003a], as well as to Angelic nondeterminism [Bodík et al., 2010], which also provides a methodology to fill the missing parts of code based on trace executions and specifications.

Quickdraw [Cheema et al., 2012] is a graphical system which rebuilds precise graphics based on vague input. It also inspired us to enforce the robustness of our system against minor graphics specification errors.

Finding a way to manage coexistence between the code and its execution has already been a source of many more or less fruitful experiments. The Khan Academy [Sal, 2012] focuses more on "play with the code" than on its graphical output, which is vigorously criticized by [Victor, 2012b]. Our approach incorporates many important points of this criticism to make graphical programming enjoyable.

Simula and Smalltalk. A pioneering object-oriented programming language Simula is an excellent programming model for physics-based games as well as other domains that can be viewed in analogy with the physical world. Smalltalk builds on this tradition and further emphasizes graphical environment and the ability to manipulate the state directly. Sánchez-Ruíz et al. [Sánchez-Ruíz and Jamba, 2008] showed that 4th and 5th graders liked to program using the object-oriented programming interface Squeak and its graphics, but disliked correcting errors. Squeak provides a graphical interface, as well as contextual menus for on-the-fly editing purposes. Our tool also aims to provide a graphical interface that even children enjoy programming. The Morphic environment allows the user to program graphical interactions between objects called Morphs and has an object-oriented language inspired from Smalltalk. The structure of Morphs is organized around a hierarchy of traits and prototypes, which allows the user to factor behaviors and attributes. Similarly to our system, it let the user bind input events to actions. It provides a graphical editor as well as an interactive way of writing code for objects, especially prototypes.

Programming for phones. We also draw inspiration and insight from the TouchStudio/TouchDevelop project [Tillmann et al., 2011]. The TouchStudio/TouchDevelop project is related in the spirit to our work, because it also uses the hand to graphically program scripts on tablets and has a language that simplifies the general programming model.

Game engines. According to [Bishop et al., 1998] there is a need to separate the game content from the game engine. For efficiency reasons, he asserts that there is a need to specialize the game engine according to the kind of game that can be produced. One of the main design goals of the game engine should be the speed of execution. With a proper

scene manager, a dynamic collision engine and detection of visible objects, they were able to obtain a reasonable speed. The design of our game engine is similarly specific for the kind of games we would like to run. We are also taking inspiration from this paper to make our game engine faster, even if our collision engine is for the moment statical.

Construct 2 is a commercial HTML5 game prototyping engine with the associated community of developers and a portal for trading game components[Scirra, 2013]. It provides a game layout, a camera usually smaller than the layout, and lets the user add his or her own sprites. The game logic is on a separate sheet that is executed 60 times per second, resulting in professional quality of animation. We observed, however, that for some on-line games the authors mentioned that they had a hard time to debug their game logic. In our own experience, we observed that, for example, writing the specification to "spawn" objects (such as a bullet from a gun) requires the programmer to go back from the game to the game engine, to the image editor, and then to the event sheet. This process supports precise modifications, but misses the opportunity of intuitive contextual modifications. Therefore, we found this system would benefit from the techniques that we incorporated into Pong Designer.

Game Maker is a nice pioneering commercial game engine [Duncan and Kay, 2013]. It features room, sprite and object management, as well as customizable rules by behaviors or event/action. Although powerful in terms of the quality of games it can produce and export, the gap between the edition of the game and its compiled playable version is quite large. Even for tutorial games, the risk of misspelling variable in code is high. If for instance the user renames an object, the system does not rename it in the code, so compile errors appear. Furthermore, when playing, if error messages occur, there is no feature to modify the code where the error is to continue the game. We would therefore say that it does not meet the principles that we aim to fulfill and that we illustrated through our system.

Functional programming. Fruit [Elliott, 2001] is a functional programming language that defines GUI logic as signals and signal transformers. Signals approximately correspond to continuous variables, and signal transformers are code that perform actions on signals, such as integrals or conditional assignments. With this approach, a Pong-like game is programmed with only 20 lines of code. Part of the efficiency of this approach can be found in our game engine, where parameters and events play the role of signals, and signal transformers resemble rules.

Sound processing. ChucK [Wang and Cook, 2004] is a strongly typed language designed to write functional audio synthesis programs. Its programming paradigm is to provide full control over time features, and to use an arrow operator which captures the sequential operations of programming. One of the specificity of ChucK is on-the-fly

programming, which allows people to modify their program without having to interrupt the execution of the program, for example during a live performance.

Debugging environments. WhyLine [Ko and Myers, 2008] is a modern interactive debugging tool where the user can ask questions during debugging about why a certain change happened. By recording the execution trace, it is possible to solve complex debugging problems by navigating through history. Our system similarly uses time-backtracking to enable the precise design, refinement and modification of rules.

According to Lieberman [Lieberman, 1993], there is a huge gap between the environment of the code and the environment of the software. He suggests that visual users should be teachers for the interface of the software itself, which in return would act like as a learning student. His graphical programming environment includes the possibility to program macros by demonstration and to generalize them when translating them into code. He suggests that the generalization process is a key part of the learning of the software, and that small errors should be detected and corrected when generalizing. Similarly, our tool aims to be a learning student, which for example tries to correct small alignment mistakes made by the user who plays the role of the programmer. It produces macros that generalize the intended behavior provided by examples.

Tools and runtimes for existing languages. There are several tools that find, rank and present the most appropriated synthesized code portions to the programmer. InSynth [Gvero et al., 2013] is an IDE extension which allows users to synthesize code snippets based on the type of the current expression. Although these approaches are not completely automatic due to the lack of complete specifications, they reduce the burden of the programmer. Similarly, our tool finds, ranks and presents different code portions, so that the user can choose among them based on their original intent. On another side of the spectrum, Chameleon [Shacham et al., 2009] assists the programmer in the difficult task of choosing the best data type for the collections in a program.

Programming language extensions with constraints. Kaplan [Köksal et al., 2012] and Comfusy [Kuncak et al., 2010a] support the use of constraints as programming structures. Such structures allow programmers to work productively on explicit specifications rather than explicit code. The automatically generated code is thus less error-prone. Decreasing the number of potential errors is also the goal of domain-specific languages like those designed by Intentional Programming [Simonyi et al., 2006], which allows the programmer to work on a language that is closer to his needs. Our game engine also has a domain-specific rule-based language that is generated by the graphical selections made by the user.

2.7 Conclusions

Pong Designer enables users to modify games while they run, to step back in time, and to provide demonstrations of desired behaviors. The system infers corresponding rules and constraints, which can be manually modified afterwards. Based on object-based programming principles, users can create their game by moving and arranging elements while stepping through time. The system can generate code in a domain-specific language embedded in Scala, which runs on the Android platform using the standard toolkit.

We believe Pong Designer can be used to make games that are as fun to modify as they are fun to play. While there already exist games whose game worlds can be edited, the changes to behavior are currently limited, and there is a large gap between the sophisticated built-in behavior on the one side and simple customizations on the other side. We believe that Pong Designer leads reduces this gap, and we hope that this encourages experimentation and building of fun logic-based games and interactive games. We believe that the system can also be used for experiments exploring the learning and teaching of programming.

We are at this point confident that the approach can be successful in particular domains. The open question is the extent to which this success generalizes to broader domains, and the extent in which this paradigm can incorporate principles for managing complexity of larger applications.

Acknowledgements

We thank our shepherd and anonymous reviewers for useful feedback. We thank Sean McDirmid for useful discussions and comments. We thank Philippe Suter for his feedback about this project, as well as Eva Darulova for her feedback on the paper. We thank Lomig Megard, who has been contributing to an upcoming new version of the system that we described in this paper.

Page 12 until this page 42 contained the content of the paper "Game Programming by Demonstration". We will now add our own comments

B. Epilogue to Game Programming by Demonstration

B.1 New Findings

I had four friends without programming background, and I asked them to try out Pong Designer. The first two were 16-year old boys, the second a 23-year old girl and the third a 25-year old girl. I interviewed the first boy in July 2013, and the second roughly at the same time as I presented the previous publication, in October 2013. At the first usage of the application, a non-interactive tutorial (only pressing the "next" button) demonstrated Pong Designer's features: the time bar, the creation of an object, playing, creating a rule, selecting a collision, changing some effects, seeing the rule being recorded and replaying.

After the tutorial, I asked the boys to perform certain actions, such as:

- In a Pong Game, add a text displaying "Game Over" when the score reaches 5, and stops the ball.
- In a maze where we move the ball by tilting the tablet, find the unanchored wall and pin it so that it does not move anymore.
- Add a "teleporter" block, which makes the ball move closer to finish.

After the first two interviews, it became clear that I had to correct several mistakes:

- There were unexpected crashes, especially when undoing/redoing actions.
- There were unwanted side-effects when using the time slider.
- The homemade physics game engine was not always physically accurate.
- The meaning of the icons, especially those for finger gestures, was unclear.
- The selection of events and objects was ambiguous.
- It was unclear how to create rules for objects.

Although my friends seemed enthusiastic about the game engine, one of them described his experience as being forced to "think like the machine", probably referring to the fact that every rule needs multiple steps. For example, I asked him to create an object that, if collided with the ball, would teleport it to a given place. After a few minutes of seeing him confounded, I showed him the solution. However, he was not able to reproduce the solution after seeing it. Creating an object and a rule associated with it was new to him. Furthermore, he assumed that teleportation would be somehow built-in, not that he had

to simplify demonstrate the ball physically moving to another place. Finally, he assumed that he did not need to select the event, just go back to the time of the collision and move the ball. The two boys had similar reactions.

This discussion gave me some insight into the limitations of this system. Pong Designer felt to him like a new programming language, not the programming-by-example experience that I had in mind.

Four months later, I interviewed the two girls separately, asking approximately the same questions. This time I added some hand icons that appeared where the finger touched the screen, a preference of selecting objects incriminated by an event, and many bug fixes.

Still, some issues appeared when they used the interface:

- One girl first tried to demonstrate the effect (e.g., changing the ball's color) before selecting an event.
- She also forgot to "confirm" the rule by pressing OK, for example by launching the game. She complained that the confirmation should be closer to the modification.
- She did not understand the icons if there was no text associated with them.
- For both girls, the tutorial was not clear; one would have preferred an interactive one.
- Both girls told me they would feel limited if something more complex was created, for example with layers.

After pondering the issues, I also found more limitations:

- A limiting expressiveness of the underlying programming language
- No way to write custom game rules
- A messy way of mixing code and game blocks in the same area.
- An exponential blow-up in the number of rules if we duplicated objects interacting between them.

B.2 Discussion

We now answer the questions we asked in the introduction (see page 9) with respect to the work of this chapter.

Question	Game
Is it faster/easier to conduct tasks using programming-by-example?	Yes/No
Do users need to see the generated program?	Yes
Do users need to take on the generated program?	Yes
Is the paraphrased version of the program useful?	Conjectured
Is it more reliable when the program is shown?	N/A
Is it more reliable when the computer asks questions?	N/A

The comparative summary for all the papers is available on page 163.

We explain the answers to these questions. In this environment, developing certain kinds of games is faster than in any other environment that we know of. For example, in less than one minute, we successfully programmed a basic two-player Pong Game with scores. However, more advanced tasks, such as copying rules or adding custom code, are harder. It appeared quickly that users needed to see the generated program, in order to tweak it. The paraphrased version of the program, available in the second version of Pong Designer, looked promising but we did not test it. We did not have any evidence to conjecture that showing the program would increase the reliability. Similarly, we did not have any active questioning mechanism; the closest we had was a tool for "repairing" on-the-fly some game rules that the system had wrongly inferred.

Pong Designer v2.0: New interaction models

I continued to work in Pong Designer this time with a master's student, Lomig Mégard. We addressed the problems encountered in the previous approach.

Lomig Mégard implemented the new system from scratch, and I reintegrated the previous menus. After a few months, we had:

- A faster native 2D engine: Box2D (box2d.org).
- A categorization of objects: Every objects belong to a category.
- An on-the-fly rule repair engine: By selecting an event, all recently executed rules and their alternatives appear, in case the user wants to correct the inferred code.
- Text-based programming: Instead of mixing the code with the game on editing, which is quickly messy, we split the window in half: the code on the right side, the game on the left side. However, we did not enable users to edit the program, except for constants and alternatives.
- Modification of physical properties: We included the modification for any physical object of its friction, elasticity and many others, all editable in the text view by

clicking on it and selecting default or custom values.

- Rule broadening: Using the repair-engine on an event, it becomes possible to generalize the rule to all objects of the same category.
- Paraphrased programs: Instead of displaying some obscure code, whenever possible we show a paraphrased representation of it.
- Multi-language: We offered the translation of all the content to English and French.
- Juicy features: Color, textures and photo for block textures, text-to-speech, string learning (e.g., concatenation of strings).
- Timed vectorial paint experiment: We added a special objects that can record drawing in time, and another that can record sound. We were thus able to demonstrate a lightweight doodle recorder.

However, after all of this, I did not push this work forward. The next steps looked too risky with little possible outcome. First, I envisioned opening the door to any coding by parsing what the user would write; I also realized that writing code with the fingers on a tablet might not be ideal. Second, I envisioned creating a social community similar to Scratch [Resnick et al., 2009], so that users could share their code, and reproduce others' works. Nonetheless, the first step of basic programming would have been a prerequisite, for it is extremely hard in Pong Designer to modify a piece of code inferred by the computer.

I began to realize that the program should play a much important role in programming-by-example, at least after reaching the limitations of pure examples. This was my motivation for the second work, integrating a programming-by-example engine into a programming language.

3 Managing Files by Using Examples

"Si vous voulez vraiment comprendre une chose, essayez de la modifier."

"If you really want to understand something, try to change it."
Kurt Lewin

A. Prologue

After having developed a game engine seemingly too simple for experts and too complex for beginners (Chapter 2), I wanted more examples of how to apply programming-by-example in everyday life. I wanted to have the examples much closer to the user's abstract wishes than it was in Pong Designer.

Ruzica Piskac, one of the co-authors of this work, had the original idea of applying a successful programming-by-example-engine for string transformation to the task of manipulating files. Imagine that you are ordering your files, moving and renaming them; then, at some point the system understands what you are doing and suggests a transformation that you can accept or not. Following this vision, we realized this work.

StriSynth: Composite Scripting using Examples

The following pages 48-73 are the extension of the peer-reviewed paper named " StriSynth: Synthesis for Live Programming" (DOI: 10.1109/ICSE.2015.227) that appeared in the International Conference of Software Engineering (ICSE) in 2015 and was published by IEEE. The name of the extended paper is "Composite Scripting Using Examples". Probably because, in the paper, the results were not strong enough, there were no conferences that would publish this extended version of the paper. We include the work here with the permission of the authors, who are myself, Sumit Gulwani, Ruzica Piskac, Filip Niksic, and Marc Santolucito.

Contributions: *I did the entire technical part of the work. For efficiency and reproducibility, I reprogrammed almost all of Flash Fill's algorithms into a JVM-based implementation StringSolver*[1]*, written in Scala, based only on published papers. I designed the language for providing examples, wrote the program exporter to PowerShell and Scala, implemented the learning algorithms for partition and filtering following Ruzica Piskac's advice, and encoded and run the benchmarks in order to time them. Furthermore, I also designed algorithms for the ellipsis construct (to avoid specifying the entire output), for multi-digit counters and for number transformations (to enable the numbering of files). Mark Santolucito and Ruzica Piskac took care of the user study.*

In this paper, we describe a live programming framework that enables end-users to perform transformations over strings by using examples and to generate reusable stand-alone PowerShell scripts. Motivated by applications for automating repetitive file manipulations, we present synthesis algorithms and a language that are able to handle related tasks. This language contains operators that can be easily combined to create more complex transformers. We developed an open-source tool named StriSynth. StriSynth produces a script from user-provided examples. The user receives a feedback after every example. This way the user can choose examples to direct synthesis. StriSynth works in a live programming framework: the user can interactively start to learn new tasks, cancel given examples, test generated functions, and export the learned scripts. We evaluate ease-of-use and preferences for StriSynth over traditional scripting languages on both expert and end users.

3.1 Introduction

Many tedious and repetitive tasks, including file manipulations and organizing data, can easily be automated by writing a program in some scripting language. Yet, such languages usually require a good knowledge of regular expressions, and often their syntax does not correspond to modern high level programming languages. Additionally, small errors in the scripts can lead to malicious behavior, such as data loss [Mazurak and Zdancewic, 2007]. As an example, consider an attempt to remove all backup emacs files with the command `rm * ~`. For these reasons, many end-users search for help on on-line forums when they need to write some script [Stack Overflow, 2009, Stack Overflow, 2011, Super User, 2011]. We noticed that when a non-expert user seeks help in writing a script on user forums, she usually provides a few illustrative examples that convey her intentions about what this script is supposed to do. In this paper we describe StriSynth, a tool that successfully synthesizes scripts for string manipulations, based on given examples.

Programming by example (PBE) [Cypher and Halbert, 1993, Lieberman, 2001, Osera and Zdancewic, 2015, Feser et al., 2015] is a form of program synthesis. It works

[1] github.com/MikaelMayer/StringSolver

by automatically generating programs that coincide with given examples. This way the examples can be seen as an incomplete, but easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it might not correspond to user's intentions, due to this incompleteness in the specification. To address this issue, we propose a live programming environment [Burckhardt et al., 2013] as a framework for PBE. In this way, a synthesized script can be refined with every new provided example, and thus yields more interactive experience for the user. Interactive PBE allows end-users to provide a single example at a time, rather than guessing at the full example set that is necessary for synthesis. More significantly, it helps them to identify new examples that will accelerate convergence to the intended result/program. Additional features of StriSynth include:

- providing a human-readable description of a learned script after each interaction so users do not need to read synthesized code.
- The user can save the learned programs, and compose them together.
- The learned scripts can be exported to other formats, for example PowerShell.

The main motivation for StriSynth came from automating file manipulation tasks, as we had seen the potential for this type of system in Flash Fill's [Gulwani, 2013] synthesis of spreadsheet manipulations. While the use of scripting language such as sed, awk, bash or PowerShell requires a certain level of expertise, many tasks can be easily described using natural language or through examples. Inspired by encouraging feedback at a demo presentation of StriSynth [Gulwani et al., 2015b], we have since expanded functionality, reevaluated performance, and explored the potential of our new interface. Supported use cases include filtering files, grouping them, and performing various other commands on both individual files and groups of files. All of these operations work on a time scale acceptable for interactive use. We also conducted a user study (Sec. 3.6) that shows that with StriSynth the learning curve is a fast and pleasant experience for users. The study shows that users with a wide range of programming experience have a clear interest in being able to complete everyday scripting tasks with StriSynth.

This paper makes the following contributions.

- We designed and implemented a live programming environment for PBE, called StriSynth, that supports learning operations such as Transform, Reduce, Partition, Filter, and Split. In addition, StriSynth supports composition of operations.
- We describe algorithms for synthesizing the above operations from given input examples. These algorithms invoke the corresponding synthesis algorithms for the atomic data-types in a manner that allows leveraging existing proprietary PBE technologies for base data-types. We also describe a powerful synthesis algorithm for string transformations that extends Flash Fill with various novel capabilities.

- We designed and implemented a lightweight example-driven language which also provides Lifting (map) and Composition to combine these operators.
- We show how concepts from live programming can be applied to make an interactive PBE system. This allows users to incrementally create scripts while receiving feedback on their progress.
- We have an open-source, core JVM implementation of the existing Flash Fill algorithm, augmented with transformations involving numbers and counters. We designed robust program set intersection for learning counters. We show that these transformations can greatly help to design scripts dealing with file manipulations.
- We provide a script exporter which exports any script designed using StriSynth to a runnable PowerShell script. We also enhance the future reusability of the generated script by including the human-readable representation as a comment at its beginning.
- We evaluated our tool with a set of expert users, including PowerShell experts. We show that if the users have the choice between a scripting language and StriSynth, they will prefer StriSynth for more complex tasks.

3.2 Motivating Examples

We start by presenting a collection of tasks that demonstrate capabilities of StriSynth.

3.2.1 Generating HTML

Our first example comes from a StackOverflow post, where the users discuss complex and challenging regular expressions [Stack Overflow, 2009]. The user asked for a script that will create a link from every item in a directory. To better illustrate her intentions, the user provided two examples: for given files

```
Document1.docx
Document2.docx
```

the script should output

```
<A HREF='Document1.docx'>Document1</A>
<A HREF='Document2.docx'>Document2</A>
```

The other users on the forum suggested a solution that checks if the input string matches the regular expression `\(^[a-zA-Z0-9]+\)\.\([a-z]+\)`, and then replaces the input

string with `\<A HREF\=\"\1\.\2\" \>\1\<\/A\>`. A script doing this find/replace task using the popular tool `sed` looks as follows:

```
sed/\(^[a-zA-Z0-9]+\)\.\([a-z]+\)/\<a href\=\'\1
\.\2\' \>\1\<\/a\>/g
```

While it was very easy for the user to express her initial intentions by providing examples, the resulting script is arguably less readable, even for such a simple problem. Furthermore, small changes might require an entirely new regular expression to be written.

We next solve the same problem with StriSynth. We first provide an example showing what a script should do:

```
> NEW
> "Document1.doc" ==> "<a href='Document1.doc'>Document1</a>"
> val F = TRANSFORM
```

The keyword `NEW` denotes the start for learning of a new script, but it can be also used to restart the learning process, if the user decides to ignore the examples that she provided previously.

After the command `NEW` the user provides examples that convey her intentions. Based on this single example, StriSynth learns a string transformer, and we save it with the next command. Every learned function can be saved using the command `val name = ...`.

The tool then outputs an automatically generated description in English about a learned function F. In this particular case, it tells us that F takes an input string s and splits it into two parts: the one before the first occurrence of "." (s_1), and the one after. F then takes the constant string `<a href="`, concatenates it with s, then adds the constant string `'>`, followed by s_1, followed by the constant string `</a>`.

In general, automatically generated messages can sometimes be confusing. Therefore we can check how F works on different examples.

```
> F("Document1.docx")
<A HREF='Document1.docx'>Document1</A>
> F("Document2.docx")
<A HREF='Document2.docx'>Document2</A>
```

We observe that the learned transformer F is a function that exactly does what the user asked initially. However, it only takes a single string as input, while the user wanted a script that operates on a list of strings. In StriSynth there is an option to naturally

extend every operator "as map". If a function G has a signature $G : T_1 \to T_2$, then applying the operator `as map` will result in

$$G \texttt{ as map} : \text{List}(T_1) \to \text{List}(T_2)$$

We can now easily create a script which takes as input a list of file names and creates a list of HTML links.

```
> val linkify = F as map
```

Note the difference compared to the `sed` script: in StriSynth it was enough to provide a single representative example illustrating what should a script do, and then save this learned transformer as a map.

3.2.2 Auto-Incrementing a Variable in a String

The next example comes again from a StackOverflow post [Stack Overflow, 2011]: users discuss how to auto-increment a part of a regular expression containing a variable that has to be incremented every time. One user even suggests that *regular expressions are for matching strings - they are not for manipulating variables as the matching occurs.* However, there are solutions to this problem, consider, for instance the following perl script: `'s/[A-Z]{3},\d*/$count++.",$&,"/egi'`

To demonstrate how StriSynth deals with counters, we take the problem given at [Stack Overflow, 2011] and abstract away unnecessary details. In summary, the goal is to put a counter in front of an input string. The user provides an example

```
> "ABC" ==> "1ABC"
```

Based on only this example, StriSynth reports that the learned transformer puts a constant "1" in front of the input string. This, however, does not convey the user's intentions. In interactive manner she provides another example:

```
> "XYZ" ==> "2XYZ"
```

With this additional example StriSynth learns a new transformer that works for both examples: it puts a 1-digit counter in front of the input string.

3.2.3 Creating Photo Albums

Our next example was motivated by [Super User, 2011], but also by our own experience. The user wants to merge all the JPG files present in a folder, and would like to create

.PDF photo albums, but based on locations where photographs were taken. While forum user did not delivered a required batch script in [Super User, 2011], StriSynth can easily generate such a script. In addition, we use this example to illustrate the composite behavior of StriSynth.

The folder contains various JPG files, and our first task is to partition those files based on their location. The file names contain the location where they were taken. In StriSynth there is the *partition* function, which learns the criteria how to divide strings into different partitions. In this particular case we provide two examples of such a division:

```
> NEW
> ==> ("London-01.jpg", "London-02.jpg")
> ==> ("Paris-01.jpg", "Paris-02.jpg")
> val par = PARTITION
```

These two examples were sufficient to derive a partition function which groups strings based on their prefix upto the first "-" symbol. All the files will be partitioned according to that rule. When `par` is applied to a list of files, the result can, of course, contain more than two groups.

We now have the partitioning function `par`; the next goal is to learn a function which merges all the files in one partition. For this purpose we use the *reduce* operator.

```
> NEW
> ("NY-01.jpg", "NY-02.jpg", "NY-03.jpg",
   "NY-04.jpg") ==>
     "merge NY-01.jpg NY-02.jpg... NY.pdf"
> val rd = REDUCE
```

By inspecting the automatically generated description of `rd`, we confirm that the learned reduce operator precisely mimics our intentions. Note that in the resulting string we did not need to write down the entire string. We used the ellipsis operator (...), and StriSynth completed the string automatically.

Finally, we construct the initially required script: it combines `par` and `rd`. The `par` operator takes as input a list of JPG files, partitions them based on their locations, and returns this partition (as a list of lists). We next apply the `rd` operator on each class in the partition. The operator `andThen` is a composition operator.

```
> val toAlbums = (par andThen (rd as map))
```

3.2.4 Filtering

Our last example illustrates the *filter* operator in StriSynth. Consider the previous example, but when the directory contains all types of files. We first need to extract only the JPG files. In order to learn the criterion used for filtering, we provide positive and negative examples:

```
> NEW
> YES ==> ("a.jpg", "b.c.jpg")
> NO ==> ("a.txt")
> val takeJpgs = FILTER
```

StriSynth correctly learned that we are interested only in the JPG files. The results of filtering are usually combined with other operators:

```
> val final = (takeJpgs andThen toAlbums)
```

3.3 StriSynth Interface

StriSynth has taken inspiration from recent work in live programming[Burckhardt et al., 2013, Victor, 2012a] to introduce interactive PBE. In interactive PBE, users incrementally provide one example at a time, as opposed to need to provide a full example set. This small change in the interface is critical for StriSynth, as scripting is often an interactive process. In general, interactive PBE can be a more effective way to reach the intended audience, a casual novice user, that PBE often targets. StriSynth has implemented interactive PBE in a command line mode, through a standard REPL (Read Evaluate Print Loop) environment:

- **read** - StriSynth reads a single example from the user.
- **evaluate** - based on the example type StriSynth can either: start a new synthesis environment, refine an existing synthesized function, save/export the current synthesized function.
- **print** - the user receives feedback from the tool - it automatically generates an English description of the synthesized function
- **loop** - this process runs in a loop, allowing a continuous interaction between the user and StriSynth

The REPL framework is an instance of *live programming* that is being utilized here for interactive PBE. In the live programming environment the user actively participates in

code development and can immediately observe the changes in the code. Additionally, the user can inspect and test a synthesized script for correctness. If the script does not convey her intentions, the user provides more examples. This shift in the code interaction model is especially important for PBE systems targeting casual users.

Once a function has been synthesized to the user's satisfaction, it may be saved to a variable for further manipulation. Users may use high level composition operators to generate different scripts from their synthesized functions. These scripts and functions may also be exported to a PowerShell executable format.

3.3.1 Functions that StriSynth can Synthesize

In Sec. 3.2 we gave examples of some functions that StriSynth can learn. In this section we list them all. The generated functions usually take a string or a list of strings as input. They return a string, or a list of strings, or a list of lists of strings.

$$\begin{aligned}
\textsc{Transform}&: \mathsf{String} \to \mathsf{String} \\
\textsc{Reduce}&: \mathsf{List}(\mathsf{String}) \to \mathsf{String} \\
\textsc{Partition}&: \mathsf{List}(\mathsf{String}) \to \mathsf{List}(\mathsf{List}(\mathsf{String})) \\
\textsc{Filter}&: \mathsf{List}(\mathsf{String}) \to \mathsf{List}(\mathsf{String}) \\
\textsc{Split}&: \mathsf{String} \to \mathsf{List}(\mathsf{String})
\end{aligned}$$

To describe those functions in more detail, Transform is just a string transformation function. The Partition function takes as input a list of strings, and divides them into groups based on the partitioning criterion. Those groups are then returned as a list of lists of strings. Filter is a function which takes a list of strings as input and removes some of the elements based on the filtering criterion. Reduce is an operator that merges the elements in a list into a single string. The Split operator does the opposite: it returns a list of strings from the input string.

All those functions are generated based on the examples provided by the user. Once created, the functions may be exported directly as shell scripts, or can be saved and further composed and augmented using the following higher-order operators:

$$\begin{aligned}
\mathsf{as\ map}&: (a \to b) \to (\mathsf{List}(a) \to \mathsf{List}(b)) \\
\mathsf{andThen}&: ((a \to b), (b \to c)) \to (a \to c)
\end{aligned}$$

The as map operator lifts any transformation so that it can be applied to a List of the original input type. The composition operator andThen, also aliased with |, composes type compatible transformations.

3.3.2 Providing Examples for Synthesis

To learn the above operators the user must provide examples. To start providing new examples, the user first types NEW into the interpreter. As we have seen in Sec. 3.2 the input examples can have various different forms. The form of the first given example will direct the user to one of the five learning modes among TRANSFORM, REDUCE, PARTITION, FILTER or SPLIT. Once in a mode, the user either needs to continue to provide examples of the same mode, or enter NEW to switch modes and start learning a different function. To remove the last provided example, the user can enter CANCEL. However, this does not change the learning mode, even if all examples are canceled. If the user provides a non-corresponding example while in a specific learning mode, StriSynth returns syntactic error.

Below we list the syntax and mapping between the first example provided and the function to be learned. We denote a string variable with S.

Example given	Will learn
$S \Longrightarrow S$	TRANSFORM
$\mathsf{List}(S, \ldots, S) \Longrightarrow S$	REDUCE
$(S, \ldots, S) \Longrightarrow S$	REDUCE
$\Longrightarrow (S, \ldots, S)$	PARTITION
$\mathsf{YES} \Longrightarrow (S, \ldots, S)$	FILTER
$\mathsf{NO} \Longrightarrow (S, \ldots, S)$	FILTER
$S \Longrightarrow (S, \ldots, S)$	SPLIT

3.4 System Design

This section details how the system described in Sec. 3.3 is implemented. A high level overview of the system design structure is shown in Fig. 3.1. StriSynth uses three primary subcomponents: a) the example parser connects the user input to the system; b) the synthesizer acts as the learning module; and c) the converters translate the synthesizer's output into an operator that the user can see.

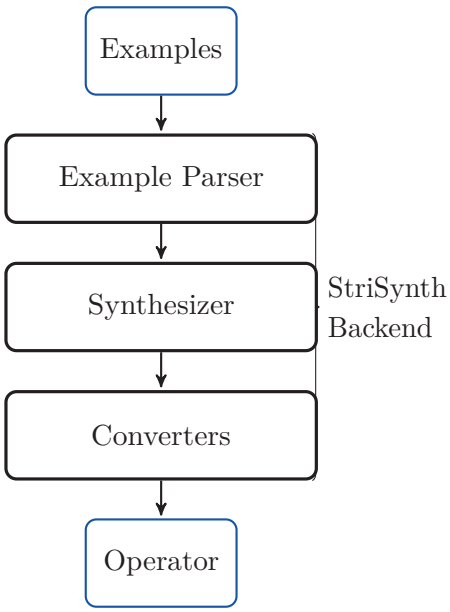


Figure 3.1 – StriSynth System Overview: the main three phases in the process of learning functions from the examples

3.4.1 Example Parser

The syntax used for examples consisting of only three main concepts.

1. The core syntactic form is the arrow symbol $\Longrightarrow$, as shown in Sec. 3.3.
2. An ellipsis string("...") is used as an optional final argument for examples to learn the Split operator. It indicates autocompletion. The ellipsis can also be used in the resulting string of examples for the Reduce operator.
3. Determining counters from examples: If a counter is present in user provided examples, StriSynth will detect and utilize them without any further user instruction. (more in Sec. 3.4.5)

We next provide some implementation details about parsing input examples.

Scala desugars the expression A $\Longrightarrow$ B as the method call A.$\Longrightarrow$(B). The $\Longrightarrow$ is implemented as an extension method for String (for Transform, Reduce and Split), List of strings (for Reduce), the objects YES and NO (for Filter). In the case of Partition, StriSynth defines a special stand-alone function $\Longrightarrow$ (not a method) so that it can be used as a static function, as in $\Longrightarrow$(P1, P2, ...). To learn the Split operator an example must provide at least 2 arguments to differentiate from Transform or Reduce.

The ellipsis construct inside example strings (...) helps StriSynth in detecting repetitive patterns. When StriSynth encounters a string of the form "s...", it analyzes the string s

with the goal of dividing into three parts: s_1, a separator $\oplus$, and s_2. Namely, it should hold that s is a concatenation of these three strings. Once we guessed the separator, substrings s_1 and s_2 should be the instances of some general expression, and thus we should be able to autocomplete the missing part of the string $s\ldots$.

In parsing the examples with the ellipsis operator the main task is to find the separator. Applying the exhaustive search would result in an $O(n^3)$ algorithm. For practical purposes, we developed various heuristics for boundaries and separators to reduce this cost.

3.4.2 Synthesizer

We build upon, but also extend the functionality of the Flash Fill[Gulwani, 2013] algorithm: only the TRANSFORM operator can be directly simulated using Flash Fill. All other operators present in StriSynth do not exist in Flash Fill. However, since most of our algorithms rely on learning string transformers from given examples, we are able to use our own, modified version of the Flash Fill algorithm as a black box. While our algorithms are structured to allow use of proprietary PBE systems, we chose to re-implement the Flash Fill algorithm ourselves. In order to understand how it works, let us denote the original Flash Fill's learning algorithm as *FF*:

$$FF\colon \mathsf{List}(\mathsf{String}^n \times \mathsf{String}) \to (\mathsf{String}^n \to \mathsf{String})\,.$$

FF takes as input examples a list of n-tuples of strings, and an additional string showing how those tuples are transformed. Flash Fill learns and returns a *transformer* that works correctly on the provided examples. StriSynth's main synthesis algorithm, denoted by S is a modified version of Flash Fill's learning algorithm, and has the following synthesis signature:

$$S\colon \mathsf{List}(\mathsf{List}(\mathsf{String}) \times \mathsf{Int} \times \mathsf{String}) \to (\mathsf{List}(\mathsf{String}) \times \mathsf{Int}) \to \mathsf{String}$$

This more general signature allows StriSynth to accept examples and generate transformers for all the operations mentioned in Sec. 3.3. The algorithm also extends Flash Fill with two additional features. First, instead of a tuple of strings for input, we let the input be a list of strings of possibly various length (List). This is useful for learning the REDUCE operator. Second, there is an integer (Int) added to every input. This integer enables learning counters (cf. Sec. 3.4.5). Additionally, it is also used for learning the SPLIT operator which internally extracts items based on an index.

The resulting output of S is called a transformer. All transformers, regardless of their

type, have the following unified signature:

$$S_T\colon (\mathsf{List}(\mathsf{String}) \times \mathsf{Int}) \to \mathsf{String}$$

3.4.3 Converter

In the last phase, a transformer is converted into an operator using a converter. For every operator there is a different converter. StriSynth's algorithms for the converters are given in Sec. 3.5.

$$\begin{aligned}
\mathsf{ConvertTransform}&\colon S_T \to (\textsc{Transform})\\
\mathsf{ConvertReduce}&\colon S_T \to (\textsc{Reduce})\\
\mathsf{ConvertPartition}&\colon S_T \to (\textsc{Partition})\\
\mathsf{ConvertFilter}&\colon (S_T \times \mathsf{String}) \to (\textsc{Filter})\\
\mathsf{ConvertSplit}&\colon S_T \to (\textsc{Split})
\end{aligned}$$

3.4.4 Language L_s: Syntax and Semantics

Motivated by the string transformation language introduced in [Gulwani, 2011], we describe a subset of that language for syntactic string manipulations, denoted by L_s. We further extend this language in Sec. 3.4.5 to allow the use of counters, and describe novel ways to use L_s in Sec. 3.5. Every operator in StriSynth is defined using L_s.

The language L_s is generated by the following context-free grammar, where t is the top-level symbol:

$$\begin{aligned}
t &::= e \mid \texttt{Concatenate}(e_1, \ldots, e_n)\\
\text{Atomic expr } e &::= v_c \mid \texttt{ConstStr}(s) \mid \texttt{SubStr}(v_c, p_1, p_2) \mid\\
&\qquad \texttt{Loop}(\lambda w.t)\\
\text{Position } p &::= k \mid \texttt{Pos}(r_1, r_2, c)\\
\text{Integer expr } c &::= k \mid k_1 w + k_2\\
\text{Regular expr } r &::= \epsilon \mid \tau \mid \texttt{TokenSeq}(\tau_1, \ldots, \tau_n)\\
\text{Token } \tau &::= C+ \mid \neg C+
\end{aligned}$$

To define the semantics of L_s, we explain each constructs of L_s individually. A better understanding of these constructs one obtains after examining the examples in Sec. 3.5.

An atomic expression e can either be a free input string variable v_c, a constant string

`ConstStr`(s), a substring of some input string v_c, or a loop construct. A string t belongs to L_s if it is either an atomic expression e, or is obtained by concatenating atomic expressions $e_1, \ldots, e_n$ using the `Concatenate` constructor. The substring expression `SubStr`(v_i, p_1, p_2) is defined partly by the position expressions (to be detailed below) p_1 and p_2, which must refer to valid positions within the string v_i. The `SubStr` constructor returns $\perp$ (undefined value) if p_1 or p_2 are outside the range of the string v_c, The loop construct `Loop`($\lambda w.t$) denotes concatenation of $e_1, \ldots, e_n$, where e_i is obtained from e by replacing all occurrences of w by i, and n is the smallest integer such that evaluation of e_{n+1} results in $\perp$.

A position expression represented by a non-negative integer constant k denotes the k-th position in the string. For a negative integer constant k, it denotes the $(n + 1 + k)$-th position in the string, where n is the length of the string. A position expression `Pos`(r_1, r_2, c) evaluates to a position t in the subject string s such that regular expression r_1 matches some suffix of $s[0 : t - 1]$, and r_2 matches some prefix of $s[t : n - 1]$, where n is the length of s, and $s[t_1 : t_2]$ is the substring of s between positions t_1 and t_2. Furthermore, if c is positive, then t is the c-th such match starting from the left side. Analogously, if c is negative, then t is the $(-c)$-th such match starting from the right side.

An integer expression c is either an integer constant k, or c has a form $k_1 w + k_2$, where w is a bounded integer variable.

A regular expression r is `TokenSeq`($\tau_1, \ldots, \tau_n$), which is a token sequence of at most three tokens. If the sequence is empty, we denote it as ϵ (empty string).

Tokens τ are constructed from character classes C: token $C+$ matches non-empty sequences of characters that belong to C, and token $\neg C+$ matches non-empty sequences of characters that do not belong to C. As an illustration, tokens $\mathsf{UpperTok} = [\mathsf{A} - \mathsf{Z}]+$, $\mathsf{NumTok} = [\mathsf{0} - \mathsf{9}]+$, and $\mathsf{AlphTok} = [\mathsf{a} - \mathsf{zA} - \mathsf{Z0} - \mathsf{9}]+$ match non-empty sequences of uppercase alphabetic characters, numeric digits, and alphanumeric characters, respectively.

3.4.5 Counters

Finally, we describe the counter construct, a new construct that we introduce in StriSynth and that does not exist in Flash Fill. Since the original Flash Fill algorithms do not contain the counters, we developed an entirely new algorithm for detecting and learning counters.

In Sec. 3.2.2, StriSynth was able to learn a counter. The counter construct belongs to atomic expressions in language L_s: `Counter`(*start*, *step*, *nDigits*). This atomic expression evaluates to a string representing some number, let us denote it with m. The resulting

string is the shortest representation of m with at least *nDigits* digits, filled with zeros on the left, if necessary.

In order to recognize and create new counters, StriSynth keeps an internal variable *index*. Its default value is zero, but it can also be explicitly set. The *index* variable increments whenever an example is added or an output is computed. Given the expression `Counter`$(start, step, nDigits)$, the number m is computed as $start + index \times step$.

When StriSynth detects a number n in the output string, it will store the possibility that this number n represents a counter. We use the following internal program set to store that fact: `CounterSet`$(nDigitSet, startSet, index, n)$. Here, *index* is the value of the *index* variable when the example was given and:

- *nDigitSet* is a set containing numbers that are candidates for the value of the *nDigits* variable. If n is a number starting with a zero, then *nDigit* is uniquely determined by the size of n. For example, if $n = 0012$, $nDigitSet = \{4\}$. On the other hand, if the first digit of n is non-zero, then *nDigit* is a number between the size of n and 1. For example, if $n = 243$, $nDigitSet = \{1, 2, 3\}$.
- *startSet* is a set containing numbers that are candidates for the value of the *start* variable. For example, if $n = 10$ at $index = 2$, then $startSet = \{8, 6, 4, 2, 0\}$. If $n = 10$ at $index = 3$, then $startSet = \{7, 4, 1\}$.

An important function in StriSynth's synthesis algorithm is computing the intersection of two program sets.

The first problem we encounter is finding the sets of numbers of digits can be intersected safely. The main issues arise with the intersection of the sets of starts. Consider the following scenario: let us suppose that at $index = 2$ the value of n is $n = 12$, and at $index = 3$ the value of n is $n = 13$. The first example returns the following possible starts $\{0, 2, 4, 6, 8, 10\}$, whereas the second example has the possible starts $\{10, 7, 4, 1\}$. Computing only the intersection of these two sets is not enough. Let us check whether 4 is a possible good start for both examples. It is not, because taking 4 as a starting counter will result that in the first example the step should be equal to 4, while in the second example the step should be equal to 3. This implies that the only possible start is 10.

Finally, we describe how to find the *start* variable. The intersection of two counters at the different values of the *index* variable can only yield one or zero possible *start* variables. The *start* variable can exist only if $n_2 - n_1$ is a multiple of $index_2 - index_1$. In that case, we can compute the new step by dividing the difference of numbers by the difference of indexes. The only start which is feasible would be given by $newStart = n_1 - index_1 \times step$. By computing the intersection of $\{newStart\}$ with the intersection of the starts, we can find the *start* variable.

3.5 Synthesizing Operations

We now show how to use the general synthesizer S described in Sec. 3.4 to synthesize each of the particular operations available in StriSynth. Each operation has its corresponding learning procedure. The list of learning procedures along with their signatures is shown in Fig. 3.2. For each learning procedure, we show how to manipulate the examples into a form acceptable for S, and how to use the transformer produced by S to synthesize the corresponding operation.

$$
\begin{aligned}
\mathsf{LearnTransform}&: \mathsf{List}(\mathsf{String} \times \mathsf{String}) \to \mathsf{String} \to \mathsf{String} \\
\mathsf{LearnReduce}&: \mathsf{List}(\mathsf{List}(\mathsf{String}) \times \mathsf{String}) \\
&\quad \to \mathsf{List}(\mathsf{String}) \to \mathsf{String} \\
\mathsf{LearnPartition}&: \mathsf{List}(\mathsf{List}(\mathsf{String})) \\
&\quad \to \mathsf{List}(\mathsf{String}) \to \mathsf{List}(\mathsf{List}(\mathsf{String})) \\
\mathsf{LearnFilter}&: \mathsf{List}(\mathsf{String}) \times \mathsf{List}(\mathsf{String}) \\
&\quad \to \mathsf{List}(\mathsf{String}) \to \mathsf{List}(\mathsf{String}) \\
\mathsf{LearnSplit}&: \mathsf{String} \times \mathsf{List}(\mathsf{String}) \\
&\quad \to \mathsf{String} \to \mathsf{List}(\mathsf{String})
\end{aligned}
$$

Figure 3.2 – List of learning procedures.

3.5.1 Transform and Reduce

Synthesizing Transform and Reduce is fairly straightforward, as they are nothing but special cases of transformers S_T. Indeed, as shown in Fig. 3.3, the learning procedures simply apply the corresponding converters, which specialize the transformers into appropriate types.

In pseudo-code listings we assume that some type conversions happen without saying. For example, in Fig. 3.3 pairs of strings are automatically promoted into triples of type $\mathsf{List}(\mathsf{String}) \times \mathsf{Int} \times \mathsf{String}$: the first string of the pair becomes a singleton list and populates the first component of the triple, the second string of the pair populates the last component of the triple, and the unused counter in the middle of the triple defaults to zero.

3.5.2 Partition

Pseudo-code for learning Partition is shown in Fig. 3.4. Recall that examples for Partition are given as a list of lists, where each inner list contains strings that should

```
LearnTransform =
  λ examples: List (String × String).
    return ConvertTransform(S(examples))

ConvertTransform = λ t: S_T, s: String. return t([s],0)

LearnReduce =
  λ examples: List (List (String) × String).
    return ConvertReduce(S(examples))

ConvertReduce =
  λ t: S_T, list: List(String). return t(list ,0)
```

Figure 3.3 – Pseudo-code for learning TRANSFORM and REDUCE. For brevity, in this and the subsequent figures we assume implicit type conversions where types do not match.

be grouped together. For each inner list l we non-deterministically[2] select a string e_l such that: (a) it is a common substring of the strings in l, and (b) it is unique among substrings selected for other inner lists. We form the input for S by conjoining strings from the inner lists with their corresponding substrings e_l.

The transformer obtained from S in this case maps a string into a classifier. The final PARTITION operation returned by ConvertPartition groups together strings from the input list if they share the same classifier.

In order to illustrate the learning process, we return to the photo album scenario from Sec. 3.2.3. In the scenario, the examples provided by the user correspond to the following list of lists:

[[London01.jpg, London02.jpg], [Paris01.jpg, Paris02.jpg]]

Common substrings that satisfy conditions (a) and (b) are London and Paris. We use them to construct the following input for S:

London01.jpg $\Longrightarrow$ London
London02.jpg $\Longrightarrow$ London
Paris01.jpg $\Longrightarrow$ Paris
Paris02.jpg $\Longrightarrow$ Paris

[2]In practice, the non-determinism is resolved by considering longest common substrings that satisfy the uniqueness condition. Longest common substring, as well as other substring extraction problems mentioned in this paper, can be efficiently solved using *suffix arrays* [Manber and Myers, 1993].

```
LearnPartition =
  λ lists : List(List(String)).
    for each l ∈ lists
      select substring e_l common to strings in l
    assume ∀ l, l' ∈ lists. l ≠ l' ⟹ e_l ≠ e_l'
    t = S( [([s],0,e_l) | l ∈ lists, s ∈ l] )
    return ConvertPartition (t)

ConvertPartition =
  λ t: S_T, list: List(String).
    m = empty map String→List(String)
    for each s ∈ list
      classifier = t([s],0)
      m[classifier].append(s)
    return m.values
```

Figure 3.4 – Pseudo-code for learning PARTITION.

When these input-output examples are given to S, it constructs a string transformation expression

$$T = \mathsf{SubStr}(v_1, 0, \mathsf{Pos}(\epsilon, \mathsf{NumTok}, 1))\,,$$

which extracts the prefix of a string up to the first occurrence of a numerical token. This is precisely the classifier that groups photos based on their location.

Note three points. First, while only giving examples for two groups, the synthesized transformer is able to correctly classify additional groups, e.g.

$$T(\mathsf{NY01.jpg}) = \mathsf{NY}\,.$$

Second, the user usually has to give at least one group with two examples, else it will put each string into its own group. She also usually has to give at least two different groups. Third, the determining substring is not necessarily the longest substring of a group. For instance, if the file names in the example were a1.jpg, a2.jpg, g1.jpg and g2.jpg, the longest common substring for both groups would be .jpg, which does not define a partition.

In the extreme case, it might happen that there are no common substrings that can uniquely distinguish a group. If that is the case, partitioning fails. However, since the user manipulates data in an interactive way, she can help the partitioning algorithm by providing a new set of examples.

```
LearnFilter =
  λ yes, no: List(String).
    select substring e_y common to strings in yes
    t = S( [([s],0,e_y) | s ∈ yes] )
    assume ∀ e ∈ no. t([e],0) ≠ e_y
    return ConvertFilter (t,e_y)

ConvertFilter =
  λ (t, s): S_T×String, list: List(String).
    result = []
    for each s' ∈ list
      if t([s'],0) = s then result.append(s')
    return result
```

Figure 3.5 – Pseudo-code for learning Filter.

3.5.3 Filter

On the first thought, filtering is conceptually just a special case of partitioning, and as such perhaps does not deserve to be singled out as a separate operation. However, there is a practical difference between partitioning and filtering. Namely, we expect each part in a partition to be homogeneous in a way that strings in a part share a common substring. In contrast, strings that are to be filtered out could be completely heterogeneous, so we cannot rely on them having common substrings. Therefore, the synthesis algorithm for Filter is slightly different.

The algorithm, shown in Fig. 3.5, accepts a list of positive examples called yes, and a list of negative examples called no. Similarly as before, we select a string e_y such that: (a) it is a common substring of the positive examples, and (b) the produced transformer does not transform any of the negative examples into e_y. The final Filter operation returned by ConvertFilter only leaves strings from the input list if they are transformed into e_y.

As with Partition, there might be no common substring of the positive examples that yields a valid transformer. If that is the case, filtering fails. Again, the user might be able to help the filtering algorithm by providing additional examples.

3.5.4 Split

Pseudo-code for synthesizing Split is shown in Fig. 3.6. Note that in this case we use numerical indices both when constructing the input for S and when converting the obtained transformer into the Split operation. When constructing the input for S, an

```
LearnSplit =
  λ s: String, list : List (String).
    t = S( [([s], k, s') | (k, s') ∈ indexed(list )] )
    return ConvertSplit (t)

ConvertSplit =
  λ t: S_T, s: String.
    result = []
    for each k=1,2,...
      r = t([s], k)
      if r is empty then break
      result .append(r)
    return result
```

Figure 3.6 – Pseudo-code for learning Split.

example is formed by conjoining the input string s, a string s′ from the input list list, and an index k of s′ in list. When converting the obtained transformer t, an index k is passed to t in order to extract the k-th component of the string s.

To better illustrate the process, consider the following string denoted by s:

img1.jpg; img2.gif; reportA.doc; reportA.pdf; reportB.doc;
report4.pdf; adw.doc; conf.mp3

The user wants to split the string into a list of strings:

[img1.jpg, img2.gif, reportA.doc, reportA.pdf, reportB.doc,
report4.pdf, adw.doc, conf.mp3]

To do so, she gives examples of the first two elements in the list: img1.jpg, img2.gif. The procedure LearnSplit constructs the following list to be passed to S:

$$[([s], 1, \text{img1.jpg}), ([s], 2, \text{img2.gif})]$$

The obtained transformer t transforms the string according to the following expression parametrized by the index k:

$$\text{SubStr}\Big(v_1, \text{Pos}(\epsilon, \text{NonSpaceTok}, k), \text{Pos}(\text{AlphTok}, \epsilon, 2k)\Big).$$

The expression reads as follows: extract a substring starting at the beginning of k-th non-space token, and ending at the end of $2k$-th alphabetic token. The non-space tokens

```
as map = λ f: A → B, list: List(A).
  result  = []
  for each a ∈ list
    result .append(f(a))
  return result

andThen = λ f: A → B, g: B → C.
  return λ a: A. g(f(a))
```

Figure 3.7 – Pseudo-code for higher-order operations.

in this case are img1.jpg;, img2.gif;, etc., and the alphabetic tokens are img, jpg, img, gif, etc. The function ConvertSplit returns a SPLIT operation that iterates over k, starting from $k = 1$, and evaluates t on s until the evaluation returns the empty string.

3.5.5 Higher-Order Operations

The pseudo-code for the higher-order operations as map and andThen is shown in Fig. 3.7. These operations do not require learning. as map is used to lift any element-wise transformation to lists of elements. It is written using the postfix notation. andThen, or its alias |, is an infix composition operator used to compose transformations with compatible signatures.

Note that if the learning algorithm for TRANSFORM or REDUCE recognizes a counter in provided examples, the counter is also incremented in the operations lifted by as map. Consider the following scenario:

```
> NEW
> "file.txt" ==> "file496.txt"
> "test.pdf" ==> "test498.pdf"
> (TRANSFORM as map)(List("doc.txt",
..> "report.pdf", "leg.jpg"))
```

The last command returns the list

[doc496.txt, report498.pdf, leg500.jpg] .

3.6 Evaluation

The main goal of our user study was to demonstrate that StriSynth, and programming-by-example in general, is a useful alternative to traditional coding approaches. In particular, we show that users prefer such a tool in various cases and to various degrees. While no one paradigm will fit all users, this study establishes the clear need for tools such as StriSynth when attempting to reach the broadest audience of potential programmers.

3.6.1 Study Design

The 50-minute study consisted of two parts: a tutorial and a questionnaire (available in appendix C page 197). The goal of the tutorial section was familiarize participants with both StriSynth and PowerShell. To this end, they completed the same three tasks with both tools, which were similar to the motivating examples presented in section 3.2. Once participants were well-acquainted with both PowerShell and StriSynth, they were asked to extrapolate their experience to more general situations. They were given a questionnaire consisting of 11 natural scripting tasks and asked users to choose between PowerShell, StriSynth, or their other favorite scripting language.

For the tasks on the questionnaire, we used a benchmark set built from selections from the online coding forums StackOverflow and Superuser [Stack Overflow, 2009, Stack Overflow, 2011, Super User, 2011]. The benchmark was built in 2013 as a separate investigative project to identify use cases for a PBE tool. All materials from the study, including results, are available online at http://santolucito.github.io/StudyData.zip.

We recruited 15 participants that had some prior experience with scripting languages, ranging from 2-7 on a seven point Likert scale, with an average of 4.86 and standard deviation of 1.75. Our participants were undergraduate and graduate students in STEM (Science, Technology, Engineering, and Math) fields, as well as professional academics and programmers (some of them being a Microsoft Most Valuable Professional (MVP)). Geographically, our participants came from the USA, Europe and Asia. Because of a relatively low response rate and the fact that participants' profession was not tied to data collection for anonymization guarantees, we cannot provide exact numbers on profession distribution.

We setup a remote machine with all necessary software installed and created user accounts for all participants. Users were allowed access to any reference materials they needed, including searching the Internet. This limited the variability in the testing environment between users, while still providing an accurate real-life simulation.

3.6.2 StriSynth Usability

The goal of this study is not to infer correlations between experience and preference to StriSynth, nor is it to show that StriSynth is a replacement for a more traditional scripting environment. Instead we remark that a wide range of users preferred to complete many tasks with StriSynth, demonstrating a strong interest in this research direction among real programmers. We present a few of the tasks from the study where the participants indicated a clear preference for our tool:

- A task similar to the one given in Sec. 3.2: the task was to generate a link to all documents in the folder.
- A task to generate a list of the file names in a folder. The list should contain the file name without the extension. For example, a directory with foo.txt, bar.pdf, and foo.bar.doc will yield [foo,bar,foo.bar].
- Users were asked to reorder commands, according to some rules. For instance, a command `./c.exe in.txt out.txt;` should be transformed into the command `./c.exe out.txt in.txt;`

Figure 3.8 shows how many times users would choose either StriSynth or PowerShell or some other scripting language. The users' responses have been semi-sorted by their choices in order to more clearly communicate the spectrum of preferences among participants. All users except one selected StriSynth for some set of tasks. The middle section illustrates the personal nature of programming by example.

Table 3.1 details the number of users that selected each tool over all the tasks in the study. While StriSynth seems overwhelmingly well suited for tasks like "Linking to items", where one can easily convey intentions through an example, it is more interesting to see that there are always some users that found StriSynth useful. For every task we presented, there existed some users who preferred to use StriSynth.

3.6.3 Feedback

The survey also collected qualitative data from the user in the form of open-ended questions. Many users enjoyed the new paradigm because *"giving examples is intuitive"*, and *"you just write what you want it do."* The live coding aspect and immediate feedback was helpful to some, who said it *"is very nice - I like the way it tries to explain what we write."* Some users had trouble getting started with the completely new paradigm. One user said *"I found difficult the first 30 minutes trying to figure out how everything worked."* This steep learning curve had a high payoff though, as one user stated *"once I had understood the basics, I enjoyed how quickly the language was learning."*

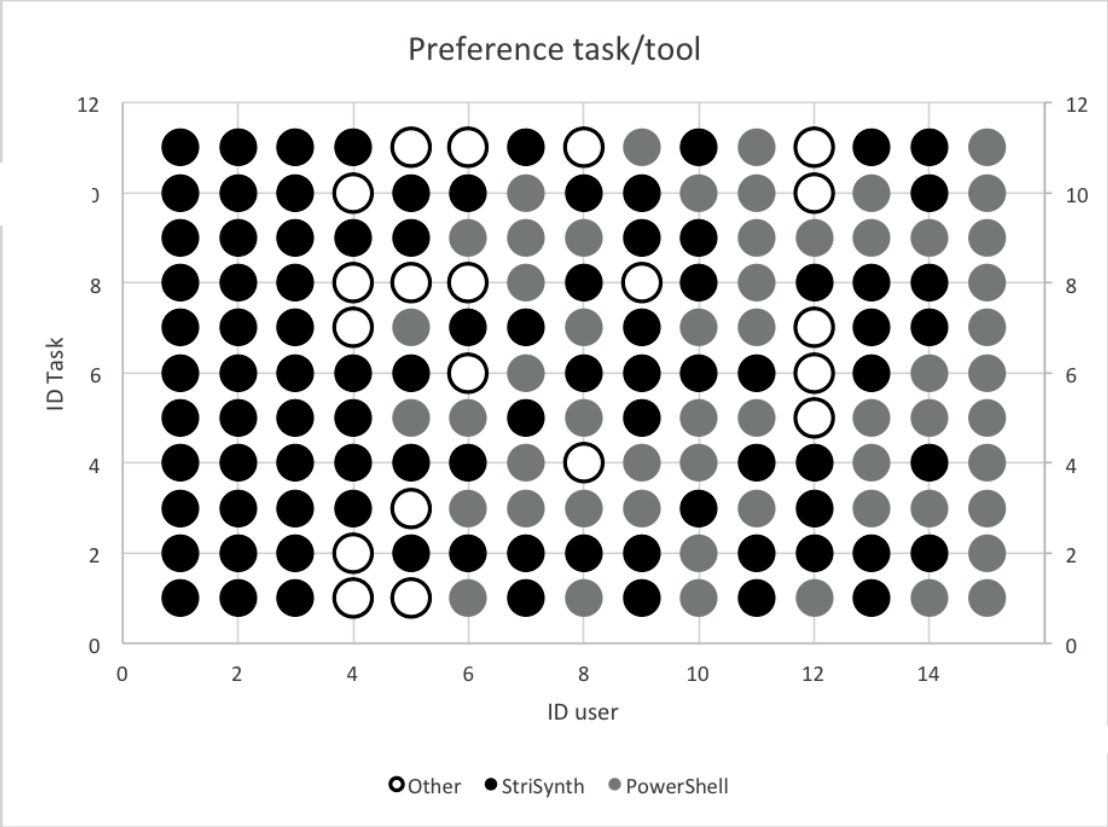


Figure 3.8 – Tool of choice for all users.

One question was: "Would any of your answers [about tool choice] changed if StriSynth were integrated into an existing scripting language." Ten out of fifteen of our participants made an affirming statement in response. Some of the answers included *"...the result would be flawless"*, *"...would make it much more attractive"*, and *"if you made it more user friendly it would be awesome."* This is encouraging as it provides a clear direction for future work that will have an impact on usability.

3.6.4 Performance

While our previous work has shown that earlier version of StriSynth can quickly and accurately synthesize functions[Gulwani et al., 2015b], in addition to usability, we also tested the StriSynth's performance. We expanded the initial timing result with more benchmarks. Additionally, rather than giving to the system a full example set, we record the interaction time between each example given. Table 3.2 presents StriSynth's performance on the tasks presented in the user study. StriSynth was able to synthesize the correct algorithm based on only one or two examples for nearly all tasks. While we wish to continue to make StriSynth faster, we seem to have reached the critical threshold of being able to synthesis a script as quickly as a user can write it by hand.

ID	Task	PowerShell	StriSynth	Other
11	Merging a book	4	8	3
10	Printing pdfs	2	8	5
9	Art Project	0	7	8
8	Move	4	8	3
7	Initials	2	8	5
6	Extract Filenames	2	10	3
5	Report Building	1	6	8
4	Reorder commands	1	9	4
3	Health Data	1	6	8
2	Linking to items	1	12	2
1	Automated Emails	2	7	6

Table 3.1 – Number of users choosing a particular tool (PowerShell, StriSynth, or some other language) for the given scripting tasks.

ID	Example	1st	2nd	3rd
11	Merging a book	40.8	-	-
10	Printing pdfs	9.9	-	-
9	Art Project	4.4	-	-
8	Move	0.170	0.130	-
7	Initials	0.330	1.7	-
6	Extract Filenames	0.013	0.015	0.180
5	Report Building	7.5	22.7	-
4	Reorder commands	16.2	47.1	-
3	Health Data	15.9	17.0	-
2	Linking to items	4.4	-	-
1	Automated Emails	15.1	-	-

Table 3.2 – Amount of time (in seconds) needed at each step for the given scripting tasks. A blank box indicates the correct script had been synthesized and no further examples were needed.

3.7 Limitations

As shown in Sec. 3.3 StriSynth relies on the Flash Fill algorithm. Although we have extended and re-implemented it as an open-source tool in Scala, there are still some potential limitations. Flash Fill only works on strings as a base dataype. We have extended it with some support for integers with the counter construct here, but more general applications will require integrating full integer theory support. Additionally, we may need to explore structural changes to the Flash Fill algorithm to reduce synthesis times.

Although the user study has yielded promising results, it has three potential shortcomings - a limited sample size, convenience selection, and the need-to-please phenomena.

Our sample size of 15 is not large enough to draw statistically significant claims for this data, however it has provided a convincing support for the arguments above.

Convenience selection can be an issue if the set of users selected systematically differs from the target population. Since our participants answered an open call and came from universities and companies across the world, there is no reason to believe this has affected our results.

The need-to-please phenomena drives participants to slightly favor a researcher's work, even if the participant has no association to the researcher. There is a possibility this has affected our results - and although the effect would be small, it should be eliminated in future user studies. We could present user's with a choice between StriSynth, and SuperShell, which would be a thin wrapper around PowerShell. We would claim that both are our tools. In this way, users would be asked about a preference between two tools of the researchers creation, thereby eliminating the need-to-please.

3.8 Related Work

Modern methods of learning by example are based on program intersection [Menon et al., 2013] until the user finds the program to be correct. Gulwani et al. [Gulwani et al., 2012] prove that this integrates well into spreadsheet manipulation, whether for strings [Gulwani, 2013, Gulwani, 2012a, Gulwani, 2011], number, table transformation [Gulwani, 2012b] or look-up [Singh and Gulwani, 2012a]. Topes [Scaffidi et al., 2008] system lets users create abstractions (called topes) for different data present in a spreadsheet. The user uses a GUI to define constraints on the data, and to generate a context-free grammar that is used to validate and reformat the data.

The differences between the mentioned approaches are in the particular domain where the synthesis is happening and the particular search technique that is used for synthesis. In case of Flash Fill [Gulwani, 2011, Gulwani, 2013], the domain is the string transformations

described in Section 3.4, and the search technique is based on version-space algebra [Lau et al., 2003b]. Interaction and evaluation developments aside, our work extends upon Flash Fill by supporting more operations (Partition, Split, Reduce). While synthesizing these operations can be reduced the to the basic Flash Fill learning problem, novel techniques were needed to support counters.

The Myth[Osera and Zdancewic, 2015] and Λ^2[Feser et al., 2015] systems support PBE for inductively defined data-types. We may be able to leverage this approach to support more complex script synthesis for the object-oriented paradigm of PowerShell. These systems however, require the user to specify a full example set, then compile the examples to code. The user must manually inspect the synthesized code to verify that it is complete to the user's intention. Although these systems target a different domain than StriSynth, they could still benefit greatly from interactive, live programming approach.

Instead of providing specification in terms of examples or demonstrations, specification can be given in more traditional ways. InSynth [Gvero et al., 2011, Gvero et al., 2013], CodeHint [Galenson et al., 2014] and more recently the C# code snippets on demand [Wei et al., 2014] aim at providing code snippets based on the context such as the inferred type or the surrounding comments. Leon [Kuncak et al., 2012] and Comfusy [Kuncak et al., 2010b, Kuncak et al., 2010a] synthesize code snippets based on complete specifications expressed in the same language that is used for programming. Sketch [Solar-Lezama, 2008] takes as input an incomplete program with holes, which it tries to fill with values that ultimately make the program meet the specification. In contrast to these code-generating solutions, StriSynth does not take a context or a complete specification into account—it only needs input-output examples. Thus, it is more suitable for isolating and solving local problems.

3.9 Conclusions

We have presented a tool for the synthesis of functions in a live coding environment that supports immediate feedback to the user. New algorithms presented here enable our tool to learn complex functions and features, such as partitions and counters. StriSynth also supports the composition of synthesized functions by using high-order operators such as andThen and asMap. Together, these developments empower users to create complex scripts by providing simple examples while they receive constructive feedback from the system.

Our user study suggests that this is a promising path for programming-by-example, sparking the interest of novice and expert users alike. An understanding of the user's needs is critical for the sufficient advancement of high-level languages. The ultimate goal is, of course, to give the user language control that is indistinguishable from magic; and as one user said of his experience with StriSynth, *"I liked the magic."*

Page 48 until this page 73 contained the content of the paper "Composite Scripting Using Examples". We will now add our own comments.

B. Epilogue to StriSynth

B.1 Usages of StriSynth

This automated approach to renaming files is so impressive that I use it each time I want to describe the power of programming-by-example. Many of StriSynth's novel features would select it as a candidate for integration in future file systems, although much more work is needed.

Recent updates in StriSynth enabled users to write programs by example in a syntax more friendly than before, such as:

```
("document.doc" ==> "<a href='document.doc'>document</a>")("report.pdf")
```

This is a valid program that, after execution, returns the following string:

```
"<a href='report.pdf'>report</a>"
```

Such an easy-to-read but valid function definition can be found nowhere, in any basic programming language.

Recently, I added the automatic insertion of the operator as map when it is obvious that an operation should be applied on a collection. This means that, for example, users can easily produce a PowerShell program scriptgen.ps1 that, when we execute it, produces a script to rename JavaScript files (.js) to TypeScript files (.ts). A way to create this program is to execute the following code in the StriSynth command line:

```
{ OK ==> ("main.js", "aux.js"); NOTOK ==> "main.doc"} andThen
"main.js" ==> "mv main.js main.ts" in PowerShell to "scriptgen.ps1"
```

Executing the program scriptgen.ps1 in a folder containing index.js, main.html and protocol.js displays the following script ready-to-execute:

```
mv index.js index.ts
mv protocol.js protocol.ts
```

I used StriSynth once for myself, and once for my own father who had to rename many files at once and asked me to do it for him. It worked well. As I was working on real files, I still felt slightly insecure about renaming them automatically, so I first created a copy of them and then I applied my tool. This experience illustrates well the problem of trust. It was not as if I saw that the computer always found the correct program before: I did see the computer fail many times to find the correct program, so it was possible, if not probable.

How could we prevent this situation? To solve the ambiguity, my first idea was the program feedback, i.e. to make an effort to paraphrase programs in order to display them, so that we could decide if the programs were right or not. However, when the program became complicated, especially with advanced regular expressions, it was pointless to show the program. I thought about the output feedback as well. When I wrote the

Windows extension to display the transformation in a balloon, before executing it, the system would display a few examples in the balloon [3]. But the space there was limited.

As a result, the interaction was missing some guidance to find the right program. I would have loved StriSynth to guide me by telling me if there were ambiguities, but I did not know at the time how to program such a guide. At the time of writing this thesis, after the work of the next chapter, the solution is much clearer; we need only to run different programs obtained by the computer, not just the first arbitrarily ranked one. This way, we can compare their results and ask further questions. The question about how to choose these programs, and what input on which to compare them, is still not evident at all.

In the next chapter, we will investigate another programming-by-example problem: It consists in extracting data from a file. Due to the experience I gained while writing this chapter, I implemented this clarification questioning. My co-authors later called it "conversational clarification". It seemed that interaction is truly about communicating bidirectionally with the computer, not just being able to give iteratively order by order, as we assumed in this chapter.

B.2 Discussion

We now answer the questions we asked in the introduction (see page 9) with respect to the work of this chapter.

Question	StriSynth
Is it faster/easier to conduct tasks using programming-by-example?	Yes/No
Do users need to see the generated program?	Yes
Do users need to take on the generated program?	No
Is the paraphrased version of the program useful?	Yes
Is it more reliable when the program is shown?	Yes
Is it more reliable when the computer asks questions?	N/A

The comparative summary for all the papers is available on page 163.

It would be definitely faster to use StriSynth for certain tasks such as renaming files using a counter than regular programming languages. However, for tasks such as complex partitioning, filtering and flattening, the learning curve prevents its adoption. Most users want to see the program, as many of our benchmarks came from forums of programmers who asked for code snippets. Displaying the program in a paraphrased version makes it shorter and more meaningful, although for complex partitioning the paraphrasing is not that clear. Using the comments, we found some evidence that showing the program

[3] youtu.be/rbhAv3uBFqw

helps users to trust them. Programmers usually do not need to take on the generated program as they are usually run once. They might, however, want to store it for later reuse. We did not experiment with questioning at this point.

4 Displaying Programs and Asking Questions

"Ne comprennent que ceux
qui ont envie de comprendre."

"Only they understand,
those who want to understand."
Bernard Werber - L'Empire des Anges

A. Prologue

The interface was looking awesome, but as soon as I tried to use it, it broke down. This is the first experience I had when testing the original interface of Flash Extract [Le and Gulwani, 2014], a very recent programming-by-example technology inspired from Flash Fill. Flash Extract enables the user to select and color some part of a text, and to automatically find a program that extracts the parts and generalizes to extract others.

The idea still looked great to me. I started to work on it, during a one-year internship at Microsoft Research. I wanted to apply what I thought were potential successes from the two previous chapters. I wanted to show the inferred program, as I did for Pong Designer and later for StriSynth.

Hence, the first step was to design a homemade converter of the programs to HTML and a way to view them side by side with the text on which to perform extraction. Soon enough, I realized that the program structure closely resembled the structure of the set of all possible programs found by the machine (Version Space Algebra [Lau et al., 2003a]),

My second major step was to make "alternative programs" be available at every program node, in a contextual select box, so that the user could choose the alternative. Then, I realized that there were too many program nodes to click on. Each time, I needed to find quickly which one deserved my attention. For this, I implemented a first program comparer. It would take the original program and the alternative program, run both of them and compare the output. If the output differed, I would add a sign close to the program node, indicating that there were not any obvious alternatives. Then, I added a

feature to temporarily highlight the output of an alternative that a user hovers with the mouse, so that I could quickly see the difference between them.

I did not think about clarification questioning at the beginning, this came later, organically. When I observed that my main workflow was to observe the program, click on the nodes with a sign, hover on the alternatives and add a new example based on these results, I decided to create a feature for that.

My third major step was therefore to create a program that identifies differences between the outputs of programs it chooses by itself, and present these outputs in a view to me so that I could choose which one was correct. This was the icing on the cake. No more program, and a clear question, understandable by anyone. This made the success of the user study.

Before writing this paper, we won first prize for our presentation of FlashProg at an internal event of Microsoft Research. Later, one of the most influential people from Microsoft, having heard of our success, asked for a private demo. He gave the comment when looking at this interface (and at other similar technologies presented by my manager that day): “We need more technologies like that”.

User Interaction Models for Disambiguation in Programming by Example

The following pages 80-105 are the content of the peer-reviewed paper named “User Interaction Models for Disambiguation in Programming by Example”
(DOI: 10.1145/2807442.2807459) appearing in the User Interface Software Technologies conference (UIST) in 2015, published by ACM. We reproduce this article in this thesis with the permission of all the authors, who are myself, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn and Sumit Gulwani.

Contributions: *Drawing inspiration from Gustavo’s previous work, I created the FlashProg interface from scratch, with custom paintbrush highlighting algorithms, highlight preview, label organization preview, and a table of results. I also invented and maintained the language for CSS-inspired paraphrasing into trees and their instances. I designed and implemented the program navigation view. I programmed and tweaked the back-end and front-end of the clarification engine. I prepared the interface for recording the user study, and I wrote the tutorial.*

Programming by Examples (PBE) has the potential to revolutionize end-user programming by enabling end users, most of whom are non-programmers, to create small scripts for automating repetitive tasks. However, examples, though often easy to provide, are

an ambiguous specification of the user's intent. Because of this, a key impedance in adoption of PBE systems is the lack of user confidence in the correctness of the program that was synthesized by the system. We present two novel user interaction models that communicate actionable information to the user to help resolve ambiguity in the examples. One of these models enables the user to effectively navigate between the huge set of programs that are consistent with the examples provided by the user. The other model uses active learning to ask directed example-based questions to the user on the test input data over which the user intends to run the synthesized program. Our user studies show that each of these models significantly reduces the number of errors in the performed task without any difference in completion time. Moreover, both models are perceived as useful, and the proactive active-learning based model has a slightly higher preference regarding the users' confidence in the result.

4.1 Introduction

Today, billions of users have access to computational devices. However, 99% of these end users do not have programming expertise and they often struggle with repetitive tasks in various domains that could otherwise be automated using small scripts. Programming-by-examples (PBE) [Lieberman, 2001, Cypher and Halbert, 1993] has the potential to revolutionize this landscape since users can often specify their intent using examples as has been observed on various help forums [Gulwani et al., 2012]. PBE involves techniques that generalize example behaviors on concrete inputs provided by the user into programs that can operate on new unseen inputs. PBE has traditionally been applied to synthesizing small programs in various domain-specific languages (DSLs) such as string and table transformations [Gulwani et al., 2012] and data extraction [Le and Gulwani, 2014]. PBE has been pursued in various communities including programming languages [Leung et al., 2015, Feser et al., 2015, Barowy et al., 2015], inductive programming [Gulwani et al., 2015a], machine learning [Menon et al., 2013], artificial intelligence [Raza et al., 2014], and databases [Shen et al., 2014]. Work in these communities has focused on addressing one of the key challenges in PBE, that of efficiently searching the huge state space (potentially infinite) of programs defined by the underlying DSL for a program that is consistent with the user-provided examples.

However, not much attention has been given to dealing with another key technical challenge in PBE, that of dealing with ambiguities. Examples are an ambiguous form of specification in the sense that there can be different programs that are consistent with the provided examples, but these programs differ in their behavior on some other inputs. The underlying PBE system might end up synthesizing an unintended program that is consistent with the examples provided by the user but does not generate the intended results on some other inputs that the user cares about. In 2009 Tessa Lau presented a critical discussion of PBE systems noting that adoption of PBE systems is not yet widespread, and proposing that this is mainly due to lack of usability and confidence

in such systems [Lau, 2009]. complementary user interaction models for PBE that help increase user confidence in the underlying system.

Motivational Real-world PBE Case Studies Recently, a first mass-market PBE product was released in the form of the *FlashFill* feature in Microsoft Excel 2013. It allows end users to automate sophisticated string transformations in real time from one or more user-provided examples [Gulwani, 2011]. While the PBE engine behind FlashFill received many positive reviews from popular media (bit.ly/flashfill) the user interface for FlashFill leaves a lot to be desired. John Walkenbach, an author renowned for his Excel textbooks, labeled FlashFill as a "controversial" feature. He wrote "It's a great concept, but it can also lead to lots of bad data. (...) Be very careful. (...) [M]ost of the extracted data will be fine. But there might be exceptions that you don't notice unless you examine the results very carefully." (spreadsheetpage.com/index.php/blog/C10)

Another mass-market PBE product, recently released as part of the Windows 10 preview, is the *ConvertFrom-String* feature in PowerShell (bit.ly/convertfrom-string). It allows end users to extract structured data out of semi-structured text/log files from one or more user-provided examples. It is based on the FlashExtract PBE engine that can synthesize sophisticated data extraction scripts in real time [Le and Gulwani, 2014]. It was well-received by various Microsoft MVPs (Most Valued Professionals), who described it as "New kid on the block", "This is super cool !!", "must admit that this cmdlet is to me one of the best improvement that came with WMF5.0 and PowerShell v5". (bit.ly/flashextract) However, the MVPs also complained that they had no visibility into the process for debugging purposes. This prompted Microsoft to release an improved version of FlashExtract that provided a flag to display the top-ranked program synthesized by FlashExtract. An MVP still complained: "If you can understand this, you're a better person than I am."

User Interaction Models We propose two novel user interaction models that aim to alleviate above-mentioned transparency concerns by exposing more information to the user in a form that can be easily understood and acted upon. These models help resolve ambiguity in the example-based specification, thereby increasing user's trust in the results produced by the PBE engine.

Program Navigation: A typical PBE engine operates by synthesizing multiple programs that are consistent with the examples provided by the user, and then ranking the programs in order of their likelihood of being the intended program [Gulwani et al., 2012]. A typical PBE interface would pick the top-ranked program and use it to automate the user's task; possibly this top-ranked program can even be shown to the user. We propose a novel user interaction model, called Program Navigation, that allows the user to navigate between all programs synthesized by the underlying PBE engine (as opposed

to displaying only the top-ranked program) and to pick one that is intended. The number of such programs can usually be huge (several powers of 10 such as 10^{30} [Gulwani et al., 2012]). However, these programs usually share common sub-expressions and are described succinctly using version space algebra based data structures [Polozov and Gulwani, 2015]. We leverage this sharing to create a navigational interface that allows the user to select from different ranked choices for various parts of the top-ranked program. Furthermore, these programs are paraphrased in English for easy readability.

Conversational Clarification: We propose a complementary novel user interaction model based on active learning, called Conversational Clarification, wherein the system asks questions to the user to resolve ambiguities in the user's specification with respect to the available test data. These questions are generated after the PBE engine has synthesized multiple programs that are consistent with the user-provided examples. The system executes these multiple programs on the test data to identify any discrepancies in the execution and uses that as the basis for asking questions to the user. The user responses are used to refine the initial example-based specification and the process of program synthesis is repeated.

FlashProg Framework for Data Manipulation We have implemented the above two user interaction models in a generic manner in a UI framework called FlashProg. The FlashProg framework provides UI support for several PBE engines related to data manipulation, namely FlashFill [Gulwani, 2011], FlashRelate [Barowy et al., 2015], FlashExtract [Le and Gulwani, 2014], and FlashWeb. Even though PBE has been applied to various application domains, we focus our attention in this paper on data manipulation, which we believe is one of the most impactful applications for PBE. Data is locked up in semi-structured formats (such as spreadsheets, text/log files, webpages, and PDF documents), which offer great flexibility in storing hierarchical data by combining presentation/formatting with the underlying data model, but make it extremely hard to manipulate that data. PBE holds the promise of enabling a delightful data wrangling experience because many tedious data manipulation tasks such as extraction, transformation, and formatting can be easily described using examples.

The FlashProg UI builds over the STEPS approach [Yessenov et al., 2013] to PBE, wherein the user breaks down a sophisticated task into a sequence of simpler steps, and each step is automated using PBE. We conducted a user study, where we asked participants to extract structured data from semi-structured text files using FlashProg. We observe that participants perform more correct extraction when they make use of the new interaction models. To our surprise, participants preferred Conversational Clarification over Program Navigation slightly more even though past case studies suggested that users wanted to look at the synthesized programs. We believe this is because Conversational Clarification is a *proactive* interface that asks clarifying questions, whereas Program Navigation is a *reactive* interface that expects an explicit correction of a mistake. This paper makes the

following contributions:

- We propose a user interaction model for PBE called Program Navigation. It lets the users browse the large space of programs that satisfy the user specification by selecting ranked alternatives for different program subexpressions.
- We propose another complementary user interaction model for PBE called Conversational Clarification. It involves asking directed example-based questions to the user, whose responses are then automatically fed back into the example-based specification model.
- We present a generic framework called FlashProg that implements Program Navigation and Conversational Clarification on top of any PBE engine. We have used FlashProg to develop user interfaces for four different PBE engines.
- We present results of a user study that evaluated our two user interaction models. We discover that both models significantly reduce the number of errors without any difference in completion time. Both models are perceived as useful, but Conversational Clarification has a slightly higher preference w.r.t. the users' confidence in the result.

4.2 Related work

FlashProg user interface is inspired by that of the STEPS system [Yessenov et al., 2013] that uses hierarchical structure coloring for text extraction and manipulation. STEPS showed the usefulness of PBE systems for text processing: STEPS users completed more tasks and were faster than conventional programmers. For disambiguation and converging to the desired task, STEPS supports two interaction mechanisms: (i) provide additional mock input-output examples that capture specific intents and corner cases, and (ii) navigate through a flattened list of a small set of programs (paraphrased in English). Since the DSLs supported by FlashProg are more expressive, there is often a huge number of programs that are consistent with few examples, which makes the interaction model of navigating the flattened list of programs unusable. Providing mock input-output examples puts additional burden on users to first identify why the system is learning an incorrect program and then construct specific examples to avoid learning them. FlashProg provides two new interaction models to alleviate this problem: 1) Program Navigation to browse the set of learned programs (paraphrased in English) in a hierarchical manner, and 2) Conversational Clarification to ask users to select the desired output on inputs for which the system has learned multiple interpretations.

Wrangler [Kandel et al., 2011] is an interactive system for data transformations on tabular data. It automatically suggests a ranked list of paraphrased transformations based on the context of user interactions. A user can then navigate the space of suggested

transformations in three ways: (i) by providing additional examples, (ii) by selecting an operator from the transform menu, and (iii) by editing the parameters of the suggested transforms. Wrangler's language is aimed at data cleaning and transformation, but not for extracting data from semi-structured sources. Moreover, the new interaction models of Program Navigation and Conversational Clarification can augment and complement Wrangler's interaction model.

LAPIS [Miller and Myers, 2002] is a text-editor that incorporates the concept of *lightweight structure* to recognize the text structure using an extensible library of patterns and parsers. Given positive and negative examples, LAPIS learns a pattern in a language called *text constraints* (TC), and highlights other matches in the file. This enables users to perform multiple selections and simultaneous editing to apply the same set of edits to a group of elements. LAPIS does not has good support for nested and overlapping regions, which are essential for data extraction tasks. LAPIS also introduced the idea of outlier detection for finding atypical pattern matches to focus user's attention for potential incorrect generalizations [Miller and Myers, 2001], which is related to the Conversational Clarification interaction model. The main difference between the two is the way in which the match discrepancies are computed. LAPIS models pattern matches as a list of binary-valued features and computes outlier matches based on their weighted Euclidean distance from the feature vector of the median match. FlashProg uses program semantics to identify ambiguous examples, where the highly ranked learnt programs generate different outputs on the examples.

Amershi et al. [Amershi et al., 2009, Amershi et al., 2011] have explored two strategies for soliciting effective training examples in interactive ML systems. The first strategy of *global overview* selects a subset of training examples that maximizes the mutual information with the high-dimensional vector space of the examples, and is most representative of the training set. The second strategy of *projected overview* projects examples onto a set of principal dimensions and then selects examples that illustrate variation amongst those dimensions. Our Conversational Clarification model presents a complementary technique for selecting training examples to learn a richer class of programs (as opposed to classifiers) based on the semantics of the learnt programs.

Several PBE-based text manipulation systems exist. FlashFill [Gulwani, 2011] learns syntactic string transformations (involving concatenation of regex-based substrings) from few examples. SmartEdit [Lau et al., 2001] automates text processing tasks from demonstrations by interactively navigating the space of learned programs (represented using a version-space algebra) using a mixed-initiative interface. Visual AWK [Landauer and Hirakawa, 1995] provides a graphical environment to drag and drop relevant text selections to learn patterns based on trial and error demonstrations. It allows users to separately learn conditionals and edit the learned programs graphically. Peridot [Myers and Buxton, 1986] allows users to interactively create graphical user interfaces by demonstrations. The TELS [Witten and Mo, 1993] system records a trace, generalizes

it, and then executes and extends the generated program based on user feedback. Marquise [Myers et al., 1993] lets users provide example actions to create user interfaces and uses a feedback window to show the inferred operation using english sentences with buttons that can be pressed to pop up the list of alternative options. Many of these systems do not expose the learned programs to the user and depend on manual inspection of generated outputs for validation. However, some systems such as SmartEdit, Peridot, Marquise, and Visual AWK do expose the learned programs, but the class of transformations supported by them are limited and are not expressive enough for learning hierarchical extraction of nested records.

FlashProg is based on automated program synthesis. Programs are synthesized in DSLs that are expressive enough to encode most common tasks, but at the same time concise enough for efficient learning. The synthesis algorithm uses a divide-and-conquer based strategy to decompose the original learning task to smaller sub-tasks [Polozov and Gulwani, 2015]. This approach has been used to develop several PBE systems in the domains of syntactic string transformations [Gulwani, 2011], semantic string transformations [Gulwani et al., 2012], data extraction from semi-structured sources [Le and Gulwani, 2014], and transformation of semi-structured tables [Barowy et al., 2015]. The FlashProg framework provides a general user interface for all these PBE systems, where users can use Program Navigation to navigate the space of learned programs in a hierarchical manner, and use Conversational Clarification to provide additional examples.

Jha et.al. [Jha et al., 2010] proposed *distinguishing inputs* for disambiguation in program synthesis - their synthesizer generates two consistent programs P_1 and P_2, and a distinguishing input on which P_1 and P_2 yield different results. The Conversational Clarification interaction model uses a similar idea to ask questions but it differs in several ways: (i) it selects distinguishing inputs from the user data instead of generating random inputs, (ii) it converges faster since it can execute all learned programs (instead of two) to ask for *more important* clarifications, and (iii) it works in real-time and is interactive unlike the constraint-solver based technique used in [Jha et al., 2010].

Topes [Scaffidi et al., 2008] allows developers to implement abstractions for interactively validating and transforming data in many different formats. It can recognize valid inputs in multiple different formats on a non-binary scale as opposed to binary-valued regular expressions. It provides transformation functions to convert inputs in different formats to a consistent format. The DSLs for FlashProg build on top of regular expressions and are quite different from the validation and transformation functions supported by Topes. Conversational Clarification uses the set of learnt programs to find ambiguous inputs unlike the non-binary valued matches used by Topes for finding questionable inputs.

Gamut [McDaniel and Myers, 1999] is a PBD system that enables non-programmers to create interactive games and educational software using demonstrations. Gamut's interaction techniques allows users to specify relationships between developer-generated

objects such as guide objects, cards, and decks of cards, and then use *nudges* and *hints* to modify or provide new behaviors. The "Do Something" interaction model lets users specify new behaviors on an object, whereas the "Stop That" interaction model lets users specify undesired behaviors. Similar to the "Stop That" model, FlashProg also lets users specify negative examples by clicking the labelled output in the input pane or marking the entry in the output table as incorrect.

`Import.Io` and `Kimono` are recent commercial tools that aim at extracting data from semi-structured sources. `Import.Io` performs extraction automatically without any human intervention. Although this works well for some simple semi-structured sources, it fails on relatively complex data sources. Adding support for handling newer semi-structured sources would require one to add new complex rules and heuristics. Kimono, on the other hand, performs data extraction by examples similar to FlashProg and provides a similar user interface. The range of logics for extracting sub-string data from html elements supported by Kimono, however, is not as rich compared to FlashProg. The learned regular expressions exposed by Kimono are too low-level to be easily understable by programmers, whereas FlashProg paraphrases the set of learned programs in a hierarchical manner. Moreover, Kimono does not support any conversational interaction model for disambiguating ambiguous cases.

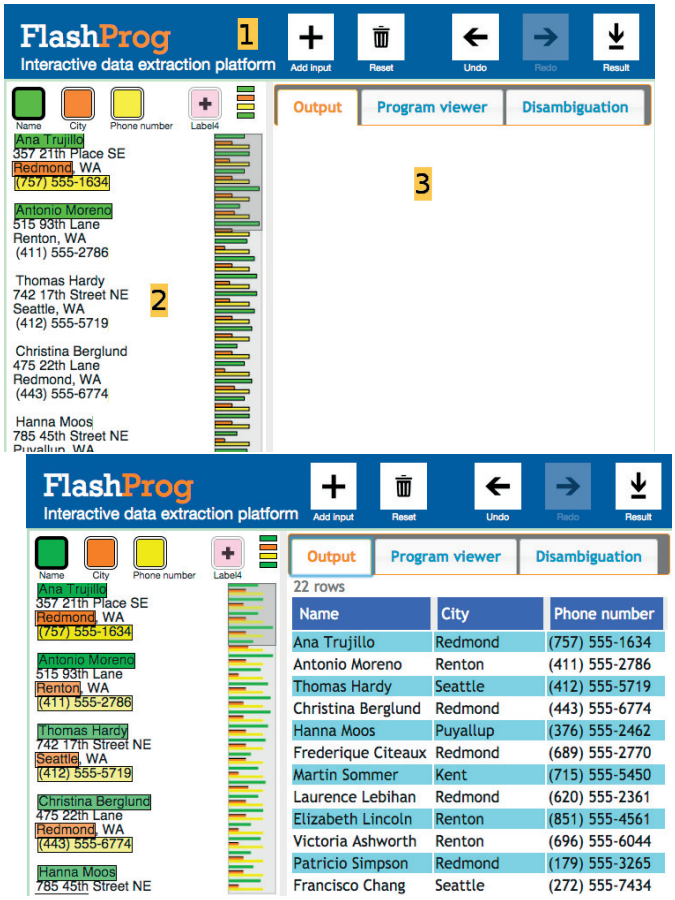


Figure 4.1 – FlashProg UI with PBE Interaction View in the "Output" mode, before and after the learning process. 1 – Top Toolbar, 2 – Input Text View, 3 – PBE Interaction View.

4.3 FlashProg User Interface

FlashProg is a web application for PBE-based data extraction from textual documents, spreadsheets, and Web pages. In this overview, we focus on the text domain, but the UI behaves similarly for all other domains as well. Figure 4.1 shows a FlashProg window after providing several examples (on the left), and after invoking the learning process (on the right). The FlashProg window consists of 3 sections: Top Toolbar (1), Input Text View (2), and PBE Interaction View (3).

The Top Toolbar contains: (a) an input button to open and upload files, (b) a button that resets FlashProg to an initial state, (c) undo/redo buttons as expected, and (d) a "Results" button to download the output as a CSV file for further processing.

The Input Text View is the main area. It gives users the ability to provide examples by highlighting desired sections of the document, producing a set of nested colored blocks. Additionally, users may omit the structure boundary and only provide examples for the fields as shown in Figure 4.1. After an automated learning phase, the output of the highest ranked program is displayed in the Output pane. Each new row in the output is also matched to the corresponding region of the original document that is highlighted with dimmer colors. The scroll bars are colored with a *bird's-eye view* of highlighting, as a minimap feature (as SublimeText.com). We have found this view helpful for looking for discrepancies in the produced highlighting. The user can also provide *negative examples* by clicking on previously marked regions to communicate to the PBE system that the region should not be selected as part of the output.

The PBE Interaction View is a tabbed pane giving users an opportunity to interact with the PBE system in three different ways: (i) exploring the produced output, (ii) exploring the learned program set paraphrased into English inside program viewer (Program Navigation), and (iii) engaging in an active learning session through the "Disambiguation" feature (Conversational Clarification).[1]

The Output pane displays the current state of data extraction result either as a relational table or as a tree. To facilitate exploration of the data, the Input Text View is scrolled to the source position of each cell when the user hovers over it. The user can also mark incorrect table rows as negative examples.

The Program viewer pane (Figure 4.2) lets users explore the learned programs. We concisely describe regexes that are used to match strings in the input text. For instance,

[1] Note that throughout the paper, we refer to the "disambiguation" as an overall problem of selecting the program that realizes user's intent in PBE. However, in our UI we use the word "Disambiguation" as a header of a pane with one iteration of the Conversational Clarification process. We found that it describes Conversational Clarification most lucidly to the users. Hereinafter in the paper, we refer to the "Disambiguation pane" in our UI if the context does not facilitate any confusion with the "disambiguation problem".

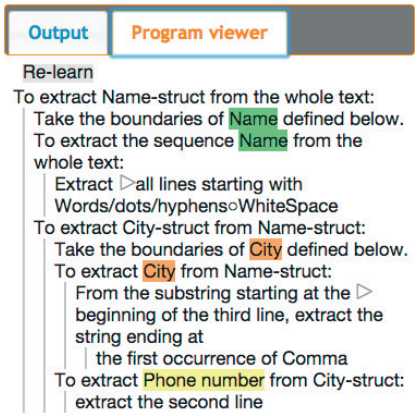


Figure 4.2 – Program Viewer tab of FlashProg. It shows the extraction programs that were learned in the session in Figure 4.1. The programs are paraphrased in English and indented.

"`Words/dots/hyphens.WhiteSpace`" (the circle is an infix concatenation) represents `[-.\pLu\pLl]+o\pZs+` (viewable in code mode). To facilitate understanding of these regexes, when the user hovers over part of a regex, our UI highlights matches of that part in the text. In Figure 4.2, `Name-Struct` refers to the region between two consecutive names; `City-Struct` refers to the region between `City` and the end of the enclosing `Name-Struct` region. Learned programs reference these regions to extract data. For instance, `Phone` is learnt relatively to enclosing `City-struct` region: "`second line`" refers to the line in the `City` region. In addition, clicking on the triangular marker opens a list of alternative suggestions for each subexpression. We show number of highlights that will be added (or changed/removed) by the alternative program as a +number (or a -number). If the program is incorrect, the user can replace some expressions with alternatives from the suggested list (Figure 4.6).

The Disambiguation pane (Figure 4.7) presents the Conversational Clarification interaction model. The PBE engine often learns multiple programs that are consistent with the examples but produce different outputs on the rest of the document. In such cases, this difference is highlighted and user is presented with an option to choose between the two behaviors. Choosing one of the options is always equivalent to providing one more example (either positive or negative), thereby invoking the learning again on the extended specification.

4.3.1 Illustrative Scenario

To illustrate the different interaction models and features of FlashProg, we consider the task of extracting the set of individual authors from the Bibliography section of a paper "A Formally-Verified C Static Analyzer" [Jourdan et al., 2015] (Figure 4.3). Our model user Alice wants to extract this data to figure out who is the most cited author in papers presented at POPL 2015.

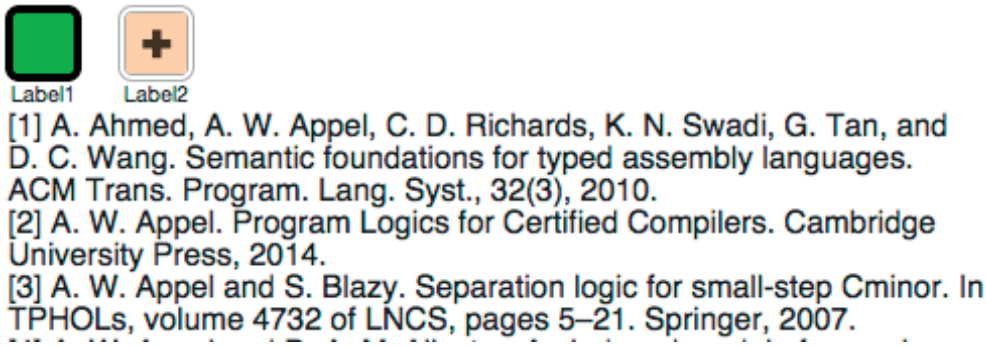


Figure 4.3 – Initial input to FlashProg in our illustrative scenario: extraction of the author list from the PDF bibliography of "A Formally-Verified C Static Analyzer" [Jourdan et al., 2015].

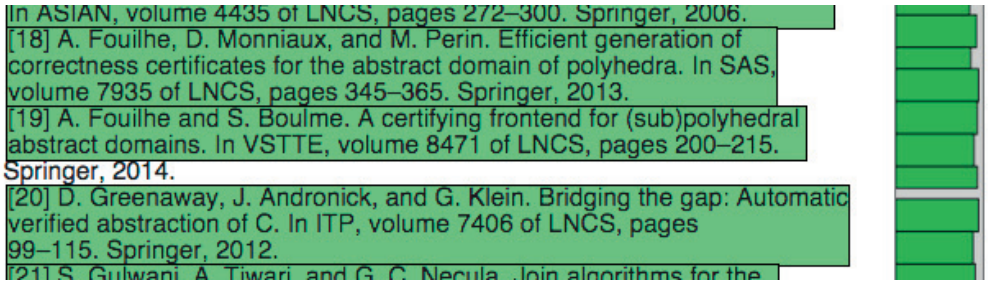


Figure 4.4 – Bird's eye view showing discrepancy in extraction.

First, Alice provides an example of an outer region containing each publication record. After providing two examples, a program is learned and other publications are highlighted, but the user notices that there is an unexpected gap between two extracted regions using the bird's-eye view (Figure 4.4). Giving another example to also include the text "`Springer, 2014.`" fixes the problem and a correct program is learned for the publication record regions.

Next, Alice wants to extract the list of authors and provides an example inside the first record. After learning, she observes that the program learned is behaving incorrectly (Figure 4.5). At this point, Alice can provide more examples as before to fix the problem, but it is easier to switch to the Program Viewer tab, and select a correct alternative for the wrong subexpression (Figure 4.6). The top-ranked program for extracting the Author list from a Record is "`extract the substring starting at first occurrence of end of whitespace and ending at the first occurrence of end of Camel Case in the second line`". The sub-program for the starting position seems correct but the sub-program for the ending position seems too specific for the given example, and Alice can ask for other alternative programs that the system has learned for the end position. Hovering over each alternative previews the extraction results in the input pane. In this case, Alice hovers over the first alternative, which generates the correct result. The final learned program turns out to be "`extract everything between first whitespace and first occurrence of Dot after CamelCase`" that is correct ("Wang" is considered to be in CamelCase by FlashProg, even though it is just one word),

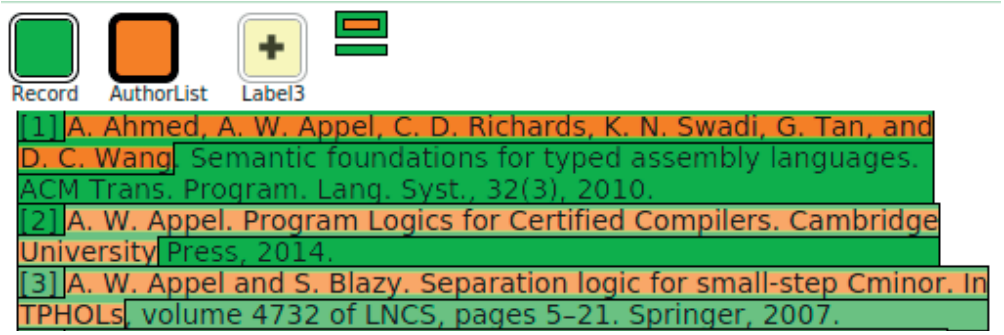


Figure 4.5 – An error during the author list extraction.

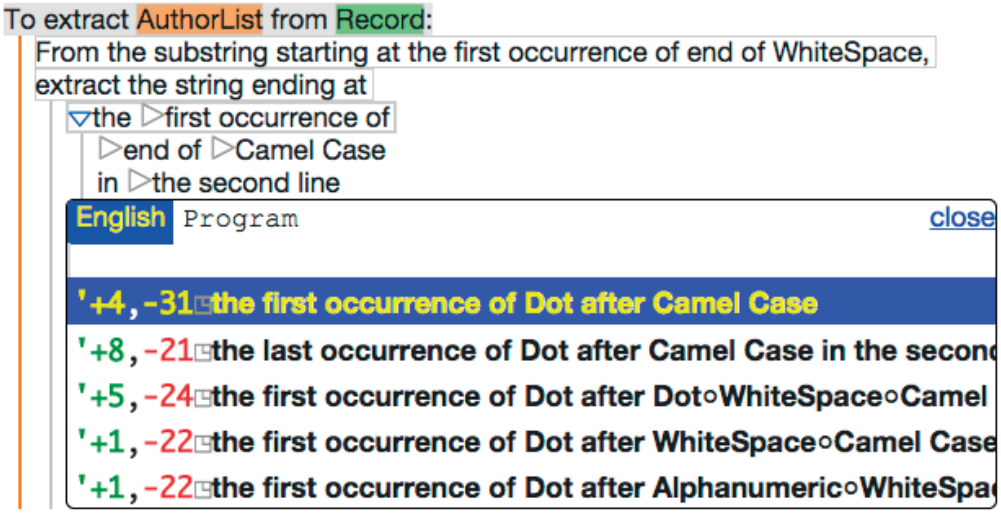


Figure 4.6 – Program Viewer tab & alternative subexpressions.

but the logic is quite non-obvious even for a programmer to come up with.

Now Alice wants to extract each author individually, and provides two examples within the first publication record. FlashProg again does not identify all authors correctly. Alice can provide additional examples or look at the extraction program, but she decides to engage the Conversational Clarification mode, and help FlashProg disambiguate between programs by answering clarifying questions (such as should the output include "`D. Richards`" or "`C. D. Richards`" and if "`and`" should be included, as shown in Figure 4.7). At each iteration, FlashProg asks her to choose between several possible highlightings in the unmarked portion of the document. Each choice is then communicated to the PBE system and the set of programs is re-learned. After two iterations of Conversational Clarification, FlashProg converges on a correct program, and Alice is confident in it (Figure 4.8).

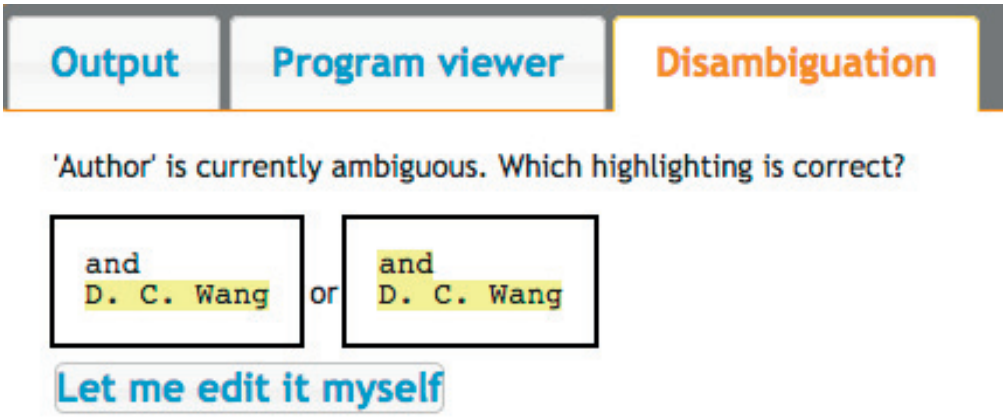


Figure 4.7 – Conversational Clarification being used to disambiguate different programs that extract individual authors.

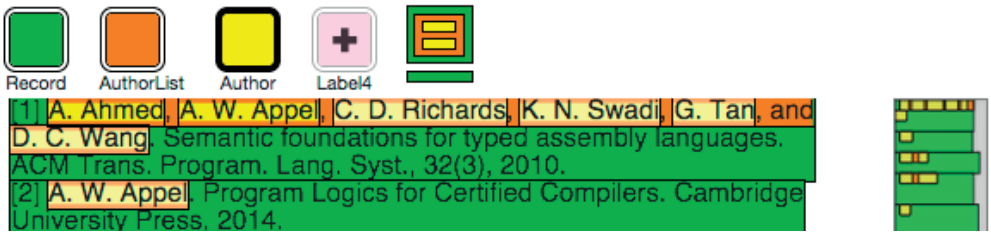


Figure 4.8 – Final result of the bibliography extraction scenario.

4.4 Implementation

Our underlying program learning engine is a rich toolkit of generic algorithms for PBE. It allows a domain expert to easily define a *domain-specific language* (DSL) of programs that perform data manipulation tasks in a given domain [Polozov and Gulwani, 2015]. The expert (DSL designer) only defines the semantics of DSL operators, from which our engine automatically generates a *synthesizer*. A synthesizer is an algorithm that, at run time, accepts a *specification* from a user, and returns a *set of DSL programs* that satisfy this specification. For instance, a specification in FlashExtract, the text processing DSL of FlashProg, is given by a sequence of positive and negative highlightings. The efficiency of our learning engine is based on two ideas from our prior work in PBE: our *synthesis algorithm* and our *program set representation*.

Synthesis algorithm Most prior work in PBE implement their synthesis algorithms by exhaustive search over the DSL, or delegate the task to constraint solvers [Alur et al., 2013]. In contrast, our engine employs an intelligent "top-down" search over the DSL structure, in which it iteratively transforms the examples given by an end user for the entire DSL program into the examples that should be satisfied by individual subexpressions in the program [Polozov and Gulwani, 2015]. Such an approach allows FlashProg to be responsive within 1-3 seconds for each learning round, whereas state-of-the-art PBE techniques can take minutes or even hours on similar tasks. Moreover, it also allows us to generate a *set* of programs satisfying a specification, instead of a single candidate. We then use a domain-specific *ranking scheme* to select a program that will

be presented to the user.

Program set representation A typical learning session can return up to 10^{30} ambiguous programs, all consistent with the current specification [Gulwani et al., 2012]. Our engine makes use of a polynomial-space representation of such a program set, known as *version space algebra* (VSA). It has been introduced by Mitchell [Mitchell, 1982] in the context of machine learning, and later used by Lau et al. [Lau et al., 2001], Polozov and Gulwani [Gulwani, 2011, Polozov and Gulwani, 2015].

The key idea of VSAs is sharing of subspaces. Consider an operator $\texttt{SubStr}(s, p_1, p_2)$, which extracts a substring of s that starts at the position p_1 and ends at the position p_2. Here p_1 and p_2 can expand to various position logics, e.g. absolute ("5^{th} character from the right") or based on regular expressions ("after the second number"). On a given example, p_1 and p_2 are known to evaluate to 1 and 4, respectively (i.e. the result of $\texttt{SubStr}(s, p_1, p_2)$ is the string $s[1:4]$). Importantly, both p_1 and p_2 may satisfy this specification in multiple possible ways. For example, p_1 can expand to a program "1^{st} character from the left", or a program "$(|s|-1)^{\text{th}}$ character from the right", or any consistent regex-based program (based on the content of s in a given example). Thus, the total number of possible consistent $\texttt{SubStr}(s, p_1, p_2)$ programs is quadratic in the number of possible consistent position programs (since any consistent p_1 can be combined with any consistent p_2).

A VSA stores these programs concisely as a *join structure* over the two program sets with learned consistent program sets for p_1 and p_2 (also represented as VSAs). Such a structure consists of the two learned program sets for p_1 and p_2 and a "join tag", which specifies that any combination of the programs sampled from these two sets is a valid combination of parameters for the `SubStr` operator. Therefore, the overall size of a VSA is typically logarithmic in the number of programs it semantically represents.

Formally, our learning engine represents program sets as a combination of shared program sets using two operators: *union* and *join*. A union of two VSAs $\tilde{N}_1$ and $\tilde{N}_2$ represents a set that is a union of two sets represented by $\tilde{N}_1$ and $\tilde{N}_2$. A join of two VSAs $\tilde{N}_1$ and $\tilde{N}_2$ represents a set that is a Cartesian product of two sets represented by $\tilde{N}_1$ and $\tilde{N}_2$. Such a representation has two major benefits: **(a)** it stores an exponential number of candidate programs using only polynomial space, and **(b)** it allows exploring the shared parts of the space of candidates, and quickly examine the alternative candidate subexpressions at any given program level.

The ideas explained above are the key to our novel Program Navigation and Conversational Clarification interaction models. We present their implementation below.

4.4.1 Program Navigation

The two key challenges in Program Navigation are: paraphrasing of the DSL programs in English, and providing alternative suggestions for program expressions.

Templating language

To enable paraphrasing, we implemented a high-level templating language, which maps *partial programs* into *partial English phrases*. Lau stated [Lau, 2009]:

"Users complained about arcane instructions such as "set the `CharWeight` *to* 1*" (make the text bold).* [...] *SMARTedit users also thought a higher-level description such as "delete all hyperlinks" would be more understandable than a series of lower level editing commands."*

Our template-based strategy for paraphrasing avoids arcane instructions by using "context-sensitive formatting rules", and avoids low-level instructions by using "idiomatic rules", solving Lau's two problems.

Paraphrasing is a conflictless bottom-up process. If possible, we use an idiom. We then remove context formatters from the template and apply them to their referenced child's template. Let us illustrate the development process with an example, a toy program named S_1:

$$\mathsf{PosPair}(\mathsf{Pos}(\mathsf{Line}(1), 1), \mathsf{Pos}(\mathsf{Line}(1), -1))$$

which evaluates to the string between the start and end of the second line. Line indexes start at 0, whereas char indexes start at 1. The relevant DSL portion is defined as a CFG:

$$S \coloneqq \mathsf{PosPair}(p, p) \qquad p \coloneqq \mathsf{Pos}(L, n)$$
$$L \coloneqq \mathsf{Line}(n) \qquad n \coloneqq \texttt{int}$$

We add three paraphrasing rules:

$$\begin{aligned} \mathsf{PosPair} &\rightarrow \text{“extract the string between \{:0\} and \{:1\}”} \\ \mathsf{Pos} &\rightarrow \text{“the char number \{:1\} of \{:0\}”} \\ \mathsf{Line} &\rightarrow \text{“line \{:0\}”} \end{aligned}$$

{:0} and {:1} refer to first and second arguments. Paraphrasing S_1 yields (parentheses added to see the paraphrase tree):

“extract the string between (the char number (1) of (line (1))) and (the char

number (-1) of (line (1)))”

To differentiate the two 1, we rewrite the last two rules above with a list of dot-prefixed formatters:

Pos → “the {:1.charNum} of {:0}”
Line → “{:0.lineNum}”

charNum (resp. lineNum) is a formatter mapping ints to a char ordinal (resp. line ordinal). Formatters are lists of ⟨regex, result⟩ pairs modifying the template of the targeted child. Its template is then replaced by the first matching regex result. For example, the formatter for charNum (and another formatter ordinal) is:

```
charNum: [ {regex: "^1$",        result: "beginning"}, ...
           {regex: "^(\\d+)$", result: "{:1.ordinal} char"}],
ordinal: [ {regex: "^1$",        result: "first"},
           {regex: "^2$",        result: "second"}... ]
```

Note how we handle corner cases. Paraphrasing S_1 yields

“extract the string between (the (beginning) of (second line)) and (the (end) of (second line)))”

The paraphrasing can be made even more concise by adding *idiom rules*, which produce more natural paraphrasing for certain idiomatic expressions. An idiom rule applies to subexpressions that satisfy given equality conditions between subterms or inner terms, specified by their paths. A *path* is a colon-separated list of symbols, function names and child indexes referring to a particular node. The rule below expresses the idiom of extracting the entire line:

PosPair(Pos(?L, 1), Pos(?L, −1)) when {:0:L} = {:1:L} → “extract the {:L}”

S_1 is finally paraphrased into “extract the (second line)”.

The limitations of this approach are mostly that all rules are written and updated manually. When the DSL changes, this is extra work. Furthermore, paraphrasing depends on order of formatters and idioms, and idiom templates may also not allow the user to explore the full program. We overcome this by letting the user switch between the paraphrase and the code (the latter being always complete).

Program alternatives

To enable alternatives, we record the original candidate program set for each subexpression in the chosen program. Since it is represented as a VSA, we can easily retrieve a subspace of alternatives for each program subexpression, and apply the domain-specific ranking scheme on them. The top 5 alternatives are then presented to the user.

4.4.2 Conversational Clarification

Conversational Clarification selects examples based on different outputs produced by generated programs. Each synthesis step produces a VSA of ambiguous programs that are consistent with the given examples. Conversational Clarification iteratively replaces the subexpressions of the top-ranked program with its top k alternatives from the VSA. This produces k clarification candidates (in FlashProg, k is set to 10). The clarifying question for the user is based on the *first discrepancy* between the outputs of the currently selected program P and the clarification candidateP'. Such a discrepancy can have three possible manifestations:

- The outputs of P and P' match until P selects a region r, which does not intersect any selection of P'. This leads to the question "Should r be highlighted or not?"
- The outputs of P and P' match until P' selects a region r', which does not intersect any selection of P. This leads to the question "Should r' have been highlighted?"
- The outputs of P and P' match until P selects a region r, P' selects a different region r', and r intersects r'. This leads to the question "Should r or r' be highlighted?"

For better usability (and faster convergence), we merge the three question types into one, and ask the user "What should be highlighted: r_1, r_2, or nothing?" Selecting r_1 or r_2 would mark the selected region as a positive example. Selecting "nothing" would mark both r_1 and r_2 as negative examples. After selecting an option, we convert the choice into one or more examples, and invoke a new synthesis process.

Analysis Since Conversational Clarification is an iterative refinement of a previous synthesis process, it is guaranteed to perform several times more efficiently compared to the last process. Moreover, since we pick a clarifying question based on different outputs produced by two ambiguous candidates, *the new set of candidates is guaranteed to be smaller than the previous one.* Therefore, Conversational Clarification converges to the program(s) representing user's intent in a finite number of rounds (if such programs exist). The number of rounds depends on the space of collisions in DSL outputs and can be exponential. In our user study and in most of our benchmarks however, the number of Conversational Clarification rounds never exceeded 5 for a single label.

A Conversational Clarification round is sound by construction (i.e. accepting a suggestion always yields a program that is consistent with both the suggestion and the prior examples). However, since our choice of clarification candidates is limited to top k alternatives at each level of the VSA, the Conversational Clarification round may be incomplete (i.e. the suggestions may not include the intended correct output). User can always provide a manual example instead of using CC suggestions in such a situation. The performance of a single Conversational Clarification round is linear in the VSA space (which is typically logarithmic in the number of ambiguous programs), since CC is implemented over our novel (recursively defined) *ranking* operation over the VSA [Polozov and Gulwani, 2015].

4.4.3 Domain-specific languages

The generic implementation of our learning engine allows rapid development of DSLs for various data manipulation domains without the accompanying burden of designing individual synthesis algorithms or other FlashProg functionality for them. Following this methodology, we easily incorporated the following data manipulation DSLs in FlashProg:

1. FlashFill – a DSL for syntactic string transformations [Gulwani, 2011].

2. FlashExtract – a DSL for extracting textual information from semi-structured documents [Le and Gulwani, 2014].

3. FlashRelate – a DSL for extracting relational tables from semi-structured spreadsheets [Barowy et al., 2015].

4. FlashWeb – a DSL for extracting webpage content based on CSS selectors.

We design these DSLs such that they are succinct enough to enable efficient learning, yet expressive enough to support many real-world tasks. If a task can be expressed in our language, our engine will learn a program for it given sufficiently many examples. The engine fails if the language cannot express the task. For example, FlashExtract does not support arbitrary boolean conjunctions and disjunctions. Hence, if the tasks require learning a complex boolean expression, FlashExtract will not be able to perform it [Gulwani, 2011, Le and Gulwani, 2014, Barowy et al., 2015].

Next, we present our user study on FlashExtract below, but the functionality of FlashProg is automatically provided for any compliant DSL. We plan to incorporate more extraction domains, such as PDF documents, in future work.

4.5 Evaluation

In this section, we present a user study to evaluate FlashProg. In particular, we address three research questions for PBE:

- RQ1: Do Program Navigation and Conversational Clarification contribute to correctness?
- RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction?
- RQ3: Do FlashProg's novel interaction models help alleviate typical distrust in PBE systems?

4.5.1 User study design

Because our tasks can be solved without any programming skills, we performed a within-subject study over an heterogeneous population of 29 people: 4 women aged between 19 and 24 and 25 men aged between 19 and 34. Their programming experience ranged from none (a 32-year man doing extraction tasks several times a month), less than 5 years (8 people), less than 10 (9), less than 15 (8) to less than 20 (3). They reported performing data extraction tasks never (4 people), several times a year (7), several times a month (11), several times a week (3) up to every day (2).

We selected 3 files containing several ambiguities these users have to find out and to resolve. We chose these files among anonymized files provided by our customers. Our choice was also motivated by repetitive tasks, where extraction programs are meant to be reused on other similar files. The three files are the following:

1. **Bank listing.** List of bank addresses and capital grouped by state. The postal code can be ambiguous.
2. **Amazon research.** The text of the search results on Amazon for the query "chair". The data is visually structured as a list of records, but contains spacing and noise.
3. **Bioinformatic log.** A log of numerical values obtained from five experiments, from bioinformatics research (Figure 4.10). Straightforward extraction misses one experiment.

We first provided users a brief video tutorial using the address file as example (Figure 4.1, youtu.be/JFRI4wIR0LE). The video shows how to perform two extractions and to use features such as undo/redo. It partially covers the Program Viewer tab and the Disambiguation tab. It explains that these features will or will not be available, depending

PDB	First	Second	Third	CN
3DD1:A:905	85.8140	92.2910	123.2000	C
	85.7630	93.6200	122.5320	C
	86.8320	94.4810	122.6360	C
3DD1:A:903	93.1740	-54.6260	125.7390	N
	93.7660	-55.4170	126.7610	C
	92.9200	-53.1980	125.9660	C

Figure 4.9 – Bioinformatic log: Result sample.

on the tasks. When users start FlashProg, they are given the same file as in the video. A pop-up encourages them to play with it, and to continue when they feel ready. The Program Viewer tab and the Disambiguation tab are both available at this point.

We then ask users to perform extraction on the three files. For each extraction task, we provide a result sample (Figure 4.9). Users then manipulate FlashProg to generate the entire output table corresponding to that task. We further instruct them that the order of labels do not matter, but they have to rename them to match our result sample.

To answer RQ1, we select a number of representative values across all fields for each task, and we automatically measure how many of them were incorrectly highlighted. These values were selected by running FlashProg sessions in advance ourselves and observing insightful checkpoints that require attention. In total, we selected 6 values for task #1, 13 for task #2 and 12 for task #3. We do not notify users about their errors. This metric has more meaning than if we recorded all errors. As an illustration, a raw error measurement in the third task for a user forgetting about the third main record would yield more than 140 errors. Our approach returns 2 errors, one for the missing record, and one for another ambiguity that needed to be checked but could not. This makes error measurement comparable across tasks.

Environments To measure the impact of Program Navigation and Conversational Clarification interaction models independently, we set up three interface environments.

Basic Interface (BI). This environment enables only the Colored Data Highlighting interaction model. It includes the following UI features: the labeling interface for mouse-triggered highlighting, the label menu to rename labels, to switch between them and the Output tab.

BI + Program Navigation (BI + PN). Besides the Colored Data Highlighting, this interface enables the Program Navigation interaction model, which includes the Program Viewer tab and its features (e.g. Regular expression highlighting, Alternative

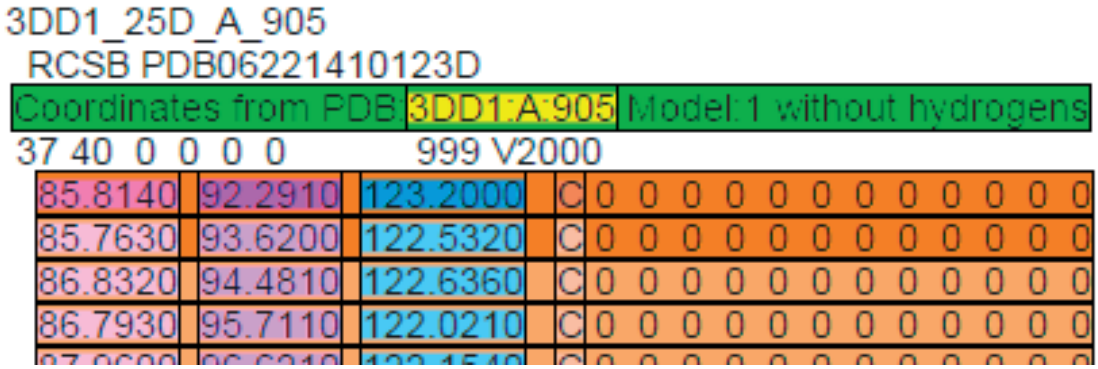


Figure 4.10 – Highlighting for obtaining Figure 4.9.

subexpression viewer).

BI + Conversational Clarification (BI + CC). Besides the Colored Data Highlighting, this environment enables the Conversational Clarification interaction model, which includes the Disambiguation tab.

To emphasize PN and CC, the system automatically opens the matching tab, if they are part of the environment.

Configurations To compensate the learning curve effects when comparing the usefulness of various interaction models, we set up the environments in three configurations A, B, and C. Each configuration has the same order of files/tasks, but we chose three environment permutations. As we could not force people to finish the study, the number of users per environment is not perfectly balanced.

	Tasks			
Config.	1. Bank	2. Amazon	3. Bio log	# of users
A	BI + PN	BI + CC	BI	8
B	BI	BI + PN	BI + CC	12
C	BI + CC	BI	BI + PN	9

Survey To answer RQ2 and RQ3, we asked the participants about the perceived usefulness of our novel interaction models, and the confidence about the extraction of each file, using a Likert scale from 1 to 7, 1 being the least useful/confident.

4.5.2 Results

We analyzed the data both from the logs collected by the UI instrumentation, and from the initial and final surveys. If a feature was activated, we counted the user for statistics even if he reported not using it.

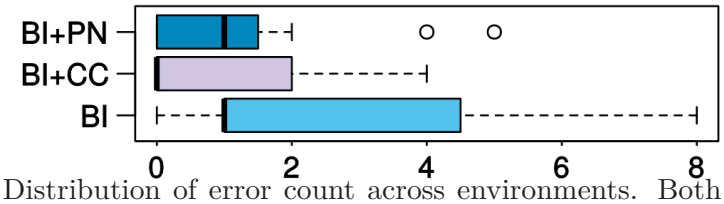


Figure 4.11 – Distribution of error count across environments. Both Conversational Clarification (CC) and Program Navigation (PN) significantly decrease the number of errors.

RQ1: Do Program Navigation and Conversational Clarification contribute to correctness? **Yes. We have found significant reduction of number of errors with each of these new interaction models** (See Figure 4.11). Our new interaction models reduce the error rate in data extraction without any negative effect on the users' extraction speed. To obtain this result, we applied the Wilcoxon rank-sum test on the instrumentation data. More precisely, users in BI + CC ($W = 78.5$, $p = 0.01$) and BI + PN ($W = 99.5$, $p = 0.06$) performed better than BI, with no significant difference between the two of them ($W = 94$, *n.s.*). There was also no statistically significant difference between the completion time in BI and completion time in BI + CC ($W = 178.5$, *n.s.*) or BI + PN ($W = 173$, *n.s.*).

RQ2: Which of Program Navigation and Conversational Clarification is perceived more useful for data extraction? **Conversational Clarification is perceived more useful than Program Navigation** (see Figure 4.12a and Figure 4.12b). Comparing the user-reported usefulness between the Conversational Clarification and the Program Navigation, on a scale from 1 (not useful at all) to 7 (extremely useful), the Conversational Clarification has a mean score of 5.4 ($\sigma = 1.50$) whereas the Program Navigation has 4.2 ($\sigma = 2.12$) Only 4 users out of 29 score Program Navigation more useful than Conversational Clarification, whereas Conversational Clarification is scored more useful by 15 users.

RQ3: Do FlashProg's novel interaction models help alleviate typical distrust in PBE systems? **Yes for Conversational Clarification.** Considering the confidence in the final result of each task, tasks finished with Conversational Clarification obtained a higher confidence score compared to those without ($W = 181.5$, $p = 0.07$). No significant difference was found for Program Navigation ($W = 152.5$, *n.s.*). Regarding the trust our users would have if they had to run the learned program on other inputs, we did not find any significant differences for Conversational Clarification ($W = 146$, *n.s.*) and Program Navigation ($W = 161$, *n.s.*) over only BI.

Regarding the question "How often would you use FlashProg, compared to other extraction tools?", on a Likert scale from 1 (never) to 5 (always), 4 users answered 5, 17 answered 4, 3 answered 3, and the remaining 4 answered 2 or less. Furthermore, all would recommend FlashProg to others. When asked how excited would they be to have such a tool on a scale from 1 to 5, 8 users answered 5, and 15 answered 4.

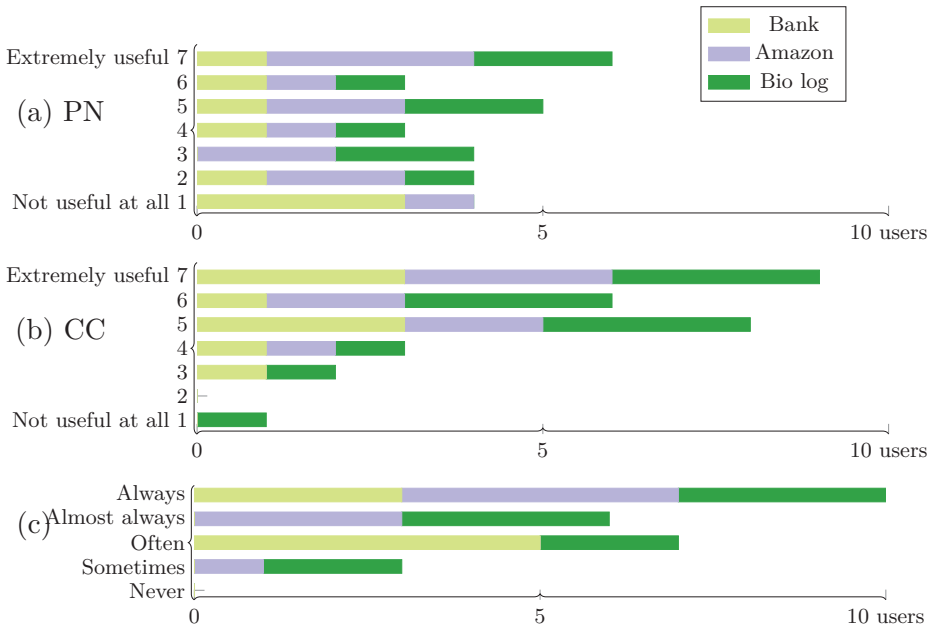


Figure 4.12 – User-reported: **(a)** usefulness of PN, **(b)** usefulness of CC, **(c)** correctness of one of the choices of CC.

The users' trust is supported by data: Perceived correctness is negatively correlated with number of errors (Spearman $\rho = -0.25$, $p = 0.07$). However, there is no significant correlation between number of errors made and the programming experience mapped between 0 and 4 ($\rho = -0.09$, *n.s.*).

Other results. We observed that only 13 (45%) of our users used the Program Viewer tab when it was available. These 13 users having experienced Program Navigation got mixed feelings about it. A 22-year woman with more than 5 years of programming experience gave a positive review: "I absolutely loved [regular expression highlighting]. I think that perfectly helps one understand what the computer is thinking at the moment and to identify things that were misunderstood". According to a 27-year man with more than 10 years of programming experience, the interaction was not easy enough: "the program [is] quite understandable but it was not clear how to modify the program". 9 users out of 13 did not report using the Alternative subexpression viewer when using the Program Navigation.

On the other hand, 27 (93%) used the Disambiguation tab when it was available. Users appreciated it. The previous woman said: "in the last example, in which I didn't have [Conversational Clarification] as an option, I felt like I miss it so much". A 27-year man with 5+ years of programming experience said: "It always helps me to find the right matching". A 19-year old novice programmer woman said: "The purpose of each box wasn't clear enough, but after the text on left became highlighted (hovering the boxes), the task became easier". Although there were tooltips, some users were initially confused about how we presented negative choices with XXX crossing the answer.

4.5.3 Discussion

With so many experienced users, we did not expect that only half of them would interact with Program Navigation, and even less with the Alternative subexpression viewer. To ensure usability, we developed FlashProg and Program Navigation iteratively based on the feedback of many demo sessions and a small 3-user pilot study before running the full user study. We did not receive any specific complaints about the paraphrasing itself, although it certainly required substantial time to understand their semantics. In the tasked they solved, users might then have thought that it would take more time to figure out where the program failed, and to find a correct alternative, than to add one more example. We believe that in other more complex scenarios, such as with larger files or multiple files, the time spent using Program Navigation could be perceived as more valuable and measured as such. The decrease of errors may then be explained by the fact that when Program Navigation was turned on, users have stared at FlashProg more and took more time to catch errors.

The negative correlation between the confidence of users in the result and the number of errors is insightful. Although we asked them to make sure the extraction is correct and never told them they did errors, users making more errors (thus unseen) reported to be less sure about the extraction. The problem is therefore not just about alleviating the users' typical distrust in the result, it is really about its correctness.

We also acknowledge that several factors may be a limitation of this study: (a) we have a limited amount of heterogeneous users; (b) the time was uncontrolled, thus we could not prevent users from getting tired or from pausing in the middle of extraction tasks; (c) besides the 29 users having completed all the study, more than 50 users who decided to start the study stopped before finishing the last task (this explains the unbalanced number of users for each condition). Thus, they were not part of the qualitative correlations (e.g. between confidence and errors), but we did include each finished task for the error metrics; (d) if a user extracts all regions manually, replacing a record not covered by the checkpoints by another, we do not measure this error (false negatives).

4.6 Conclusion and further work

We have implemented FlashProg, a prototype PBE system for data extraction and manipulation that supports two novel user interaction models for disambiguation in PBE, namely Program Navigation and Conversational Clarification. In user studies, where users were given three data extraction tasks, we found that a significant majority of users found the tool effective and were confident in the results. This confidence is supported by data: both models significantly reduced the number of extraction errors. Further, the users found the proactive behavior of Conversational Clarification very useful, and preferred it to the Program Navigation interface.

As PBE technologies such as FlashProg are made more widely available in the marketplace, we will better understand the interplay between the user's task understanding and the tool's ability to support them. Beyond our current work, there are many opportunities for improvements, including helping users with greater automation, helping them deal with incomplete and sometimes incorrect data, and identifying when the user has made a mistake in their examples.

4.7 Acknowledgments

This work was sponsored by Microsoft Research. Mikael, Gustavo, Vu, and Alex were supported by a one-year position, and Maxim was supported by a six-month position at MSR.

Page 80 until this page 105 contained the content of the paper "User Interaction Models for Disambiguation in Programming by Example". We will now add more insights.

B. Epilogue to FlashProg

B.1 Generalizing Conversational Clarification

After the experience of the last paper, and before the end of my internship at Microsoft, I tried to generalize these interaction models to other languages built on top of FlashMeta [Polozov and Gulwani, 2015]. I was able to implement programming-by-example extraction engines for FlashRelate, a language that is learnable by example and that describes relational transformations of tables [Barowy et al., 2015], and 'FlashWeb', a language without any publication attached to it for extracting data from web pages, but still available on Microsoft Prose's Playground[2]. I successfully reused the paraphrasing engines for both languages, and I also implemented conversational clarification for the extraction of web pages.

However, I could not figure out how to support a conversational clarification feature for FlashRelate. First, the engine of FlashRelate did not support the negative examples FlashProg needed when the user says that the extraction is wrong. Second, because the examples consisted in a row of related cells, the ambiguous input/output pairs provided by the machine were partially correct, not completely wrong. However, we could not figure a consistent way of asking for this information from the user without disclosing the language behind. We did not want to disclose the language of FlashRelate because it was meant to evolve. Besides, we wanted to keep conversational clarification from programming notions.

I also wanted to generalize the notion of disambiguation to FlashMeta itself. Disambiguation would be then configurable for any new language. However, because of the short amount of time remaining, I was not able to do so. Two years later, the team at Microsoft continued the effort and prepared a paper formalizing many notions related to disambiguation [Le et al., 2017]. In May 2017, two years after I left Microsoft, contrary to many abandoned research projects, this work was still more alive than ever. I received the following comment from the Head of Products of Microsoft Prose:

> "You were the one who, together with Gustavo [Soares], laid the foundation of the FlashProg demo site that we continue to use even today to find opportunities for our team! Once again, thanks a lot for all your contributions."

B.2 Discussion

We now answer the questions we asked in the introduction (see page 9) with respect to the work of this chapter.

[2] microsoft.github.io/prose

Question	FlashProg
Is it faster/easier to conduct tasks using programming-by-example?	Yes
Do users need to see the generated program?	Yes
Do users need to take on the generated program?	No
Is the paraphrased version of the program useful?	Yes
Is it more reliable when the program is shown?	**Yes**
Is it more reliable when the computer asks questions?	**Yes**

The comparative summary for all the papers is available on page 163.

Arguably, the cognitive model of text extraction is very close to what users might want, and it takes much less time to extract tables from the text, in comparison with designing and writing suitable regular expressions. Using the feedback of the tool ConverFrom-String, we acknowledge that users asked to see the program. However, users prefer to stay in the extraction interface, rather than trying to tweak the program, as suggested by the user study. Thanks to users' comments and results, we noted that the paraphrased version of the program is useful and provides reliability to this kind of programming-by-example tasks. Finally, the questioning was a feature that helped users gain trust of this kind of programming-by-example.

5 Complete Interactive Questioning

"Celui qui aime à demander conseil grandira."
Proverbe chinois

"He who loves to ask for advice will grow."
Chinese proverb

A. Prologue: Attempts to Reproduce Interaction Models

Back at EPFL, I wondered if I could replicate the exciting results of the previous chapter in some other domains, specifically how new interaction models could possibly bring a new life to programming-by-example and lower the barrier between abstract wishes and specifications. I began investigating database-management systems[1], website engines, and looking for opportunities for computers to actively communicate with users to create software. With a few thousand questions, I dreamed that we would be able to create something as complex as a car-pooling website. I started to look for benchmarks that serve as a basepoint for such systems. Quickly, it became extremely difficult to figure out the line between programming-by-example and basic programming. In complicated websites, it seemed that giving examples of how the website should work would take much more time than just coding it.

After three months of investigation, I had a precise, simple, yet encouraging idea of what kind of problem related to the programming-by-example that I wanted to solve: Build a static webpage produced by a programming language, enabling dual modifications in either the webpage or the programming language. For this, I started to encode a simple specialized solver for strings, where one side of the equation is a constant. At the completion of my thesis, I have not fully completed the original dream of dual programming, because I came across another challenge that raised my interest and was related to my previous work.

[1] https://github.com/OlivierBlanvillain/SlickChair-report/blob/master/report/main.pdf

Back from a conference, Jad Hamza and Viktor Kuncak had a new challenge for me. When asked to prove conjectures, the prover of the lab was returning counter-examples on the form of trees that were so big that they were unintelligible. Here is one of them:

```
f(Cons[Event[StackTid, StackMethodName, Option[BigInt], Option[BigInt]]](CallEvent[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]](T1(), Push(), Some[BigInt](5)), Cons[
  Event[StackTid, StackMethodName, Option[BigInt], Option[BigInt]]](InternalEvent[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]](T1()), Cons[Event[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]]](CallEvent[StackTid, StackMethodName,
  Option[BigInt], Option[BigInt]](T2(), Pop(), None[BigInt]()), Cons[Event[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]]](RetEvent[StackTid, StackMethodName,
  Option[BigInt], Option[BigInt]](T1(), Push(), Some[BigInt](5), None[BigInt]()), Cons[Event[
  StackTid, StackMethodName, Option[BigInt], Option[BigInt]]](InternalEvent[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]](T2()), Cons[Event[StackTid,
  StackMethodName, Option[BigInt], Option[BigInt]]](RetEvent[StackTid, StackMethodName,
  Option[BigInt], Option[BigInt]](T2(), Pop(), None[BigInt](), Some[BigInt](5)), Nil[Event[
  StackTid, StackMethodName, Option[BigInt], Option[BigInt]]]()))))))
```

Even after pruning the generic types, it is still not convenient to read:

```
Cons(CallEvent(T1(), Push(), Some(5)), Cons(InternalEvent(T1()), Cons(CallEvent(T2(), Pop(),
  None()), Cons(RetEvent(T1(), Push(), Some(5), None()), Cons(InternalEvent(T2()), Cons(
  RetEvent(T2(), Pop(), None(), Some(5)), Nil()))))))
```

Viktor Kuncak and Jad Hamza wanted a way to convert these trees to strings in order to alleviate the task of understanding them. Even better, if they had been able to provide *examples* on-the-fly of how to print, that would have been great. For example, Jad Hamza told me he wanted to print the above tree as follows:

```
T1: call  Push(5)
T1: internal
T2: call  Pop()
T1: ret  Push(5)
T2: internal
T2: ret  Pop() → 5
```

There were several challenges. For example, the system should print Some and None differently, depending on the context.

I started to work on Jad Hamza's idea. After a few months, I had a prototype working, built on top of my previous specialized solver on strings. My first solution was to build function templates and guess the missing string constants using examples. I obtained the templates of mutually recursive functions directly from the definitions of datatypes. After describing the way it should build the above counter-example, the system would ask the following question:

```
How (...) should be rendered?
T1: ret  Pop(1) → 2T1: ret Pop(3) → 4
```

```
Or T1: ret Pop(1) → 2
T1: ret Pop(3) → 4
```

to which a user could answer the second one, apparently eliminating the remaining ambiguities. *Conversational Clarification* (page 83) was working again.

In order to print datatypes differently, depending on the context – as for the previous Some datatype – I started to enable the automatic unrolling of methods, so that the engine would have more degrees of freedom to find programs. Eventually, I made this approach complete by proving that it could theoretically reach any grammar-defined pretty printer, by enumerating grammars recursively one by one. This was not very efficient, but it worked.

I also wanted, however, that the engine asks questions to find the constants, instead of initially forcing the user to provide all the necessary examples. As the outputs are composed of a concatenation of the constants of the program and the strings of the input, my first guess was to find inputs covering the entire program.

Therefore, I made a coverage engine to compute a set of inputs that cover the whole program. For this, I modified the program to return pairs, such that the second element would be true if and only if the program reached a particular line. Using the verification engine, by asking it to prove that the second element of the pair is always false, I would obtain one by one counter-examples that precisely covered the mentioned line.

Quickly, I discovered that coverage was not enough. Even if executing the program on the inputs would reach all the constants of the programs, I found examples that were ambiguous. For example, I could miss the order in which trees ought to be printed.

```
def print1(l: List) = l match {
  case Cons(a: String, b: List) => a + print1(b)
  case Nil() => ""
}
def print2(l: List) = l match {
  case Cons(a: String, b: List) => print2(b) + a
  case Nil() => ""
}
print1(Cons("1", Nil)) = "1"
print2(Cons("1", Nil)) = "1"
```

In this case, Cons(1, Nil) obviously covers all program lines, but the two functions print the lists in a different order.

To circumvent this problem, I tried to obtain a more general notion of coverage. I originally defined the notion of a "rec-covering" input set, which is a covering set such that for all concatenation in the program $\sum_i M_i$, by permuting the concatenation yields

either the same program, or changes the output of one of the elements of the set.

Then I needed to check whether any two programs are the same. As we will see in the next chapter, this is not an easy task.

Even though I had an incomplete coverage theory, I was still able to generate questions because I would run the rec-coverage engine on many programs that I found consistent with the provided input/output examples. Whenever I would have two different outputs, the engine created the question for the user. After she answered this question by choosing an output, the engine would add a new constraining input/chosen-output pair to the problem.

We realized that we were going in the wrong direction, because there were infinitely many different programs that the system could generate based on successive unfoldings of the grammar and that were eventually consistent with the set of input/output examples. I remember the system asking me the following question, when printing lists:

> Should we print Cons(1, Cons(2, Cons(3, Nil))) either
> [1, 23] or [1, 2, 3] ?

and right after I answered by the second, it would ask:

> Should we print Cons(1, Cons(2, Cons(3, Cons(4, Nil)))) either
> [1, 2, 34] or [1, 2, 3, 4] ?

This could continue forever. Thus arose the not-so-obvious question:

When do we stop questioning the user?
"How can any system successfully guess the user's intended program out of an infinite space of possible programs?" [Lau, 2009]

Even at the time of writing this thesis, I do not have an answer for this question. However, in the next chapter, we will study an apparently smaller problem that consists of learning tree-to-string functions without state and without unfoldable grammars. Nonetheless, we still obtained the not-so-obvious result that questioning is so important that it reduces the complexity of learning printers.

B. The Patented Sandwich Metaphor

In the area of program verification, proving conjectures is a task that requires understanding the counter-examples that any automated prover can provide. By analyzing counter-examples, we can refine the conjecture and transform it into a theorem. With

a programming-by-example technique to teach the computer how it should display its counter-examples, to make sure it understood correctly, we might be tempted to enable the computer to ask more questions. However, the computer might begin to ask an infinite number of questions. Hence, knowing when to stop questioning is a major element in the building of programs (helpers) that correctly render counter-examples.

In order to not fall asleep while I was driving on a long trip, my wife kindly asked me once again to explain what I was currently working on. As the result of an effort to get the best metaphor, I invented the educative stories on patented sandwiches which I present on the next page.

Once upon a time, in a country where only one type of coin was in circulation, someone started to sell sandwich machines. These sandwich machines became so popular that most merchants began to use them. Their use was the following. Merchants provided the machine great quantities of middle toppings (M), top toppings (T) and bottom toppings (B), each having the form of a square. When a user spent one coin in the machine, the machine would provide a middle topping M. By inserting one more coin, the machine would append a bottom topping B below the sandwich in construction, and a top topping T above the sandwich in construction. Repeating this operation by inserting more coins, the machine would continue to add for each coin one topping B below and one topping T above. For example, one merchant assigned a slice of white bread to M, brown bread above ham to T and cereal bread below salad to B (see Figure 5.1). The more coins a user put, the more ham/brown bread and cereal bread/salad the machine adds to their sandwich.

Then there came the problem of patents. Merchants could patent their machine after they configured the toppings. But then, they started to sue each other for creating machines that produced the same kind of sandwiches.

B.1 The First Case: Chocolate Brioches

A merchant brought another merchant to the court, because their machines were both producing a "sandwich" consisting of a slice of brioche below a slice of chocolate, repeated as many times as users gave coins.

The second merchant had learned that the first merchant assigned brioche/chocolate to M, brioche/chocolate to T, and nothing to the bottom. Therefore, he decided to configure his machine slightly different by producing the same sandwiches, but exchanging T and B:

Machine 1

Top:	Chocolate Brioche
Middle:	Chocolate Brioche
Bottom:	

Machine 2

Top:	
Middle:	Chocolate Brioche
Bottom:	Chocolate Brioche

However, obviously, the obtained sandwiches would be the same. The judge decided that this was a violation of the patent. No matter the internal details, machines owned by different merchants should not produce the same sandwiches.

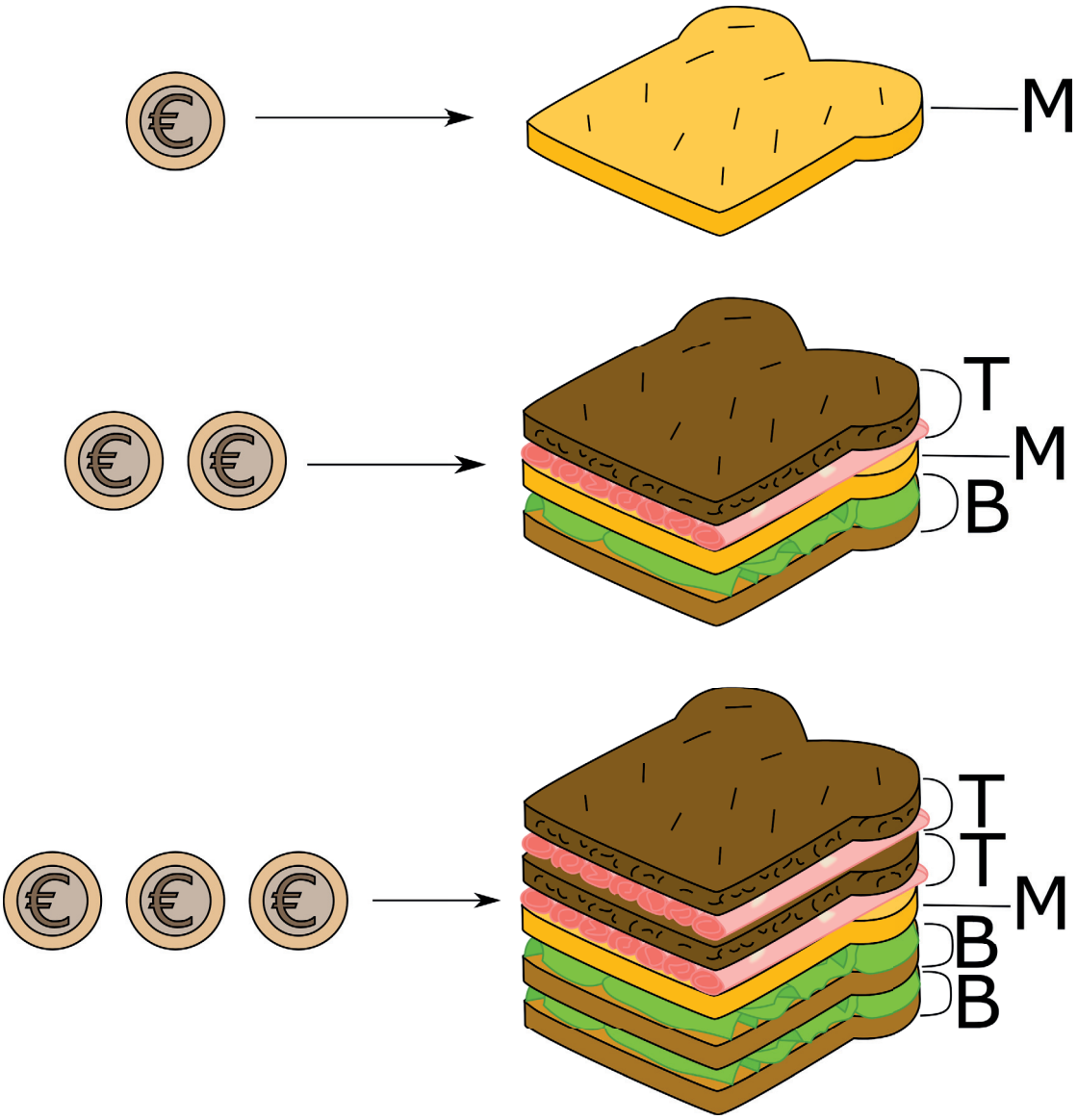


Figure 5.1 – Outputs produced for respectively 1, 2 and 3 coins from a machine configured for M to be a slice of white bread, T to be brown bread over ham and B to be salad over cereal bread. In this setting, for n coins, the sandwich composition will always be from bottom to top $B^{n-1}MT^{n-1}$.

B.2 The Second Chocolate-Brioche Case

Then a curious case was brought before the judge. The first merchant tested the second merchant's machine with one, and then two coins, and saw that it produced the same sandwiches as his own:

Machine 1

One coin:	Chocolate Brioche
Two coins:	Brioche Chocolate Brioche Chocolate Brioche

Machine 2

One coin:	Chocolate Brioche
Two coins:	Brioche Chocolate Brioche Chocolate Brioche

Therefore, he complained to the judge that the sandwiches were always the same. Were the machines equivalent? The judge asked to look inside the machines, saw that with three coins they would produce different sandwiches. Usually, I allow people time to think about what could have happened. Nevertheless, here is the solution:

Machine 1

Top:	Brioche
Middle:	Chocolate Brioche
Bottom:	Chocolate Brioche

Machine 2

Top:	Brioche Chocolate Brioche
Middle:	Chocolate Brioche
Bottom:	

Testing with three coins thus revealed different sandwiches, since the sandwich of the first machine would have two brioches on top, and not the second one, which will always have brioche/chocolate on top.

Therefore, the judge asked all merchants to test equality for not only one and two coins, but also for three coins.

B.3 Are Three Coins Enough?

But what if next time the merchants test their sandwich machines with up to 3 coins, will it still be enough to conclude that the machines are always equivalent ? Fortunately, in appendix D page 205, I developed a computer-verified proof that it is sufficient.

B.4 One More Type of Coin

After the growth of the country, the original coin E was not enough; the machines were replaced by new machines accepting the coin E and a new coin type F. With two coins E and F, the merchants had to provide two kinds of Top, Middle and Bottom toppings, one for each coin. This provided more freedom for users on how to compose sandwiches.

B.5 The Third Case: Bread/Ham

Having one more type of coin gave more freedom to the merchants, but another case appeared in the court. The first merchant had tested the second merchant's machine with E, EE and EEE, F, FF, FFF and it gave the same sandwiches as his own machine. Here are the results of the test:

Machine 1

E:	
EE:	Bread
EEE:	Bread
	Bread
F:	
FF:	Ham
FFF:	Ham
	Ham

Machine 2

E:	
EE:	Bread
EEE:	Bread
	Bread
F:	
FF:	Ham
FFF:	Ham
	Ham

Can two such machines be different?

Opening the machines, the judge saw their difference:

Machine 1

Coin E	Top:	Bread
	Middle:	
	Bottom:	
Coin F	Top:	Ham
	Middle:	
	Bottom:	

Machine 2

Coin E	Top:	
	Middle:	
	Bottom:	Bread
Coin F	Top:	Ham
	Middle:	
	Bottom:	

The judge then tested the two machines with coins EFE in this order, and obtained two different sandwiches:

Machine 1

EFE:	Bread
	Ham

Machine 2

EFE:	Ham
	Bread

Hence, there was no violation of patent. The judge decided that all machines had to be

tested with all configurations of up to three coins before being brought to him.

B.6 Advanced Bread/Ham Case

Later, a merchant found that another merchant's machine was giving the same sandwiches as his own machine for any of up to three coin combinations: E, F, EE, EF, FE, FF, EEE, EEF, EFE, EFF, FEE, FEF, FFE and FFF. He subsequently concluded that the machines were equivalent. Having a small doubt, the judge decided to look into the internals of both machines to check for equivalence.

Machine 1

Coin E	Top:	Bread
	Middle:	
	Bottom:	Ham

Coin F	Top:	Bread Ham
	Middle:	
	Bottom:	

Machine 2

Coin E	Top:	Bread
	Middle:	
	Bottom:	Ham

Coin F	Top:	
	Middle:	
	Bottom:	Bread Ham

The judge thus realized that the two sandwiches given by both machines for the coin order FEFE would not be the same, so there was no patent infringement.

B.7 General Case

Should we try all combinations of four coins? What should we do if we have more types of coins?

Due to the results in the remainder of this chapter, we know that checking up to four coins is sufficient to prove or disprove the equality of both machines, whatever the number of types of coins there is. We even have a more general result applicable to other data structures, not just linear ones such as coins.

I am still aware that by applying our results to the above problem for one coin type, we obtain that we have to check up to four coins, which we know is too much. Actually we can prove that checking the sandwiches for two and three coins is sufficient because then we can infer the sandwich for just one coin (this is left as an exercise for the reader). Therefore, there are still efforts to be made in this direction.

Nonetheless, in the interactive approach (see Section 5.8 page 149) we do not need to test all combinations. Without it, the number of sandwiches to test would be at least in the cubic order, compared to the number of type of coins. With interactivity, the

computer never asks a question if the answer is unambiguous, by leveraging previous answers. The nice property we proved is that the number of questions, with respect to the size of the input, is linear.

Proactive Synthesis of Recursive Tree-to-String Functions from Examples

The following pages 120-160 contain the reformatted content of the peer-reviewed paper named " Proactive Synthesis of Recursive Tree-to-String Functions from Examples" (DOI: 10.4230/LIPIcs.ECOOP.2017.19) that appeared in the European Conference on Object-Oriented Programming (ECOOP) in 2017. I reproduce this work here with the permission of the authors, who are myself, Jad Hamza and Viktor Kuncak. In addition to this paper, at the end of my thesis, I provide an appendix with all the remaining proofs. (page 208 to page 214)

Contributions: *Overall, my key contributions to this paper are as follows: the walkthrough example and the examples throughout the paper; for Algorithm 2, the intersection of the automata is a subset of the two; for Algorithm 3, the proof that it asks a linear number of questions; the parser to parse scala files; the multiple choice questioning; and the adaptation of the algorithms for strings and values. Jad Hamza's key contributions were the idea to obtain the test set of size $O(n^3)$, the idea to use automata to represent solutions for Algorithm 2, the proof that the naive algorithm is NP-complete, and the core implementation of Prosy. Jad Hamza and I share the tribute for all other ideas, as we were together when we discovered them.*

Synthesis from examples enables non-expert users to generate programs by specifying examples of their behavior. A domain-specific form of such synthesis has been recently deployed in a widely used spreadsheet software product. In this paper we contribute to the foundations of such techniques and present a complete algorithm for the synthesis of a class of recursive functions defined by structural recursion over a given algebraic data type definition. The functions that we consider map an algebraic data type to a string; they are useful for pretty printing and serialization of programs and data, for example. We formalize our problem as learning deterministic sequential top-down tree-to-string transducers with a single state (1STS).

The first problem we consider is how to learn a tree-to-string transducer from any set of input/output examples provided by the user. We show that, given a set of input/output examples, checking whether there exists a 1STS consistent with these examples is NP-complete in general. In contrast, the problem can be solved in polynomial time under a (practically useful) closure condition that each subtree of a tree in the input/output example set is also part of the input/output examples.

As coming up with relevant input/output examples can be difficult for the user while creating hard constraint problems for the synthesizer, we also study a more automated active learning scenario in which the algorithm chooses the inputs for which the user provides the outputs. To determine a unique transducer, Our algorithm asks a worst-case linear number of queries as a function of the size of the algebraic data type definition.

To construct our algorithms we present two new results on formal languages.

First, we define a class of word equations, called sequential word equations, for which we prove that satisfiability can be solved in deterministic polynomial time. This is in contrast to the general word equations for which the best known complexity upper bound is in linear space.

Second, we close a long-standing open problem about the asymptotic size of test sets for context-free languages. A test set of a language of words L is a subset T of L such that any two word homomorphisms equivalent on T are also equivalent on L. We prove that it is possible to build test sets of cubic size for context-free languages, thus matching for the first time the lower bound found 20 years ago.

5.1 Introduction

Synthesis-by-example has been very successful in helping users deal with the tedious task of writing programs. This technique enables the user to specify input/output examples in order to describe the intended behavior of a desired program. Synthesis-by-example will then inspect the examples given by the user and generalize them into a program that respects these examples and that is able to handle other inputs.

Therefore, synthesis-by-example enables non-programmers to write programs without programming experience, and gives experienced users one more way of programming that could fit their needs. Current synthesis techniques usually rely on domain-specific heuristics to try and infer the desired program from the user. When there are multiple (non-equivalent) programs that are compatible with the input/output examples provided by the user, these heuristics might fail to choose the program that the user had in mind when writing the examples.

We believe it is important to have algorithms that provide formal guarantees based on strong theoretical foundations. Our algorithms ensure that the solution is found whenever it exists in a class of functions of interest. Furthermore, the algorithms ensure that the generated program is indeed the program the user wants by detecting once the solution is unique, or otherwise identifying a differentiating example whose output reduces the space of possible solutions.

In this paper, we focus on synthesizing the printing functions for objects or algebraic data types (ADT), which are at the core of many programming languages. Converting such structured values to strings is very common, including uses such as pretty printing, debugging, and serialization. Writing methods to convert objects to strings is repetitive and usually requires the user to code himself mutually recursive `toString` functions. Although some languages have default printing functions, these functions are often not adequate. For example, the object Person(“Joe”, 31) might have to be printed “Joe is 31

years old" for better readability, or "<td>Joe</td><td>31</td>" if printed as part of an HTML table. How *feasible* is it for the computer to learn these "printing" functions from examples?

The state of the art in this context [Laurence et al., 2014, Laurence, 2014] requires the user to provide *enough* examples. If the user gives *too few* examples, the synthesis algorithm is not guaranteed to return a valid printing function, and there is no simple way for the user to know which examples should be added so that the synthesis algorithm finishes properly.

Our contribution is to provide an algorithm that is able to determine exactly which questions to ask the user so that the desired function can be derived. Moreover, in order to learn a function, our algorithm (Algorithm 3) only needs to ask a linear number of questions (as a function of the size of the ADT declaration).

Our results hold for recursive functions that take ADT as input, and output strings. We model these functions by tree-to-string transducers, called *single-state sequential top-down tree-to-string transducers* [Boiret, 2016, Graehl and Knight, 2004, Helmut Seidl et al., 2015, Laurence et al., 2014, Staworko et al., 2009], or 1STS for short. In this formalism, objects are represented as labelled trees, and a transducer goes through the tree top down in order to display it as a string. *Single-state* means the transducer keeps no memory as it traverses the tree. *Sequential* is a shorthand for *linear* and *order-preserving*, meaning that each subtree is printed only once (linear), and the subtrees of a node are displayed in order (order-preserving). In particular, such transducers cannot directly represent recursive functions that have extra parameters alongside the tree to print. Our work on 1STSs establishes a foundation that may be used for larger classes of transducers.

Our goal is to learn a 1STS from a set of positive input/output examples, called a *sample*. We prove the problem of checking whether there exists a 1STS consistent with a given sample is NP-complete in general. Yet, we prove that when the given sample is closed under subtree, i.e., every tree in the sample has all of its subtrees in the sample, the problem of finding a compatible 1STS can be solved in polynomial time. For this, we reduce the problem of checking whether there exists an 1STS consistent with a sample to the problem of solving word equations. The best known algorithm to solve word equations takes linear space, and exponential time [Plandowski, 1999, Jeż, 2017]. However, we prove that the word equations we build are of a particular form, which we call sequential, and our first algorithm learns 1STSs by solving sequential equations in polynomial time.

We then tackle the problem of ambiguities that come from underspecified samples. More precisely, it is possible that, given a sample, there exist two 1STSs that are consistent with the sample, but that are not equivalent on a domain D of trees. We thus define the notion of *tree test set* of a domain D, which guarantees that, any two 1STSs which are equivalent on the tree test set are also equivalent on the whole domain D. We

give a method to build tree test sets of size $O(|D|^3)$ from a domain of trees given as a non-deterministic top-down automaton. Our second learning algorithm takes as input a domain D, builds the tree test set of D, and asks for the user the output to all trees in the tree test set. Our second algorithm then invokes our first algorithm on the given sample.

This construction relies on fundamental results on a known relation between sequential top-down tree-to-string transducers and morphisms (a morphism is a function that maps the concatenation of two words to the concatenation of their images), and on the notion of *test set* [Staworko et al., 2009]. Informally, a test set of a language of words L is a subset $T \subseteq L$ such that any two morphisms which are equivalent on T are also equivalent on L. In the context of 1STSs, the language L is a context-free language, intuitively representing the yield of the domain D mentioned above. Prior to our work announced in [Mayer and Hamza, 2016], the best known construction for a test set of a context-free grammar G produced test sets of size $O(|G|^6)$, while the best known lower bound was $O(|G|^3)$ [Plandowski, 1994, Plandowski, 1995]. We show the $O(|G|^3)$ is in fact tight, and give a construction that, given any grammar G, produces a test set for G of size $O(|G|^3)$.

Finally, our third and, from a practical point of view, the main algorithm, improves the second one by analyzing the previous outputs entered by the user, in order to infer the next output. More specifically, the outputs previously entered by the user give constraints on the transducer being learned, and therefore restrict the possible outputs for the next questions. Our algorithm computes these possible outputs and, when there is only one, skips the question. Our algorithm only asks the user a question when there are at least two possible outputs for a particular input. The crucial part of this algorithm is to prove that such ambiguities happen at most $O(|D|)$ times. Therefore, our third algorithm asks the user only $O(|D|)$ questions, greatly improving our second one that asks $O(|D|^3)$ questions. Our result relies on carefully inspecting the word equations produced by the input/output examples.

We implemented our algorithms in an open-source tool available at https://github.com/epfl-lara/prosy. In sections 5.9 and 5.10, we describe how to extend our algorithms and tool to ADTs which contain String (or Int) as a primitive type. We call the implementation of our algorithms *proactive* synthesis, because it produces a *complete* set of questions ahead-of-time whose answers will help to synthesize a unique tree-to-string function, *filters out* future questions whose answer could be actively inferred after each user's answer, and produces *suggestions* as multiple choice or pre-filled answers to minimize the answering effort.

Contributions

Our paper makes the following contributions:

1. A new efficient algorithm to synthesize recursive functions from examples. We give a polynomial-time algorithm to obtain a 1STS from a sample *closed under subtree*. When the sample is not necessarily closed under subtree, we prove that the problem of checking whether there exists a 1STS consistent with the sample is NP-complete (Section 5.6). This result is based on a fundamental contribution:
   - A polynomial-time algorithm for solving a class of word equations that come from a synthesis problem (*sequential* word equations, Section 5.6).
2. An algorithm that synthesize recursive functions without ambiguity by generating an exhaustive set of questions to ask to the user, in the sense that any two recursive functions that agree on these inputs, are equal on their entire domain (Section 5.7). This is based on the following fundamental contribution:
   - A constructive upper bound of $O(|G|^3)$ on the size of a test set for a context-free grammar G, improving on the previous known bound of $O(|G|^6)$ [Plandowski, 1994, Plandowski, 1995] (Section 5.7).
3. A proactive and efficient algorithm that synthesizes recursive functions, which only requires the user to enter outputs for the inputs determined by the algorithm. Formally, we present an interactive algorithm to learn a 1STS for a domain of trees, with the guarantee that the obtained 1STS is functionally unique. Our algorithm asks the user only a *linear* number of questions (Section 5.8).
4. A construction of a linear tree test set for data types with Strings, which enables constructing a small set of inputs that distinguish between two recursive functions (Section 5.9).
5. An implementation of our algorithms as an interactive command-line tool (Section 5.10)

We note that the fundamental contributions of (1) and (2) are new general results about formal languages and may be of interest on their own.

For readability purposes, we only show proof sketches and intuition; detailed proofs are located in the appendices (pages 208 to 214).

5.2 Example Run of Our Synthesis Algorithm

To motivate our problem domain, we present a run of our algorithm on an example. The example is an ADT representing a context-free grammar. It defines its custom alphabet (Char), words (CharList), and non-terminals indexed by words (NonTerminal). A rule (Rule) is a pair made of a non-terminal and a sequence of symbols (ListSymbol), which can be non-terminals or terminals (Terminal). Finally, a grammar is a pair made of a (starting) non-terminal and a sequence of rules.

The input of our algorithm is the following file (written in Scala syntax):

```
abstract class Char
case class a() extends Char
case class b() extends Char

abstract class CharList
case class NilChar() extends CharList
case class ConsChar(c: Char, l: CharList) extends CharList

abstract class Symbol
case class Terminal(t: Char) extends Symbol
case class NonTerminal(s: CharList) extends Symbol

case class Rule(lhs: NonTerminal, rhs: ListSymbol)

abstract class ListRule
case class ConsRule(r: Rule, tail: ListRule) extends ListRule
case class NilRule() extends ListRule

abstract class ListSymbol
case class ConsSymbol(s: Symbol, tail: ListSymbol) extends ListSymbol
case class NilSymbol() extends ListSymbol

case class Grammar(s: NonTerminal, r: ListRule)
```

We would like to synthesize a recursive tree-to-string function print, such that if we compute, for example:

```
print(Grammar(NonTerminal(NilChar()),
    ConsRule(Rule(NonTerminal(NilChar()),
      ConsSymbol(Terminal(a()),
        ConsSymbol(NonTerminal(NilChar()),
          ConsSymbol(Terminal(b()), NilSymbol())))),
      ConsRule(Rule(NonTerminal(NilChar()),
          NilSymbol())), NilRule()))
```

the result should be:

```
Start: N
N -> a N b
N ->
```

We would like the print function to handle any valid Grammar tree.

When given these class definitions above, our algorithm precomputes a set of terms from the ADT, so that any two single-state recursive functions which output the same Strings for these terms also output the same Strings for any term from this ADT. (This is related

to the notion of *tree test set* defined in Section 5.7.2.) Our algorithm will determine the outputs for these terms by interacting with the user and asking questions. Overall, for this example, our algorithm asks the output for 14 terms.

For readability, question lines provided by the synthesizer are indented. Lines entered by the user finish by the symbol ↩ , meaning that she pressed the ENTER key. Everything after ↩ on the same line is our comment on the interaction. "It" usually refers to the synthesizer. After few interactions, the questions themselves are shortened for conciseness. The interaction is the following:

```
  Proactive Synthesis.
  If you ever want to enter a new line, terminate your line by \ and press Enter.
  What should be the function output for the following input tree?
  a
a↩
  What should be the function output for the following input tree?
  b
b↩
  NilChar ?
↩                 indeed, NilChar is an empty string.
  NilSymbol ?
↩                 No symbol at the right-hand-side of a rule
  NilRule ?
↩                 No rule left describing the grammar
  What should be the function output for the following input tree?
  Terminal(a)
  Something of the form: [...]a[...]
a↩                Terminals contain only one char. Note the hint provided by the synthesizer.
  NonTerminal(NilChar) ?
N↩
  ConsChar(b,NilChar) ? Something of the form: [...]b[...]
b↩                A ConsChar is a concatenation of a char and a string
  What should be the function output for the following input tree?
  NonTerminal(ConsChar(b,NilChar))
  1) Nb
  2) bN
  Please enter a number between 1 and 2, or 0 if you really want to enter your answer manually
1↩                Note that it was able to infer only two possibilities, thus the closed question.
  Grammar(NonTerminal(NilChar),NilRule) ? Something of the form: [...]N[...]
Start: N↩
  ConsSymbol(Terminal(a),NilSymbol) ? Something of the form: [...]'a'[...]
 a↩               Symbols on the right-hand-side of a Rule are prefixed with a space
Rule(NonTerminal(NilChar),NilSymbol) ? Something of the form: [...]N[...]
N ->↩             A rule with no symbols on the right-hand-side
  ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule) ?
  Something of the form: [...]N  ->[...]
\↩                A newline
```

```
N ->↩
  What should be the function output for the following input tree?
  Rule(NonTerminal(NilChar),ConsSymbol(Terminal(‘a‘),NilSymbol))
  1) N ‘a‘->
  2) N - ‘a‘>
  3) N -> ‘a‘
  4) N ‘a‘ ->
  Please enter a number between 1 and 4, or 0 if you really want to enter your answer manually
3↩
```

The synthesizer then emits the desired recursive tree-to-string function, along with a complete set of the tests that determine it:

```
def print(t: Any): String = t match {
  case a() ⇒ "a"
  case b() ⇒ "b"
  case NilChar() ⇒ ""
  case ConsChar(t1,t2) ⇒ print(t1) + print(t2)
  case Terminal(t1) ⇒ "‘" + print(t1) + "‘"
  case NonTerminal(t1) ⇒ "N" + print(t1)
  case Rule(t1,t2) ⇒ print(t1) + " ->" + print(t2)
  case ConsRule(t1,t2) ⇒ "\n" + print(t1) + print(t2)
  case NilRule() ⇒ ""
  case ConsSymbol(t1,t2) ⇒ " " + print(t1) + print(t2)
  case NilSymbol() ⇒ ""
  case Grammar(t1,t2) ⇒ "Start: " + print(t1) + print(t2)
} // the part below is a contract, not needed to execute the recursive function
ensuring { (res: string) => res == (t match {
  case a() => "a"
  case b() => "b"
  case NilChar() => ""
  case NilSymbol() => ""
  case NilRule() => ""
  case Terminal(a()) => "a"
  case NonTerminal(NilChar()) => "N"
  case ConsChar(b(),NilChar()) => "b"
  case NonTerminal(ConsChar(b(),NilChar())) => "Nb"
  case Grammar(NonTerminal(NilChar()),NilRule()) => "Start: N"
  case ConsSymbol(Terminal(a()),NilSymbol()) => " a"
  case Rule(NonTerminal(NilChar()),NilSymbol()) => "N ->"
  case ConsRule(Rule(NonTerminal(NilChar()),NilSymbol()),NilRule()) => "\nN ->"
  case Rule(NonTerminal(NilChar()),ConsSymbol(Terminal(a()),NilSymbol())) => "N -> a"
  case _ => res})
}
```

Observe that, in addition to the program, the synthesis system emits as a postcondition (after the `ensuring` construct) the recorded input/output examples (tests). Our work

enables the construction of an IDE that would automatically maintain the bidirectional correspondence between the body of the recursive function and the postcondition that specifies its input/output tests. If the user modifies an example in the postcondition, the system could re-synthesize the function, asking for clarification in cases where the tests become ambiguous. If the user modifies the program, such system can regenerate the tests.

Depending on user's answers, the total number of questions that the synthesizers asks varies (see section 5.11). Nonetheless, the properties that we proved for our algorithm guarantee that the number of questions remains at most *linear* as a function of the size of the algebraic data type declaration.

When the user enters outputs which are not consistent, i.e., for which there exists no printing function in the class of functions that we consider, our tool directly detects it and warns the user. For instance, for the tree
ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule)
if the user enters N- > with the space and the dash inverted, the system detects that this output is not consistent with the output provided for tree
Rule(NonTerminal(NilChar),NilSymbol)
and asks the question again.

```
We cannot have the transducer convert ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule)
to N− >.
Please enter something consistent with what you previously entered (e.g. 'N −>','N −>bar
  ',...)?
```

5.3 Discussion

5.3.1 Advantages of Synthesis Approach

It is important to emphasize that in the approach we outline, the developer not only enters less text in terms of the number of characters than in the above source code, but that the input from the user is entirely in terms of concrete input-output *values*, which can be easier to reason about for non-expert users than recursive programs with variable names and control-flow.

It is notable that the synthesizer in many cases offered suggestions, which means that the user often simply needed to check whether one of the candidate outputs is acceptable. Even in cases where the user needed to provide new parts of the string, the synthesizer in many cases guided the user towards a form of the output consistent with the outputs provided so far. Because of this knowledge, the synthesizer could also be stopped early by, for example, guessing the unknown information according to some preference (e.g. replacing all unknown string constants by empty strings), so the user can in many cases

obtain a program by providing a very small amount of information.

Such easy-to-use interactions could be implemented as a pretty printing wizard in an IDE, for example triggered when the user starts to write a function to convert an ADT to a String.

Our experience in writing pretty printers manually suggests that they often require testing to ensure that the generated output corresponds to the desired intuition of the developer, suggesting that input-output tests may be a better form of specification even if in cases where they are more verbose. We therefore believe that it is valuable to make available to users and developers sucn an alternative method of specifying recursive functions, a method that can co-exist with the conventional explicitly written recursive functions and the functions derived automatically (but generically) by the compiler (such as default printing of algebraic data type values in Scala), or using polytypic programming approaches [Jansson, 2000] and serialization libraries [Miller et al., 2013]. (Note that the generic approaches can reduce the boilerplate, but do not address the problem of unambiguously generalizing *examples* to recursive functions.)

5.3.2 Challenges in Obtaining Efficient Algorithms

The problem of inferring a program from examples requires recovering the constants embedded in the program from the results of concatenating these constants according to the structure of the given input tree examples. This presents two main challenges. The first one is that the algorithm needs to split the output string and identify which parts correspond to constants and which to recursive calls. This process becomes particularly ambiguous if the alphabet used is small or if some constants are empty strings. A natural way to solve such problems is to formulate them as a conjunction of word equations. Unfortunately, the best known deterministic algorithms for solving word equations run in exponential time (the best complexity upper bound for the problem takes linear space [Plandowski, 1999, Jeż, 2017]). Our paper shows that, under an assumption that, when specifying printing of a tree, we also specify printing of its subtrees, we obtain word equations solvable in *polynomial time*.

The next challenge is the number of examples that need to be solved. Here, a previous upper bound derived from the theory of test sets of context-free languages was $\Omega(n^6)$, which, even if polynomial, results in impractical number of user interactions. In this paper we improve this theoretical result and show that tests sets are in fact in $O(n^3)$, asymptotically matching the known lower bound.

Furthermore, if we allow the learning algorithm to choose the inputs one by one after obtaining outputs, the overall learning algorithm has a *linear* number of queries to user and to equation solving subroutine, as a function of the size of tree data type definition. Our contributions therefore lead to tools that have completeness guarantees with much

less user input and a shorter running time than the algorithms based on prior techniques.

We next present our algorithms as well as the results that justify their correctness and completeness.

5.4 Notation

We start by introducing our notation and terminology for some standard concepts. Given a (partial) function from $f : A \to B$, and a set C, $f_{|C}$ denotes the (partial) function $g : A \cap C \to B$ such that $g(a) = f(a)$ for all $a \in A \cap C$.

A word (string) is a finite sequence of elements of a finite set Σ, which we call an *alphabet*.

A *morphism* $f : \Sigma^* \to \Gamma^*$ is a function such that $f(\varepsilon) = \varepsilon$ and for every $u, v \in \Sigma^*$, $f(u \cdot v) = f(u) \cdot f(v)$, where the symbol '$\cdot$' denotes the concatenation of words (strings).

A *non-deterministic finite automaton (NFA)* is a tuple $(\Gamma, Q, q_i, F, \delta)$ where Γ is the alphabet, Q is the set of states, $q_i \in Q$ is the initial state, F is the set of final states, $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation. When the transition relation is deterministic, that is for all $q, p_1, p_2 \in Q, a \in \Gamma$, if $(q, a, p_1) \in \delta$ and $(q, a, p_2) \in \delta$, then $p_1 = p_2$, we say that A is a *deterministic finite automaton (DFA)*.

A *context-free grammar* G is a tuple (N, Σ, R, S) where:

- N is a set of *non-terminals*,
- Σ is a set of *terminals*, disjoint from N,
- $R \subseteq N \times (N \cup \Sigma)^*$ is a set of *production rules*,
- $S \in N$ is the starting non-terminal symbol.

A production $(A, rhs) \in R$ is denoted $A \to rhs$. The *size* of G, denoted $|G|$, is the sum of sizes of each production in R: $\sum_{A \to rhs \in R} 1 + |rhs|$. A grammar is *linear* if for every production $A \to rhs \in R$, the *rhs* string contains at most one occurrence of N. By an abuse of notation, we denote by G the set of words produced by G.

5.4.1 Trees and Domains

A *ranked alphabet* Σ is a set of pairs (f, k) where f is a symbol from a finite alphabet, and $k \in \mathbb{N}$. A pair (f, k) of a ranked alphabet is also denoted $f^{(k)}$. We say that symbol f has a *rank* (or *arity*) equal to k. We define by $\mathcal{T}_\Sigma$ the set of trees defined over alphabet Σ. Formally, $\mathcal{T}_\Sigma$ is the smallest set such that, if $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, and $f^{(k)} \in \Sigma$ for some $k \in \mathbb{N}$,

then $f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$. A set of trees T is *closed under subtree* if for all $f(t_1, \ldots, t_k) \in T$, for all $i \in \{1, \ldots, k\}$, $t_i \in T$.

A top-down tree automaton T is a tuple (Σ, Q, I, δ) where Σ is a ranked alphabet, $I \subseteq Q$ is the set of initial states, and $\delta \subseteq \Sigma \times Q \times Q^*$. The set of trees $\mathcal{L}(T)$ recognized by T is defined recursively as follows. For $f^{(k)} \in \Sigma$, $q \in Q$, and $t = f(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$, we have $t \in \mathcal{L}(T)_q$ iff there exists $(f, q, q_1 \cdots q_k) \in \delta$ such that for $1 \leq i \leq k$, $t_i \in \mathcal{L}(T)_{q_i}$. The set $\mathcal{L}(T)$ is then defined as $\bigcup_{q \in I} \mathcal{L}(T)_q$.

Algebraic data types are described by the notion of *domain*, which is a set of trees recognized by a top-down tree automaton $T = (\Sigma, Q, I, \delta)$. The *size* of the domain is the sum of sizes of each transition in δ, that is $\sum_{(f^{(k)}, q, q_1 \cdots q_k) \in \delta} 1 + k$.

Example 1. In this example and the following ones, we illustrate our notions using an encoding of HTML-like data structures. Consider the following algebraic data type definitions in Scala:

```
abstract class Node
case class node(t: Tag, l: List) extends Node

abstract class Tag
case class div() extends Tag
case class pre() extends Tag
case class span() extends Tag

abstract class List
case class cons(n: Node, l: List) extends List
case class nil() extends List
```

The corresponding domain D_{html} is described by the following:

$$\begin{aligned}
\Sigma &= \{\mathrm{nil}^{(0)}, \mathrm{cons}^{(2)}, \mathrm{node}^{(2)}, \mathrm{div}^{(0)}, \mathrm{pre}^{(0)}, \mathrm{span}^{(0)}\} \\
Q &= \{\mathrm{Node}, \mathrm{Tag}, \mathrm{List}\} \\
I &= \{\mathrm{Node}, \mathrm{Tag}, \mathrm{List}\} \\
\delta &= \{(\mathrm{node}, \mathrm{Node}, (\mathrm{Tag}, \mathrm{List})), \\
&\qquad (\mathrm{div}, \mathrm{Tag}, ()), (\mathrm{pre}, \mathrm{Tag}, ()), (\mathrm{span}, \mathrm{Tag}, ()), \\
&\qquad (\mathrm{cons}, \mathrm{List}, (\mathrm{Node}, \mathrm{List})), \\
&\qquad (\mathrm{nil}, \mathrm{List}, ())\}
\end{aligned}$$

5.4.2 Transducers

A *deterministic, sequential, single-state, top-down tree-to-string transducer* τ (1STS for short) is a tuple (Σ, Γ, δ) where:

- Σ is a ranked alphabet (of trees),
- Γ is an alphabet (of words),
- δ is a function over Σ such that $\forall f^{(k)} \in \Sigma.\ \delta(f) \in (\Gamma^*)^{k+1}$.

Note that the transducer does not depend on a particular domain for Σ, but instead can map any tree from $\mathcal{T}_\Sigma$ to a word. Later, when we present our learning algorithms for 1STSs, we restrict ourselves to particular domains provided by the user of the algorithm.

We denote by $[\![\tau]\!]$ the function from trees to words associated with the 1STS τ. Formally, for every $f^{(k)} \in \Sigma$, we have $[\![\tau]\!](f(t_1, \ldots, t_k)) = u_0 \cdot [\![\tau]\!](t_1) \cdot u_1 \cdots [\![\tau]\!](t_k) \cdot u_k$ if $\delta(f) = (u_0, u_1, \ldots, u_k)$. When clear from context, we abuse notation and use τ as a shorthand for the function $[\![\tau]\!]$.

Example 2. A transducer $\tau = (\Sigma, \Gamma, \delta)$ converting HTML trees into a convenient syntax for some programmatic templating engines[2] may be described by:

$$\begin{aligned}
\Sigma =& \{\mathrm{nil}^{(0)}, \mathrm{cons}^{(2)}, \mathrm{node}^{(2)}, \mathrm{div}^{(0)}, \mathrm{pre}^{(0)}, \mathrm{span}^{(0)}\}\} \\
\Gamma =& [\textit{All symbols}] \\
\delta(\mathrm{node}) =& (\text{“<.”}, \varepsilon, \varepsilon) \\
\delta(\mathrm{div}) =& (\text{“div”}) \qquad \delta(\mathrm{pre}) = (\text{“pre”}) \qquad \delta(\mathrm{span}) = (\text{“span”}), \\
\delta(\mathrm{cons}) =& (\text{“(”}, \text{“)”}, \varepsilon) \qquad \delta(\mathrm{nil}) = (\varepsilon)
\end{aligned}$$

In Scala, this is written as follows:

```
def tau(input: Tree) = input match {
  case node(t, l) ⇒ "<." + tau(t) + "" + tau(l) + ""
  case div() ⇒ "div"
  case pre() ⇒ "pre"
  case span() ⇒ "span"
  case cons(n, l) ⇒ "(" + tau(n) + ")" + tau(l) + ""
  case nil() ⇒ ""
}
```

For example, tau(node(div,cons(node(span,nil,cons(node(pre,nil))))))= "<.div(<.span())(<.pre())"

5.5 Transducers as Morphisms

For a given alphabet Σ, a 1STS (Σ, Γ, δ) is completely determined by the constants that appear in δ. This allows us to define a one-to-one correspondence between transducers

[2] https://github.com/lihaoyi/scalatags

and morphisms. This correspondence is made through what we call the *default transducer*. More specifically, Γ is the set $\overline{\Sigma} = \{(f, i) \mid f^{(k)} \in \Sigma \wedge 0 \leq i \leq k\}$ and for all $f^{(k)} \in \Sigma$, we have $\delta(f) = ((f, 0), (f, 1), \ldots, (f, k))$. The default transducer produces sequences of pairs from $\overline{\Sigma}$.

Example 3. For $\Sigma = \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}$, τ_Σ is:

$$\begin{aligned}
\Gamma =& \{ \text{(node, 0), (node, 1), (node, 2), (div, 0), (pre, 0), (span, 0)} \\
& \text{(cons, 0), (cons, 1), (cons, 2), (nil, 0)}\} \\
\delta(\text{node}) =& ((\text{node}, 0), (\text{node}, 1), (\text{node}, 2)) \\
\delta(\text{div}) =& (\text{div}, 0) \qquad\qquad \delta(\text{pre}) = (\text{pre}, 0) \quad \delta(\text{span}) = (\text{span}, 0) \\
\delta(\text{cons}) =& ((\text{cons}, 0), (\text{cons}, 1), (\text{cons}, 2)) \qquad \delta(\text{nil}) = (\text{nil}, 0))
\end{aligned}$$

In Scala, τ_Σ can be written as follows (+ is used to concatenate elements and lists):

```
def tauSigma(input: Tree): List[Σ̄] = input match {
  case node(t, l) ⇒ (node,0) + tauSigma(t) + (node,1) + tauSigma(l) + (node,2)
  case div() ⇒ (div,0)
  case pre() ⇒ (pre,0)
  case span() ⇒ (span,0)
  case cons(n, l) ⇒ (cons,0) + tauSigma(n) + (cons,1) + tauSigma(l) + (cons,2)
  case nil(n, l) ⇒ (nil,0)
}
```

Lemma 5.5.1.1. *For any ranked alphabet Σ, the function $[\![\tau_\Sigma]\!]$ is injective.*

Proof. The proof of this result is located in Appendix E.1. □

Following Lemma 5.5.1.1, for a word $w \in \overline{\Sigma}^*$, we define $\mathsf{tree}(w)$ to be the unique tree (when it exists) such that $\tau_\Sigma(\mathsf{tree}(w)) = w$. We show in Figure 5.2 how to obtain $\mathsf{tree}(w)$ in linear time from w.

For a 1STS $\tau = (\Sigma, \Gamma, \delta)$, we define the morphism $\mathtt{morph}[\tau]$ from $\overline{\Sigma}$ to Γ^*, and such that, for all $f^{(k)} \in \Sigma$, $i \in \{0, \ldots, k\}$, $\mathtt{morph}[\tau](f, i) = u_i$ where $\delta(f) = (u_0, u_1, \ldots, u_k)$. Conversely, given a morphism $\mu : \overline{\Sigma} \to \Gamma^*$, we define $\mathtt{sts}(\mu)$ as τ_Σ where each output $l \in \overline{\Sigma}$ is replaced by $\mu(l)$.

```
def tree(w: List[Σ̄]): Tree =
  if w is empty or does not start with some (f, 0):
    throw error
  let (f, 0) = w.head
  w ← w.tail
  for i from 1 to arity(f)
    t_i = tree(w)
    assert(w starts with (f, i))
    w ← w.tail
  return f(t_1, ..., t_k)
```

Figure 5.2 – Parsing algorithm to obtain $\mathsf{tree}(w)$ from a word $w \in \overline{\Sigma}^*$. When the algorithm fails, because of a pattern matching error or because of the thrown exception, it means there exists no t such that $\tau_\Sigma(t) = w$.

Example 4. For Example 2, $\mathtt{morph}[\tau]$ is defined by:

$$\begin{aligned}
\mathtt{morph}[\tau](\mathrm{node}, 0) &= \text{“<.”} & \mathtt{morph}[\tau](\mathrm{cons}, 0) &= \text{“(”} \\
\mathtt{morph}[\tau](\mathrm{node}, 1) &= \varepsilon & \mathtt{morph}[\tau](\mathrm{cons}, 1) &= \text{“)”} \\
\mathtt{morph}[\tau](\mathrm{node}, 2) &= \varepsilon & \mathtt{morph}[\tau](\mathrm{cons}, 2) &= \varepsilon \\
\mathtt{morph}[\tau](\mathrm{div}, 0) &= \text{“div”} & \mathtt{morph}[\tau](\mathrm{nil}, 0) &= \varepsilon \\
\mathtt{morph}[\tau](\mathrm{pre}, 0) &= \text{“pre”} & \mathtt{morph}[\tau](\mathrm{span}, 0) &= \text{“span”}
\end{aligned}$$

Note that for any morphism: $\mu : \overline{\Sigma} \to \Gamma^*$, $\mathtt{morph}[\mathtt{sts}(\mu)] = \mu$ and for any 1STS τ, $\mathtt{sts}(\mathtt{morph}[\tau]) = \tau$. Moreover, we have the following result, which expresses the output of a 1STS τ using the morphism $\mathtt{morph}[\tau]$.

Lemma 5.5.1.2. *For a* 1STS τ*, and for all* $t \in \mathcal{T}_\Sigma$*,* $\mathtt{morph}[\tau](\tau_\Sigma(t)) = \tau(t)$*.*

Proof. Follows directly from the definitions of $\mathtt{morph}[\tau]$ and τ_Σ. □

Example 5. Let $t = \mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil})$. For `morph[]` defined as in Example 4 and the transducer τ as in Example 2, the left-hand-side of the equation of Lemma 5.5.1.2 translates to:

$$\begin{aligned}
&\mathtt{morph}[\tau](\tau_\Sigma(t)) \\
&= \mathtt{morph}[\tau](\tau_\Sigma(\mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil}))) \\
&= \mathtt{morph}[\tau]((\mathrm{cons}, 0)(\mathrm{node}, 0)(\mathrm{div}, 0)(\mathrm{node}, 1)(\mathrm{nil}, 0)(\mathrm{node}, 2)(\mathrm{cons}, 1)(\mathrm{nil}, 0)(\mathrm{cons}, 2)) \\
&= \text{“(”} \cdot \text{“<.”} \cdot \text{“div”} \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon \\
&= \text{“(<.div)”}
\end{aligned}$$

Similarly, the right-hand-side of the equation can be computed as follows:

$$\begin{aligned}
&\tau(t)\\
&= \tau(\mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil}))\\
&= \text{“(”} \cdot \tau(\mathrm{node}(\mathrm{div}, \mathrm{nil})) \cdot \text{“)”} \cdot \tau(\mathrm{nil}) \cdot \varepsilon\\
&= \text{“(”} \cdot \text{“<.”} \cdot \tau(\mathrm{div}) \cdot \varepsilon \cdot \tau(\mathrm{nil}) \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon\\
&= \text{“(<.div)”}
\end{aligned}$$

We thus obtain that checking equivalence of 1STSs can be reduced to checking equivalence of morphisms on a context-free language.

Lemma 5.5.1.3 (See [Staworko et al., 2009])**.** *Let τ_1 and τ_2 be two* 1STS*s, and $D = (\Sigma, Q, I, \delta)$ a domain. Then $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$ if and only if $\mathtt{morph}[\tau_1]_{|G} = \mathtt{morph}[\tau_2]_{|G}$ where G is the context-free language $\{\tau_\Sigma(t) \mid t \in D\}$.*

Proof. Follows from Lemma 5.5.1.2. G is context-free, as it can be recognized by the grammar $(N_G, \overline{\Sigma}, R_G, S_G)$ where:

- $N_G = \{S_G\} \cup \{A_q \mid q \in Q\}$, where S_G is a fresh symbol used as the starting non-terminal,
- The productions are:

$$\begin{aligned}
R_G = \quad & \{A_q \to \ (f,0) \cdot A_{q_1} \cdot (f,1) \cdots A_{q_k} \cdot (f,k) \mid f^{(k)} \in \Sigma \ \wedge (q, f, (q_1, \ldots, q_k)) \in \delta\}\\
& \cup \{S_G \to A_q \mid q \in I\}
\end{aligned}$$

Note that the size of G is linear in the size of $|D|$ (as long as there are no unused states in D). □

5.6 Learning 1STS from a Sample

We now present a learning algorithm for learning 1STSs from sets of input/output examples, or a *sample*. Formally, a sample $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$ is a partial function from trees to words, or alternatively, a set of pairs (t, w) with $t \in \mathcal{T}_\Sigma$ and $w \in \Gamma^*$ such that each t is paired with at most one w.

5.6.1 NP-completeness of the general case

In general, we prove that finding whether there exists a 1STS consistent with a given a sample is an NP-complete problem. To prove NP-hardness, we reduce the one-in-three

positive SAT problem. This problem asks, given a formula φ with no negated variables, whether there exists an assignment such that for each clause of φ, exactly one variable (out of three) evaluates to true.

Theorem 5.6.1.1. *Given a sample $\mathcal{S}$, checking whether there exists a* 1STS τ *such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ is an* NP*-complete problem.*

Proof. (Sketch) We can check for the existence of τ in NP using the following idea. Every input/output example from the sample gives constraints on the constants of τ. Therefore, to check for the existence of τ, it is sufficient to non-deterministically guess constants which are subwords of the given output examples. We can then verify in polynomial-time whether the guessed constants form a 1STS τ which is consistent with the sample $\mathcal{S}$.

To prove NP-hardness, we consider a formula φ, instance of the one-in-three positive SAT. The formula φ has no negated variables, and is satisfiable if there exists an assignment to the boolean variables such that for each clause of φ, exactly one variable (out of three) evaluates to true.

We construct a sample $\mathcal{S}$ such that there exists a 1STS τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ if and only if φ is satisfiable. For each clause $(x, y, z) \in \varphi$, we construct an input/output example of the form $\mathcal{S}(x(y(z(\mathsf{nil})))) = a\#$ where x, y and z are symbols of arity 1 corresponding to the variables of the same name in φ, nil is a symbol of arity 0, and a and $\#$ are two special characters. Moreover, we add an input/output example stating that $\mathcal{S}(\mathsf{nil}) = \#$.

This construction forces the fact that a 1STS τ consistent with $\mathcal{S}$ will have a non-empty output (a) for exactly one symbol out of x, y, and z (therefore matching the requirements of one-in-three positive SAT formulas).

The complete proof is located in Appendix E.2. □

In the sequel, we prove that if the domain of the given sample is closed under subtree, this problem can be solved in polynomial time.

5.6.2 Word Equations

Our learning algorithm relies on reducing the problem of learning a 1STS from a sample to the problem of solving word equations. In general, the best known algorithm for solving word equations is in linear space [Plandowski, 1999, Jeż, 2017], and takes exponential time to run. When the domain of the sample $\mathcal{S}$ is closed under subtree, the equations we construct have a particular form, and we call them *sequential formulas*. We show there is a polynomial-time algorithm for checking whether a sequential word formula is satisfiable.

Definition 6. Let $\mathbb{X}$ be a finite set of *variables*, and Γ a finite alphabet. A *word equation* e is a pair $y_1 = y_2$ where $y_1, y_2 \in (\mathbb{X} \cup \Gamma)^*$. A *word formula* φ is a conjunction of word equations. An *assignment* is a function from $\mathbb{X}$ to Γ^*, and can be seen as a morphism $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that $\mu(a) = a$ for all $a \in \Gamma$.

A word formula is satisfiable if there exists an assignment $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that for all equations $y_1 = y_2$ in φ, $\mu(y_1) = \mu(y_2)$.

A word formula φ is called *sequential* if: 1) for each equation $y_1 = y_2 \in \varphi$, $y_2 \in \Gamma^*$ contains no variable, and $y_1 \in (\Gamma \cup \mathbb{X})^*$ contains at most one occurrence of each variable, 2) for all equations $y = \_$ and $y' = \_$ in φ, either y and y' do not have variables in common, or $y_{|\mathbb{X}} = y'_{|\mathbb{X}}$, that is y and y' have the same sequence of variables. We used the name *sequential* due to this last fact.

Example 7. For $X_1, X_2, X_3, X_4, X_5 \in \mathbb{X}$ and $p, q \in \Gamma^*$, each of the four formulas below is sequential:

$$X_1 = pq \qquad X_1X_3 = qpqpqqpqpq \wedge X_1qX_3 = qpqpqqqpqpq$$
$$X_1pX_2qX_3 = qppq \quad X_1pqX_2X_3 = pqpqpp \wedge X_1X_2qpX_3 = pqpppqp \wedge X_5pX_4 = qpq$$

The following formulas (and any formula containing them) are not sequential:

$$X_1pqX_2X_3 = pX_3pq \quad \text{(rhs is not in } \Gamma^*\text{)}$$
$$X_1pqX_2pX_3X_2 = ppqqpp \quad (X_2 \text{ appears twice in lhs})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_2pX_5 = qpq \quad (X_2 \text{ is shared})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_1pX_3X_2 = pqppp \quad \text{(different orderings of } X_1\ X_2\ X_3\text{)}$$

We prove that any sequential word formula φ can be solved in polynomial time.

Lemma 5.6.2.1. *Let φ be a sequential word formula. Let n be the number of equations in φ, V the number of variables, and C be the size of the largest constant appearing in φ. We can determine in polynomial time $O(nVC)$ whether φ is satisfiable. When it is, we can also produce a satisfying assignment for φ.*

Proof. By definition of sequential, φ can be written as $\varphi_1 \wedge \cdots \wedge \varphi_l$ for some $l \in \mathbb{N}$, where for $i \neq j$, φ_i and φ_j do not have variables in common. We can thus check for satisfiability of φ by checking satisfiability of each φ_i independently. Let ψ be one of φ_i for $i \in \{1, \ldots, l\}$.

By definition of sequential, we know there exists $n \in \mathbb{N}$ with $\psi \equiv y_1 = w_1 \wedge \cdots \wedge y_n = w_n$, and there exist $k \in \mathbb{N}$ and $X_0, \ldots, X_k \in \mathbb{X}$, such that for all $i \in \{1, \ldots, n\}$, $w_i \in \Gamma^*$, and $y_{i|\mathbb{X}} = X_0 \cdots X_k$.

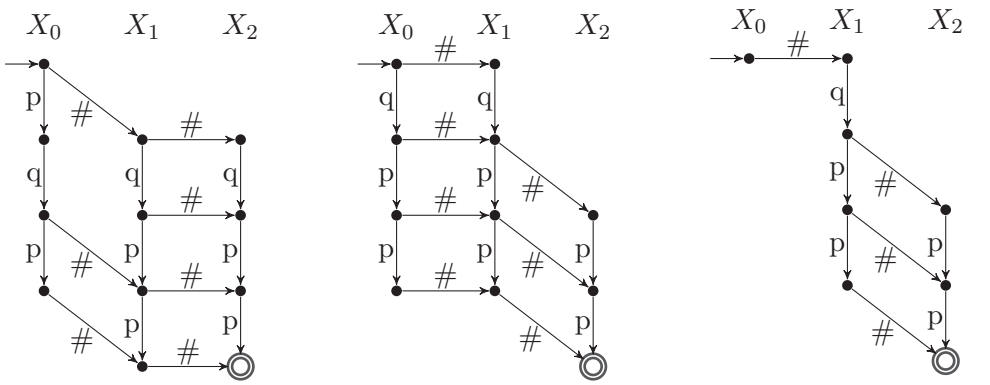


Figure 5.3 – On the left, two automata representing the solutions of equations X_0 p X_1 X_2 = pqpp and X_0 X_1 p X_2 = qppp respectively. On the right, their intersection represents the solutions of the conjunction of equations. Note that the third automaton can be obtained from the first (and the second) by removing states and transitions.

The outline of the proof is the following. For $i \in \{1, \ldots, n\}$, we build an acyclic DFA, denoted $A_i = \mathsf{automaton}(y_i, w_i)$, that recognizes the set

$$\{\mu(X_0)\#\mu(X_1)\cdots\#\mu(X_k) \mid \mu : \mathbb{X} \to \Gamma^* \wedge \mu(y_i) = w_i\}$$

where # is a special character we introduce, used as a separator.

Then, there exists an assignment $\mu : \mathbb{X} \to \Gamma^*$ such that for all $i \in \{1, \ldots, n\}$, $\mu(y_i) = w_i$ if and only if $A_1 \cap \cdots \cap A_n \neq \emptyset$. We then show that the emptiness of this intersection can be checked in polynomial time, due to the particular form of the automata. (In general, checking the emptiness of the intersection of n automata is a PSPACE-complete problem, and thus takes exponential time to check.)

We now give the formal details of the proof. Let $i \in \{1, \ldots, n\}$, and $y_i = X_0 u_1 X_1 \cdots u_k X_k$. We define $A_i = \mathsf{automaton}(y_i, w_i) = (Q_i, q_i, \delta_i)$ as follows:

- $Q_i = \{0, \ldots, k\} \times \{0, \ldots, |y_i|\}$ is the set of states,
- $q_i = (0, 0)$ is the initial state,
- for $a \in \{0, \ldots, k\}$, $b \in \{0, \ldots, |y_i|\}$,
  - $\delta((a, b), \#) = (a + 1, b + |u_{a+1}|)$ if $a < k$ and $y_i[b, b + |u_{a+1}|] = u_{a+1}$,
  - $\delta((a, b), \sigma) = (a, b + 1)$ if the $(b + 1)$th letter of $|y_i|$ is σ.

We now prove that the intersection $A_1 \cap \cdots \cap A_n$ can be represented by an automaton

which has as many states as the smallest A_i. We first compute the intersection between A_1 and A_2, and show that the resulting automaton can be obtained from A_1 by deleting transitions (see Figure 5.3).

We denote the states of A_1 by $P = \{p_{(i,j)} \mid i \in \{0, \ldots, k\}, j \in \{0, \ldots, |w_1|\}\}$ and the states of A_2 by $Q = \{q_{(i,j)} \mid i \in \{0, \ldots, k\}, j \in \{0, \ldots, |w_2|\}\}$.

Let $y_1 = X_0 u_1 X_1 \cdots u_k X_k$, and $y_2 = X_0 v_1 X_1 \cdots v_k X_k$, where $u_1, \ldots, u_k, v_1, \ldots, v_k \in \Gamma^*$

We know that whenever there is a transition from a state $p_{(i,j)}$ to $p_{(i',j')}$ in A_1 then either:

- $i' = i$ and $j' = j + 1$ (Γ-transitions), or
- $i' = i + 1$ and $j' = j + |u_{i+1}|$ (#-transitions)

The same property holds for A_2, by replacing u_{i+1} with v_{i+1}.

We compute the cartesian product B_2 of A_1 and A_2. The states of B_2 are pairs from $P \times Q$. Consider such a state $(p_{(i,j)}, q_{(i',j')})$ which is reachable in B_2 from the initial state $(p_{(0,0)}, q_{(0,0)})$.

First, we can show that $i = i'$. The only transitions that increase i and i' in A_1 and A_2 are #-transitions, and they increase i and i' by 1. Thus, in the cartesian product B_2, we always have $i = i'$.

Similarly, the following invariant holds for the reachable states $(p_{(i,j)}, q_{(i,j')})$ of B_2:

$$j - j' = \sum_{k=1}^{i} |u_k| - |v_k|$$

In particular, this means that each state $p \in P$ can be paired with at most one state $q \in Q$ in B_2 (and each state of Q can be paired with at most one state of P). This entails that B_2 can be obtained from A_1 or A_2 by erasing transition, and that it has at most as many reachable states as $min(|P|, |Q|)$.

For $3 \leq i \leq n$, we then compute $B_i = A_i \cap B_{i-1}$ similarly, and obtain that $B_n = A_1 \cap \cdots \cap A_n$ has at most as many reachable states as the smallest A_i. □

5.6.3 Algorithm for Learning from a Sample

Consider a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree. Given $(t, w) \in \mathcal{S}$, we define the word equation $\mathsf{equation}(t, w)$ as:

$$\tau_\Sigma(t) = w$$

Algorithm 1 Learning 1STSs from a sample.

Input: A sample $\mathcal{S}$ whose domain is closed under subtree.
Output: If there exists a 1STS τ such that $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$, output Yes and τ, otherwise, output No.

1. Build the sequential formula $\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})$
2. Check whether φ has a satisfying assignment μ as follows: (see Lemma 5.6.2.1):
   - For every word equation $\mathsf{regEquation}(t, w, \mathcal{S})$ where t has root f, build a DFA that represents all possible solutions for the words $\mu(f, 0),\ldots,\mu(f, k)$.
   - Check whether the intersection of all DFAs contains some word w.
     - If no, exit the algorithm and return No.
     - If yes, define the words $\mu(f, 0),\ldots,\mu(f, k)$ following w.
3. Return (Yes and) $\mathtt{sts}(\mu)$.

where the left hand side $\tau_\Sigma(t)$ is a concatenation of elements from $\overline{\Sigma}$, considered as word variables, and the right hand side $w \in \Gamma^*$ is considered to be a word constant.

Assume all equations corresponding to a set of input/output examples are simultaneously satisfiable, with an assignment $\mu : \overline{\Sigma} \to \Gamma^*$. Our algorithm then returns the 1STS $\tau = \mathtt{sts}(\mu)$, thus guarantying that $\tau(t) = w$ for all $(t, w) \in \Sigma$.

If the equations are not simultaneously satisfiable, our algorithm returns No.

Example 8. For $\Sigma = \{\mathrm{nil}^{(0)}, \mathrm{cons}^{(2)}, \mathrm{node}^{(2)}, \mathrm{div}^{(0)}, \mathrm{pre}^{(0)}, \mathrm{span}^{(0)}\}$, given the examples:

$$\begin{aligned}
\tau_\Sigma(\mathrm{node}(\mathrm{div}, \mathrm{nil})) &= \text{“<.div”} \\
\tau_\Sigma(\mathrm{div}) = \text{“div”} \qquad \tau_\Sigma(\mathrm{span}) &= \text{“span”} \qquad \tau_\Sigma(\mathrm{pre}) = \text{“pre”} \\
\tau_\Sigma(\mathrm{cons}(\mathrm{node}(\mathrm{div}, \mathrm{nil}), \mathrm{nil}) = \text{“(<.div)”} \qquad \tau_\Sigma(\mathrm{nil}) &= \text{“”}
\end{aligned}$$

we obtain the following equations:

$$\begin{aligned}
(\mathrm{node}, 0) \cdot (\mathrm{div}, 0) \cdot (\mathrm{node}, 1) \cdot (\mathrm{nil}, 0) \cdot (\mathrm{node}, 2) &= \text{“<.div”} \\
(\mathrm{div}, 0) &= \text{“div”} \\
(\mathrm{span}, 0) &= \text{“span”} \\
(\mathrm{pre}, 0) &= \text{“pre”} \\
(\mathrm{cons}, 0) \cdot (\mathrm{node}, 0) \cdot (\mathrm{div}, 0) \cdot (\mathrm{node}, 1) \cdot (\mathrm{nil}, 0) \cdot & \\
(\mathrm{node}, 2) \cdot (\mathrm{cons}, 1) \cdot (\mathrm{nil}, 0) \cdot (\mathrm{cons}, 2) &= \text{“(<.div)”} \\
(\mathrm{nil}, 0) &= \text{“”}
\end{aligned}$$

A satisfying assignment for these equations is the morphism $\mathtt{morph}[\tau]$ given in Example 4.

Note that this assignment is not unique (see Example 9). We resolve ambiguities in Section 5.7.

To check for satisfiability of $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$, we slightly transform the equations in order to obtain a sequential formula. For $(t,w) \in \mathcal{S}$, with $t = f(t_1,\ldots,t_k)$, we define the word equation $\mathsf{regEquation}(t,w,\mathcal{S})$ as:

$$(f,0)\, w_1\, (f,1) \cdots w_k\, (f,k) = w$$

where for all $i \in \{1,\ldots,k\}$, $w_i = \mathcal{S}(t_i)$. Note that $\mathcal{S}(t_i)$ must be defined, since t is in the domain of $\mathcal{S}$, which is closed under subtree. Moreover, the formula

$$\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t,w,\mathcal{S})$$

is satisfiable iff $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$ is satisfiable.

Finally, φ is a sequential formula. Indeed, two equations corresponding to trees having the same root $f^{(k)} \in \Sigma$ have the same sequence of variables $(f,0)\ldots(f,k)$ in their left hand sides. And two equations corresponding to trees not having the same root have disjoint variables. Thus, using Lemma 5.6.2.1, we can check satisfiability of φ in polynomial time (and obtain a satisfying assignment for φ if there exists one).

Theorem 5.6.3.1 (Correctness and running time of Algorithm 1)**.** *Let $\mathcal{S}$ be a sample whose domain is closed under subtree. If there exists a* 1STS *τ such that $\tau(t) = w$ for all $(t,w) \in \mathcal{S}$, Algorithm 1 returns one such* 1STS*. Otherwise, Algorithm 1 returns* No*. Algorithm 1 terminates in time polynomial in the size of $\mathcal{S}$.*

Proof. Assume φ has a satisfying assignment $\mu : \overline{\Sigma} \to \Gamma^*$, in step (2) of Algorithm 1. In that case, Algorithm 1 returns $\tau = \mathtt{sts}(\mu)$. By definition of φ, we know, for all $(t,w) \in \mathcal{S}$, $\mu(\tau_\Sigma(t)) = w$. Moreover, since $\mathtt{morph}[\tau] = \mu$, we have by Lemma 5.5.1.2 that $\tau(t) = \mu(\tau_\Sigma(t))$, so $\tau(t) = w$.

Conversely, if there exists τ such that $\tau(t) = w$ for all $(t,w) \in \mathcal{S}$. Then, again by Lemma 5.5.1.2, $\mathtt{morph}[\tau]$ is a satisfying assignment for φ, and Algorithm 1 must return Yes.

The polynomial running time follows from Lemma 5.6.2.1.

Remark. For samples whose domains are not closed under subtree, we may modify Algorithm 1 to check for satisfiability of word equations which are not necessarily sequential. In that case, we are not guaranteed that the running time is polynomial.

□

5.7 Learning 1STSs Without Ambiguity

The issue with Algorithm 1 is that the 1STS expected by the user may be different than the one returned by the algorithm (see Example 9 below). To circumvent this issue, we use the notion of *tree test set*. Formally, a set of trees $T \subseteq D$ is a *tree test set for the domain* D if for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T}$ implies $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$.

Example 9. The transducer τ_2 defined below satisfies the requirements of Example 8 but is different than the transducer in Example 2. Namely, the values in the box have been switched.

$$\begin{aligned}
\delta_2(\text{node}) &= (\text{“<.”}, \varepsilon, \varepsilon) \\
\delta_2(\text{div}) &= (\text{“div”}) \qquad & \delta_2(\text{pre}) &= (\text{“pre”}) \qquad & \delta_2(\text{span}) &= (\text{“span”}) \\
\delta_2(\text{cons}) &= (\text{“(”}, \boxed{\varepsilon, \text{“)”}}) \qquad & \delta_2(\text{nil}) &= (\varepsilon)
\end{aligned}$$

We can verify that the two transducers are not equal on the domain D_{html}:

$$\begin{aligned}
\tau(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) &= \text{“(<.div)(<.div)”} \\
\tau_2(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) &= \text{“(<.div(<.div))”}
\end{aligned}$$

Therefore, if a user had the 1STS τ in mind when giving the sample of Example 8, it is still possible that Algorithm 1 returns τ_2. However, by definition of *tree test set*, if the sample given to Algorithm 1 contains a tree test set for D_{html}, we are guaranteed that the resulting transducer is equivalent to the transducer that the user has in mind, for all trees on D_{html}.

Our goal in this section is to compute from a given domain D a tree test set for D. The notion of tree test set is derived from the well-known notion of *test set* in formal languages. The *test set* of a language L (a set of words) is a subset $T \subseteq L$ such that for any two morphisms $f, g : \Sigma^* \to \Gamma^*$, $f_{|T} = g_{|T}$ implies $f_{|L} = g_{|L}$.

To compute a tree test set T for D, we first compute a test set T_G for the context-free language $G = \{\tau_\Sigma(t) \mid t \in D\}$ (built in Lemma 5.5.1.3), and then define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. We prove in Lemma 5.7.2.1 that T is indeed a tree test set for D.

We introduce in Section 5.7.1 a new construction, asymptotically optimal, for building test sets of context-free languages. We show in Section 5.7.2 how this translates to a construction of a tree test set for a domain D. We also give a sufficient condition of D so that the obtained tree test set is closed under subtree. This allows us to present, in Section 5.7.3, an algorithm that learns 1STSs from a domain D in polynomial-time (by building the tree test set T of D, and asking to the user the outputs corresponding to the trees of T).

5.7.1 Test Sets for Context-Free Languages

We show in this section how to build, from a context-free grammar G, a test set of size of $O(|G|^3)$. Our construction is asymptotically optimal. We reuse lemmas from [Plandowski, 1994, Plandowski, 1995], which were originally used to give a $O(|G|^6)$ construction.

Plandowski's Test Set

The following lemma was originally used in [Plandowski, 1994, Plandowski, 1995] to show that any linear context-free grammar has a test set containing at most $O(|R|^6)$ elements. We show in Section 5.7.1 how this lemma can be used to show a $2|R|^3$ bound.

Let $\Sigma_4 = \{a_i, \overline{a_i}, b_i, \overline{b_i} \mid i \in \{1,2,3,4\}\}$ be an alphabet. We define:

$$L_4 = \{x_4\, x_3\, x_2\, x_1\, \overline{x_1}\, \overline{x_2}\, \overline{x_3}\, \overline{x_4} \mid \forall i \in \{1,2,3,4\}.\ (x_i, \overline{x_i}) = (a_i, \overline{a_i}) \vee (x_i, \overline{x_i}) = (b_i, \overline{b_i})\}$$

and $T_4 = L_4 \setminus \{b_4\, b_3\, b_2\, b_1\, \overline{b_1}\, \overline{b_2}\, \overline{b_3}\, \overline{b_4}\}$.

The sets $L_4, T_4 \subseteq \Sigma_4$ have 16 and 15 elements respectively.

Lemma 5.7.1.1 ([Plandowski, 1994, Plandowski, 1995])**.** *T_4 is a test set for L_4.*

Linear Context-Free Grammars

We now prove that for any linear context-free grammar G, there exists a test set whose size is $2|R|^3$. Like the original proof of [Plandowski, 1994, Plandowski, 1995] that gave a $O(|R|^6)$ upper bound, our proof relies on Lemma 5.7.1.1. However, our proof uses a different construction to obtain the new, tight, bound.

Theorem 5.7.1.1. *Let $G = (N, \Sigma, R, S)$ be a linear context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|R|^3$ elements.*

Proof. Before building the test set, we introduce some notation.

Graph of G. Define the labeled graph $\mathsf{graph}(G) = (V, E)$ where $V = N \cup \{\bot\}$, $\bot$ is a new symbol, and $E \subseteq V \times R \times V$ such that:

- for non-terminals $A, B \in N$ and a rule $r \in R$, let $(A, r, B) \in E$ iff r is of the form $A \to uBv$ where $u, v \in \Sigma^*$ (i.e., B is the only non-terminal occurring in *rhs*).
- for a non-terminal $A \in N$ and $r \in R$, $(A, r, \bot) \in E$ if and only if $r = A \to rhs$ for some $rhs \in \Sigma^*$.

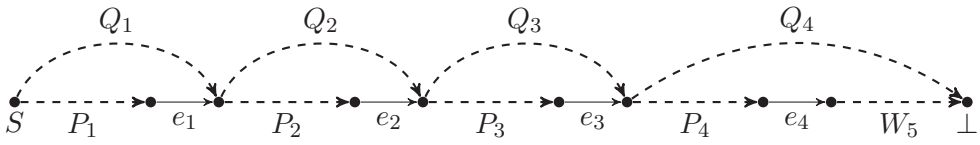


Figure 5.4 – The four optimal subpaths Q_1, Q_2, Q_3, and Q_4 define 15 alternative paths from S to $\perp$ which are all strictly smaller (with respect to order $<$) than $P_1e_1P_2e_2P_3e_3P_4e_4W_5$.

A *path* of $\mathsf{graph}(G)$ is a (possibly cyclic) sequence of edges of E, of the form: $(A_1, r_1, A_2) \cdot (A_2, r_2, A_3) \cdots (A_n, r_n, A_{n+1})$. A path is *accepting* if $A_1 = S$ and $A_{n+1} = \perp$.

Link between $\mathsf{graph}(G)$ and G. Given a rule $A \to uBv \in R$, where $A, B \in N$ and $u, v \in \Sigma^*$, we denote $\pi(r) = u$ and $\overline{\pi}(r) = v$. For a rule of the form $A \to u$ where $u \in \Sigma^*$ we denote $\pi(r) = u$ and $\overline{\pi}(r) = \varepsilon$. For a path $P = (A_1, r_1, A_2) \cdot (A_2, r_2, A_3) \cdots (A_n, r_n, A_{n+1})$ we define $\pi(P) = \pi(r_1) \cdots \pi(r_n)$, and $\overline{\pi}(P) = \overline{\pi}(r_n) \cdots \overline{\pi}(r_1)$.

Each accepting path P in $\mathsf{graph}(G)$ corresponds to a word $\pi(P) \cdot \overline{\pi}(P)$ in G, and conversely, for any word $w \in G$, there exists an accepting path (not necessarily unique) in $\mathsf{graph}(G)$ corresponding to w.

Total order on paths. We fix an arbitrary total order $<$ on R, and extend it to sequence of edges in R^* as follows. Given paths $P_1, P_2 \in R^*$, we have $P_1 < P_2$ iff

- $|P_1| < |P_2|$ (length of P_1 is smaller than length of P_2), or
- $|P_1| = |P_2|$ and P_1 is smaller lexicographically than P_2.

A path P is called *optimal* if it is the minimal path from the first vertex of P to the last vertex of P.

Test set for G. Let $\Phi_k(G)$ be the set of words of G corresponding to accepting paths of the form $P_1e_1P_2 \cdots P_ne_nP_{n+1}$, $n \leq k$, with $P_i \in R^*$, $e_i \in R$, and where for $i \in \{1, \ldots, n+1\}$, P_i is optimal, and for $i \in \{1, \ldots, n\}$, P_ie_i is not optimal. By construction, a path in $\Phi_k(G)$ is uniquely determined (when it exists) by the choice of edges $e_1, \ldots, e_n$, as optimal paths between two vertices are unique. Therefore, $\Phi_k(G)$ contains at most $\sum_{i=0}^{k} |R|^i \leq 2|R|^k$ words.

We now show that $\Phi_3(G)$ is a test set for G (which gives us the desired bound of the theorem: $2|R|^k$). Assume there exist two morphisms $f, g : \Sigma^* \to \Gamma^*$ such that $f_{|\Phi_3(G)} = g_{|\Phi_3(G)}$ and there exists $w \in G$ such that $f(w) \neq g(w)$.

By assumption, w does not belong to $\Phi_3(G)$, and must correspond to a path $P = P_1e_1P_2 \cdots P_ne_nP_{n+1}$ for $n \geq 4$, such that for $i \in \{1, \ldots, n+1\}$, P_i is optimal, and P_ie_i is not optimal. We pick w having the property $f(w) \neq g(w)$ such that the path P is the smallest possible (according to the order $<$ defined above).

The path P can be written $P_1e_1P_2e_2P_3e_3P_4e_4W_5$ where for $i \in \{1, 2, 3, 4\}$, P_i is optimal, and P_ie_i is not optimal (W_5 is not necessarily optimal). For $i \in \{1, 2, 3\}$, we define Q_i to be the optimal path from the source of P_ie_i to its target; hence $Q_i < P_ie_i$. Moreover, Q_4 is defined to be the optimal path from the source of $P_4e_4W_5$ to its target, with $Q_4 < P_4e_4W_5$. Effectively, as shown in Figure 5.4, this defines 15 paths that can be derived from P by replacing subpaths by their corresponding optimal path (Q_1, Q_2, Q_3, Q_4).

Let P' be one of those 15 paths (where at least one subpath has been replaced by its optimal counterpart Q_1, Q_2, Q_3, or Q_4), and let $w' \in G$ be the word corresponding to P'. By construction of P', and by definition of the order $<$, we have $P' < P$. Since we have chosen P to be the smallest possible path such that f and g are not equal on the corresponding word, we deduce that $f(w') = g(w')$.

To conclude, we show that we obtain a contradiction, thanks to Lemma 5.7.1.1. For this, we construct two morphisms $f', g' : \Sigma_4 \to \Gamma$ as follows (i ranges over $\{1, 2, 3, 4\}$ and j over $\{1, 2, 3\}$):

- $f'(a_i) = f(\pi(Q_i))$,
- $f'(\overline{a_i}) = f(\overline{\pi}(Q_i))$,
- $f'(b_j) = f(\pi(P_je_j))$,
- $f'(\overline{b_j}) = f(\overline{\pi}(P_je_j))$.
- $f'(b_4) = f(\pi(P_4e_4W_5))$,
- $f'(\overline{b_4}) = f(\overline{\pi}(P_4e_4W_5))$.

The morphism g' is defined similarly, using g instead of f. We can then verify that f' and g' coincide on T_4, but are not equal on the word $b_4\, b_3\, b_2\, b_1\, \overline{b_1}\, \overline{b_2}\, \overline{b_3}\, \overline{b_4} \in L_4$, thus contradicting Lemma 5.7.1.1. □

We make use of this theorem in the next section to obtain test sets for context-free grammars which are not necessarily linear.

Context-Free Grammars

To obtain a test set for a context-free grammar G which is not necessarily linear, [Plandowski, 1994] constructs from G a linear context-free grammar, $\mathrm{Lin}(G)$, which produces a subset of G, and which is a test set for G.

Formally, $\mathrm{Lin}(G)$ is derived from G as follows:

- For every productive non-terminal symbol A in G, choose a word x_A produced by A.
- Every rule $r : A \to x_0 A_1 x_1 \dots A_n x_n$ in G, where for every i, $x_i \in \Sigma^*$ and $A_i \in N$ is productive, is replaced by n different rules, each one obtained from r by replacing all A_i with x_{A_i}, except one.

Note that the definition of $\mathrm{Lin}(G)$ is not unique, and depends on the choice of the words x_A. The following result holds for any choice of the words x_A.

Lemma 5.7.1.2 ([Plandowski, 1994, Plandowski, 1995])**.** *Lin(G) is a test set for G.*

Using Theorem 5.7.1.1, we improve the $O(|G|^6)$ bound of [Plandowski, 1994, Plandowski, 1995] for the test set of G to $2|G|^3$.

Theorem 5.7.1.2. *Let $G = (N, \Sigma, R, S)$ be a context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|G|^3$ elements.*

Proof. Follows from Theorem 5.7.1.1, Lemma 5.7.1.2, and from the fact that $\mathrm{Lin}(G)$ has at most $|G| = \sum_{A \to rhs \in R}(|rhs| + 1)$ rules. (When constructing $\mathrm{Lin}(G)$, each rule $A \to rhs$ of G is duplicated at most $|rhs|$ times.) □

5.7.2 Tree Test Sets for Transducers

We use the results of the previous section to construct a tree test set for a domain D.

Lemma 5.7.2.1. *Any domain $D = (\Sigma, Q, I, \delta)$ has a tree test set T of size at most $O(|D|)^3$. Moreover, if $I = Q$, then we can build T such that T is closed under subtree.*

Proof. Intuitively, we build the tree test set for D by taking the set of trees corresponding to the test set of G, where G is the grammar built in Lemma 5.5.1.3.

Let τ_1 and τ_2 be two 1STSs. Let T_G be a test set for G. Define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. By Theorem 5.7.1.2, we can assume T_G has size at most $|G|^3$, and hence, T has size at

most $|D|^3$. Let μ_1 and μ_2 be $\texttt{morph}[\tau_1]$ and $\texttt{morph}[\tau_2]$, respectively. We have:

$$
\begin{aligned}
&[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T} \iff \\
&\forall t \in T.\ \tau_1(t) = \tau_2(t) \iff \\
&\forall w \in T_G.\ \tau_1(\mathsf{tree}(w)) = \tau_2(\mathsf{tree}(w)) \iff \text{(by Lemma 5.5.1.2)} \\
&\forall w \in T_G.\ \mu_1(\tau_\Sigma(\mathsf{tree}(w))) = \mu_2(\tau_\Sigma(\mathsf{tree}(w))) \iff \text{(by definition of } \mathsf{tree}) \\
&\forall w \in T_G.\ \mu_1(w) = \mu_2(w) \iff \text{(since } T_G \text{ is a test set for } G) \\
&\forall w \in G.\ \mu_1(w) = \mu_2(w) \iff \text{(see Lemma 5.5.1.3)} \\
&[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}
\end{aligned}
$$

This ends the proof that T is a tree test set for D.

We now show how to construct T such that it is closed under subtree. For every non-terminal A of G, we define the minimal word w_A. These words are built inductively, starting from the non-terminals which have a rule whose right-hand-side is only made of terminals. In the definition of $\mathrm{Lin}(G)$, we use these words when modifying the rules of G into linear rules.

When then define T_G as the test set of $\mathrm{Lin}(G)$ (which is also a test set of G), and $T = \{\mathsf{tree}(w) \mid w \in T_G\} \cup \{\mathsf{tree}(w_A) \mid A \in G\}$. As shown previously, T is a tree test set for D. We can now prove that T is closed under subtree. Let $t = f(t_1, \ldots, t_k) \in T$. Let $i \in \{1, \ldots, k\}$. We want to prove that $t_i \in T$.

We consider two cases. Either there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$, or there exists $A \in G$, $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w_A)$.

- First, if there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$. Consider a derivation p for w in the $\mathrm{Lin}(G)$. By construction of $\mathrm{Lin}(G)$, the first rule is an ε-transition of the form $S \to N$ while the second rule is of the form:

  $$N \to (f,0) \cdot w_1 \cdot (f,1) \cdots w_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f,j) \cdot w_{j+1} \cdots w_k \cdot (f,k).$$

  This second rule corresponds to a rule in G, of the form:

  $$N \to (f,0) \cdot N_1 \cdot (f,1) \cdots N_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f,j) \cdot N_{j+1} \cdots N_k \cdot (f,k).$$

  We then have two subcases to consider. Either $i \neq j$, and in that case $t_i = \mathsf{tree}(w_i)$. By construction of $\mathrm{Lin}(G)$, w_i must be equal to w_A for some $A \in G$. Thus, we have $t_i \in T$ by definition of T.

  Or $i = j$, in that case $t_i = \mathsf{tree}(w')$, where w' is derived by the derivation p where the first two derivation rules, outlined above, are replaced with the ε-rule $S \to N_i$.

This production rule is ensured to exist in $\mathrm{Lin}(G)$, as all states of D are initial, so there exists a rule $S \to N_q$ for all $q \in Q$. (see definition of G in Lemma 5.5.1.3). Then, since $w \in \Phi_3(\mathrm{Lin}(G))$, and by construction of $\Phi_3(\mathrm{Lin}(G))$, we conclude that $w' \in \Phi_3(\mathrm{Lin}(G))$. This ensures that $w' \in T_G$, and $t_i \in T$.

- Otherwise, there exists $A \in G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w_A)$. Using the fact that w_A was build inductively in the grammar G, using other minimal words $w_{A'}$ for $A' \in G$, we deduce there exists $A' \in G$ such that $t_i = \mathsf{tree}(w_{A'})$, and $t_i \in T$.

□

Lemma 5.7.2.2 shows the bound given in Lemma 5.7.2.1 is tight, in the sense that there exists an infinite class of growing domains D for which the smallest tree test set has size $|D|^3$.

Lemma 5.7.2.2. *There exists a sequence of domains $D_1, D_2, \ldots$ such that for every $n \geq 1$, the smallest tree test set of D_n has at least n^3 elements, and the size of D_n is linear in n. Furthermore, this lower bound holds even with the extra assumption that all states of the domain are initial.*

Proof. (Sketch) Our proof is inspired by the lower bound proof for test sets of context-free languages [Plandowski, 1994, Plandowski, 1995]. For $n \geq 1$, we build a particular domain D_n (whose states are all initial), and we assume by contradiction that it has a test set T of size less than n^3. From this assumption, we expose a tree $t \in D_n$, as well as two 1STSs τ_1 and τ_2 such that $\tau_{1|T} = \tau_{2|T}$ but $\tau_1(t) \neq \tau_2(t)$.

The complete example for the lower bound is located in Appendix E.3 □

5.7.3 Learning 1STSs Without Ambiguity

Algorithm 2 Learning 1STSs from a domain.

Input: A domain D, and an oracle 1STS τ_u.
Output: A 1STS τ functionally equivalent to τ_u.

1. Build a tree test set $\{t_1 \ldots t_n\}$ of D, following Lemma 5.7.2.1.
2. For every $t_i \in \{t_1 \ldots t_n\}$, ask the oracle for $w_i = \tau_u(t_i)$.
3. Run Algorithm 1 on the sample $\{(t_i, w_i) \mid 1 \leq i \leq n\}$.

Our second algorithm (see Algorithm 2) takes as input a domain D, and computes a tree test set $T \subseteq D$. It then asks the user the expected output for each tree $t \in T$. The user is modelled by a 1STS τ_u that can be used as an oracle in the algorithm. Algorithm 2 then

runs Algorithm 1 on the obtained sample. The 1STS τ_u expected by the user may still be syntactically different the 1STS τ returned by our algorithm, but we are guaranteed that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$ (by definition of tree test set).

Theorem 5.7.3.1 (Correctness and running time of Algorithm 2)**.** *Let τ_u be a* 1STS *(used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 2 is a* 1STS *τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.*

Furthermore, Algorithm 2 invokes the oracle $O(|D|^3)$ times, and terminates in time polynomial in $|D|$.

Proof. The correctness of Algorithm 2 follows from the correctness of Algorithm 1 and from the fact that T is a tree test set for D. The fact that Algorithm 2 invokes the algorithm $O(|D|^3)$ times follows from the size of the tree test set (see Lemma 5.7.2.1).

Moreover, since all states of D are initial, the tree test set of D that we build is closed under subtree. The polynomial running time then follows from the fact that Algorithm 1 ends in polynomial time for samples whose domains are closed under subtree.

Remark. Similarly to Algorithm 1, Algorithm 2 also applies for domains such that $I \neq Q$, but the running time is not guaranteed to be polynomial.

□

5.8 Learning 1STS Interactively

Our third algorithm (see Algorithm 3) takes as input a domain D, and computes a tree test set $T \subseteq D$. For this algorithm, we require from the beginning that all states of D are initial, so that T is closed under subtree. For a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree, and for $(t, w) \in \mathcal{S}$, we denote by $\mathsf{automaton}(t, w)$ the automaton $\mathsf{automaton}(y, w)$ where $y = w$ is the equation $\mathsf{regEquation}(t, w, \mathcal{S})$.

Instead of building the sample $\mathcal{S}$ and the intersection $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$ all at once, like algorithms 1 and 2 do, Algorithm 3 builds $\mathcal{S}$ and the intersection incrementally. It then uses the intermediary results to infer outputs, in order to avoid calling the oracle τ_u too many times. Overall, we prove that Algorithm 3 invokes the oracle τ_u at most $O(|D|)$ times, while Algorithm 2 invokes it $O(|D|^3)$ times.

To infer outputs, Algorithm 3 maintains the following invariant for the while loop. First $\mathcal{S}$ is such that $dom(\mathcal{S}) \subseteq T$, and its domain increases at each iteration. Then, for any $f^{(k)} \in \Sigma$, $\mathtt{sol}(f)$ is equal to $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$, and thus recognizes the set

$$\{\mu(f,0)\#\mu(f,1)\#\ldots\#\mu(f,k) \mid \mu : \overline{\Sigma} \to \Gamma \text{ satisfies } \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})\}.$$

Algorithm 3 Interactive learning of 1STSs.

Input: A domain D, and an oracle 1STS τ_u whose output alphabet is Γ.
Output: A 1STS τ functionally equivalent to τ_u.

1. Initialize a map $\texttt{sol}$ from Σ to Automata, such that for $f^{(k)} \in \Sigma$, $\texttt{sol}(f)$ recognizes $\{x_0\# \cdots \#x_k \mid x_i \in \Gamma^*\}$,
2. Build a tree test set T of D, following Lemma 5.7.2.1.
3. Initialize a partial function $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$, initially undefined everywhere.
4. While $dom(\mathcal{S}) \neq T$:
   - Choose a tree $f(t_1, \ldots, t_k) \notin dom(\mathcal{S})$ such that all subtrees of t belong to $dom(\mathcal{S})$ (possible since T is closed under subtree).
   - Build the automaton A recognizing $\{x_0\, \mathcal{S}(t_1)\, x_1 \cdots \mathcal{S}(t_k)\, x_k \mid x_0\#x_1 \cdots \#x_k \in \texttt{sol}(f)\}$, representing all possibles values of $\tau_u(t)$ that do not contradict previous outputs.
     - If A recognizes only 1 word w, define $\mathcal{S}(t) = w$.
     - Otherwise (A recognizes at least 2 words), define $\mathcal{S}(t) = \tau_u(t)$ using the oracle.
   - Update $\texttt{sol}(f) = \texttt{sol}(f) \cap \mathsf{automaton}(t, \mathcal{S}(t))$.
5. Run Algorithm 1 on $\mathcal{S}$.

Intuitively, $\texttt{sol}(f)$ represents the possible values for the output of f in the transducer τ_u, based on the constraints given so far.

To infer the output of a tree $t = f(t_1, \ldots, t_k)$, for some $f^{(k)} \in \Sigma$, Algorithm 3 uses the fact that $\tau_u(f(t_1, \ldots, t_k))$ must be of the form $\mu(f,0)\mathcal{S}(t_1)\mu(f,1)\cdots\mathcal{S}(t_k)\mu(f,k)$ for some morphism $\mu : \overline{\Sigma} \to \Gamma$ satisfying $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$. By construction, the NFA A, that recognizes the set $\{x_0\,\mathcal{S}(t_1)\,x_1\cdots\mathcal{S}(t_k)\,x_k \mid x_0\#x_1\cdots\#x_k \in \texttt{sol}(f)\}$, recognizes exactly these words of the form $\mu(f,0)\mathcal{S}(t_1)\mu(f,1)\cdots\mathcal{S}(t_k)\mu(f,k)$.

We then check whether A recognizes exactly one word w, in which case, we know $\tau_u(t) = w$, and we do not need to invoke the oracle. Otherwise, there are several alternatives which are consistent with the previous outputs provided by the user, and we cannot infer $\tau_u(t)$. We thus invoke the oracle (the user) to obtain $\tau_u(t)$.

Before proving the theorem corresponding to Algorithm 3, we give a lemma on words which we use extensively in the theorem.

Lemma 5.8.1.1. *Let $u, v, w \in \Gamma^*$. If $uv = vu$ and $uw = wu$ and $u \neq \varepsilon$, then $vw = wv$.*

Proof. A word $p \in \Gamma^*$ is *primitive* if there does not exist $r \in \Gamma^*$, $i > 1$ such that $p = r^i$. Proposition 1.3.2 of [Lothaire, 1997] states that the set of words commuting with a non-empty word u is a monoid generated by a single primitive word p. Since v and w both commute with u, there exist i and j such that $v = p^i$ and $w = p^j$, thus $vw = wv = p^{i+j}$. □

The difficult part of Theorem 5.8.1.1 is to show the number of times the oracle τ_u is invoked is $O(|D|)$. We prove this by assuming by contradiction that the number of times τ_u is invoked is strictly greater than $3|D| + |Q|$ times. We prove this entails there are four trees which are nearly identical and for which our algorithm invokes the oracle (the four trees have the same root, and differ only for one child). Then, by a close analysis of the word equations corresponding to these four terms, we obtain a contradiction by proving our algorithm must have been able to infer the output for at least one of those terms.

Theorem 5.8.1.1 (Correctness and running time of Algorithm 3)**.** *Let τ_u be a* 1STS *(used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 3 is a* 1STS *τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.*

Algorithm 3 ends in time polynomial in $|D|$ and the number of times it invokes the oracle τ_u is in $O(|D|)$.

Proof. The correctness and the polynomial running time of Algorithm 3 can be proved similarly to Algorithm 2.

Note that we can check whether the NFA A recognizes only one word using the following polynomial time procedure. First, check if there exists a word w recognized by A. If there is, pick a minimal word $w \in \mathsf{A}$, and compute the automaton $A \cap B$, where B recognizes all words different than w (the size of B is roughly $|w|$). If the automaton $A \cap B$ is empty, then A recognizes only w, otherwise A recognizes more than one word.

The crucial part of Algorithm 3 is that it invokes the oracle τ_u at most $O(|D|)$ times. More precisely, we show that Algorithm 3 invokes τ_u at most $|Q| + 3\sum_{(q,f^{(k)},(q_1,\ldots,q_k)\in\delta} 1 + k$ times, which is $|Q| + 3|D|$, and in $O(|D|)$.

Let $T_{inv} \subseteq T$ be the set of trees for which Algorithm 3 invokes the oracle. Let $T_{min} \subseteq T$ be the set of trees which are of the form $\mathsf{tree}(w_A)$ for some $A \in G$ (where w_A is the minimal word that can be produced from A, see Lemma 5.7.2.1). Note that, by construction of G, $|T_{min}| \leq |Q|$.

Consider the set $T'_{inv} = T_{inv} \setminus T_{min}$. Let $W'_{inv} = \{\tau_\Sigma(t) \mid t \in T'_{inv}\}$ (or equivalently, $T'_{inv} = \{\mathsf{tree}(w) \mid w \in W'_{inv}\}$). We want to prove that $|T'_{inv}| \leq 3|D|$, thus implying that $|T_{inv}| \leq 3|D| + |Q|$. Assume by contradiction $|T'_{inv}| > 3|D|$.

Remember that, by construction of T, we have $W'_{inv} \subseteq \mathrm{Lin}(G)$. For $w \in W_{inv}$, consider the first non-epsilon rule in some derivation of w in $\mathrm{Lin}(G)$. By construction, $\mathrm{Lin}(G)$ has at most $|D|$ rules.

Moreover, since W_{inv} contains strictly more than $3|D|$ words, W_{inv} must contain at least four words that share the same first non-epsilon rule in their derivation. Let w_a, w_b, w_c, w_d be four such words, and t_a, t_b, t_c, t_d their corresponding trees (with $t_l = \mathsf{tree}(w_l)$ for $l \in \{a, b, c, d\}$).

Without loss of generality, assume that Algorithm 3 invoked τ_u on the trees t_a, t_b, t_c, and t_d in that order. By construction of $\mathrm{Lin}(G)$, and from the fact that w_a, w_b, w_c, and w_d share the first non-epsilon rule, we know there exists $f^{(k)} \in \Sigma$, $i \in \{1, \ldots, k\}$, $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k \in T$, and $t_i^a, t_i^b, t_i^c, t_i^d \in T$ such that:

$$
\begin{aligned}
t_a &= f(t_1, \ldots, t_{i-1}, t_i^a, t_{i+1}, \ldots, t_k) \\
t_b &= f(t_1, \ldots, t_{i-1}, t_i^b, t_{i+1}, \ldots, t_k) \\
t_c &= f(t_1, \ldots, t_{i-1}, t_i^c, t_{i+1}, \ldots, t_k) \\
t_d &= f(t_1, \ldots, t_{i-1}, t_i^d, t_{i+1}, \ldots, t_k).
\end{aligned}
$$

Said otherwise t_a, t_b, t_c, and t_d only differ on their ith subtree. Let $x_i^l = \tau_u(t_i^l)$ and $x_l = \tau_u(t_l)$, for $l \in \{a, b, c, d\}$.

Given an assignment $\mu : \overline{\Sigma} \to \Gamma^*$, we define $u_\mu = \mu(f, 0) \cdot \tau_u(t_1) \cdot \mu(f, 1) \cdots \mu(f, i-1)$, and $v_\mu = \mu(f, i+1) \cdots \tau_u(t_k) \cdot \mu(f, k)$.

We consider several cases, all of them leading to a contradiction.

- There exist $x_i^k, x_i^l \in \{x_i^a, x_i^b, x_i^c\}$ such that x_1 is not a prefix of x_2 and x_2 is not a prefix of x_1. Assume without loss of generality $x_1 = x_i^a$ and $x_2 = x_i^b$. Since Algorithm 3 invoked the oracle τ_u on the tree t_d, there are two assignments $\mu, \mu' : \overline{\Sigma} \to \Gamma^*$ such that $u_\mu x_i^d v_\mu \in A$, $u_{\mu'} x_i^d v_{\mu'} \in A$, and $u_\mu x_i^d v_\mu \neq u_{\mu'} x_i^d v_{\mu'}$. Using the invariant of Algorithm 3 on trees t_a and t_b, we know:

  $$x_a = u_\mu x_i^a v_\mu = u_{\mu'} x_i^a v_{\mu'}$$
  $$x_b = u_\mu x_i^b v_\mu = u_{\mu'} x_i^b v_{\mu'}$$

  Finally, since x_i^a is not a prefix of x_i^b, and x_i^b is not a prefix of x_i^a, we have $\mathsf{lcp}(x_a, x_b) = u_\mu \,\mathsf{lcp}(x_i^a, x_i^b)$ and $\mathsf{lcp}(x_a, x_b) = u_{\mu'} \,\mathsf{lcp}(x_i^a, x_i^b)$. Thus, $u_\mu = u_{\mu'}$ and we can deduce $v_\mu = v_{\mu'}$. Thus, $u_\mu \, x_i^d \, v_\mu = u_{\mu'} \, x_i^d \, v_{\mu'}$, and we have a contradiction.

- Two elements of $\{x_i^a, x_i^b, x_i^c\}$ are equal. For instance if $x_i^a = x_i^b$, then Algorithm 3 could not have invoked the oracle τ_u on the tree t_b, as the only possible solution for x_b is x_a.

- If we are not in one of the previous cases, we know that x_i^a, x_i^b, x_i^c are of the form x_1, x_1x_2, $x_1x_2x_3$ (not necessarily in that order) for some $x_1, x_2, x_3 \in \Gamma^*$, and $x_2 \neq \varepsilon$ and $x_3 \neq \varepsilon$. Consider the case where $x_2x_3 \neq x_3x_2$.

  Since Algorithm 3 invoked τ_u on the tree t_d, there are two assignments $\mu, \mu' : \overline{\Sigma} \to \Gamma^*$ such that $u_\mu x_i^d v_\mu \in A$, $u_{\mu'} x_i^d v_{\mu'} \in A$, and $u_\mu x_i^d v_\mu \neq u_{\mu'} x_i^d v_{\mu'}$. Using the invariant of Algorithm 3, we know:

  $$u_\mu x_1 v_\mu = u_{\mu'} x_1 v_{\mu'}$$
  $$u_\mu x_1 x_2 v_\mu = u_{\mu'} x_1 x_2 v_{\mu'}$$
  $$u_\mu x_1 x_2 x_3 v_\mu = u_{\mu'} x_1 x_2 x_3 v_{\mu'}$$

  Without loss of generality, assume that $u_{\mu'}$ is a prefix of u_μ and v_μ is a suffix of $v_{\mu'}$. So $u_\mu = u_{\mu'} u''$ and $v_{\mu'} = v'' v_\mu$ for some $u'', v'' \in \Gamma^*$. Then, we have $u'' \, x_1 = x_1 \, v''$, $u'' \, x_1x_2 = x_1x_2 \, v''$, and $u'' \, x_1x_2x_3 = x_1x_2x_3 \, v''$. We deduce, $x_1 \, v'' x_2 = x_1x_2 \, v''$, and v'' commutes with x_2.

  Similarly, v'' commutes with x_2x_3. Assume by contradiction that $v'' \neq \varepsilon$. Then, by Lemma 5.8.1.1, x_2 must commute with x_2x_3, and $x_2x_2x_3 = x_2x_3x_2$, which implies $x_2x_3 = x_3x_2$, and x_2 commutes with x_3, contradicting our assumption. This means that $v'' = \varepsilon$, $v_\mu = v_{\mu'}$, and $u_\mu = u_{\mu'}$. We thus conclude $u_\mu x_i^d v_\mu = u_{\mu'} x_i^d v_{\mu'}$, contradicting the fact that Algorithm 3 invoked τ_u on tree t_d.

- The last case is when the x_i^a, x_i^b, x_i^c are of the form x_1, x_1x_2, $x_1x_2x_3$ (not necessarily in that order) for some $x_1, x_2, x_3 \in \Gamma^*$, with $x_2 \neq \varepsilon$ and $x_3 \neq \varepsilon$, and $x_2x_3 = x_3x_2$.

Since Algorithm 3 invoked τ_u on the tree t_c, there are two assignments $\mu, \mu' : \overline{\Sigma} \to \Gamma^*$ such that $u_\mu x_i^c v_\mu \in A$, $u_{\mu'} x_i^c v_{\mu'} \in A$, and $u_\mu x_i^c v_\mu \neq u_{\mu'} x_i^c v_{\mu'}$, where A is the automaton constructed in Algorithm 3 at the iteration where $\tau_u(t_c)$ was invoked.

Without loss of generality, assume that $u_{\mu'}$ is a prefix of u_μ and v_μ is a suffix of $v_{\mu'}$. So $u_\mu = u_{\mu'} u''$ and $v_{\mu'} = v'' v_\mu$ for some $u'', v'' \in \Gamma^*$. We consider three subcases:

- Case $x_i^c = x_1 x_2 x_3$. Using the invariant of Algorithm 3 for trees t_a and t_b, we know:

$$u_\mu x_1 v_\mu = u_{\mu'} x_1 v_{\mu'}$$
$$u_\mu x_1 x_2 v_\mu = u_{\mu'} x_1 x_2 v_{\mu'}$$

  Then, we have $u''\, x_1 = x_1\, v''$, $u''\, x_1 x_2 = x_1 x_2\, v''$. We deduce, $x_1\, v'' x_2 = x_1 x_2\, v''$, and v'' commutes with x_2.

  Thus, x_2 commutes both with v'' and x_3. By Lemma 5.8.1.1, v'' commutes with x_3. We deduce that $x_1 x_2 x_3 v'' = x_1 v'' x_2 x_3 = u'' x_1 x_2 x_3$, and finally that $u_\mu x_1 x_2 x_3 v_\mu = u_{\mu'} x_1 x_2 x_3 v_{\mu'}$, contradicting $u_\mu x_i^c v_\mu \neq u_{\mu'} x_i^c v_{\mu'}$.

- Case $x_i^c = x_1 x_2$. Using the invariant of Algorithm 3 for the trees t_a and t_b, we know:

$$u_\mu x_1 v_\mu = u_{\mu'} x_1 v_{\mu'}$$
$$u_\mu x_1 x_2 x_3 v_\mu = u_{\mu'} x_1 x_2 x_3 v_{\mu'}$$

  Then, we have $u''\, x_1 = x_1\, v''$, $u''\, x_1 x_2 x_3 = x_1 x_2 x_3\, v''$. We deduce, $x_1\, v'' x_2 x_3 = x_1 x_2 x_3\, v''$, and v'' commutes with $x_2 x_3$.

  Since $x_2 x_3$ commutes both with v'' and x_2 (as x_2 and x_3 commute), we know by Lemma 5.8.1.1 that x_2 and v'' commute. We deduce that $x_1 x_2 v'' = x_1 v'' x_2 = u'' x_1 x_2$, and finally that $u_\mu x_1 x_2 v_\mu = u_{\mu'} x_1 x_2 v_{\mu'}$, contradicting $u_\mu x_i^c v_\mu \neq u_{\mu'} x_i^c v_{\mu'}$.

- Case $x_i^c = x_1$. Using the invariant of Algorithm 3 for the trees t_a and t_b, we know:

$$u_\mu x_1 x_2 v_\mu = u_{\mu'} x_1 x_2 v_{\mu'}$$
$$u_\mu x_1 x_2 x_3 v_\mu = u_{\mu'} x_1 x_2 x_3 v_{\mu'}$$

  Then, we have $u''\, x_1 x_2 = x_1 x_2\, v''$, $u''\, x_1 x_2 x_3 = x_1 x_2 x_3\, v''$. We deduce, $x_1 x_2\, v'' x_3 = x_1 x_2 x_3\, v''$, and v'' commutes with x_3.

  Thus, x_3 commutes both with v'' and x_2. By Lemma 5.8.1.1, v'' commutes with x_2. Moreover, since $u''\, x_1 x_2 = x_1 x_2\, v''$, we have $u''\, x_1 x_2 = x_1\, v'' x_2$, and $u''\, x_1 = x_1\, v''$. We conclude that $u_\mu x_1 v_\mu = u_{\mu'} x_1 v_{\mu'}$, contradicting $u_\mu x_i^c v_\mu \neq u_{\mu'} x_i^c v_{\mu'}$.

□

5.9 Tree with Values

Until now, we have considered a set of trees $\mathcal{T}_\Sigma$ which contained only other trees as subtrees, and with a test set of size $O(n^3)$, although we have a linear learning time if we have interactivity. However, in practice, data structures such as XML are usually trees containing *values*. Values are typically of type stringor int, and may be used instead of subtrees. For convenience, we will suppose that we only have string elements, and that string elements are rendered *raw*. We will demonstrate how we can directly obtain a test set of size $O(n)$.

Formally, let us add a special symbol $v \in \Sigma$, of arity 0, which has another version which can have a parameter. For each string $s \in \Gamma^*$ we can thus define the symbol v_s and extend the notion of trees and domains as follows.

For a set of trees $\mathcal{T}$, we define the extended set $\mathcal{T}'$ by:

$$\mathcal{T}' = \{t' \mid \exists t \in \mathcal{T}, t' \text{ is obtained from } t \text{ by replacing each } v \text{ by a } v_s \text{ for some } s \in \Gamma^*\}$$

Note that given a domain D and a height h, there is an infinite number of trees of height h in D', while only a finite number in D. Fortunately, thanks to the semantics of the transducers on v_s we define below, finding the tree test sets is easier in this setting.

For any transducer τ we extend the definition of $[\![\tau]\!]$ to $\mathcal{T}'_\Sigma$ by defining $[\![\tau]\!](v_s) = s$. We naturally extend the definition of tree test set of an extended domain D' to be a set $T' \subset D'$ such that for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T'} = [\![\tau_2]\!]_{|T'}$ implies $[\![\tau_1]\!]_{|D'} = [\![\tau_2]\!]_{|D'}$. After proving the following lemma, we will state and prove the theorem on linear test sets.

Lemma 5.9.1.1. *For $a, b, x, y \in \Gamma^*$, $c \neq d$ in Γ, if $acx = bcy$ and $adx = bdy$, then $a = b$.*

Proof. Either a or b is a prefix of the other. Let us suppose that $a = bk$ for some suffix $k \in \Gamma^*$. It follows that $kcx = cy$ and $kdx = dy$. If k is not empty, then k starts with c and with d, which is not possible. Hence k is empty and $a = b$. □

Theorem 5.9.1.1. *If the domain $D = (\Sigma, Q, I, \delta)$ is such that for every $f \in \Sigma$ of arity $k > 0$, there exist trees in $t_1, \ldots, t_k \in D$ such that $f(t_1, \ldots, t_k) \in D$ and each t_i contains at least one v, then there exists a tree test set of D' of linear size $O(|\Sigma| \cdot A)$ where A is the maximal arity of a symbol of Σ.*

Proof. (Sketch) Using the trees provided in the theorem's hypothesis, we build a linear

set of trees of D' where the v nodes are replaced successively by two different symbols $v_{\text{“\#”}}$ and $v_{\text{“?”}}$. Then, we prove that any two 1STSs which are equal on this set of trees, are syntactically equal.

The complete proof is located in Appendix E.5. □

5.10 Implementation

Our tool (walkthrough in Section 5.2) is open-source and available at https://github.com/epfl-lara/prosy. It takes as input an ADT represented by case class definitions written in a Scala-like syntax, and outputs a recursive printer for this ADT. For the automata constructions of Algorithm 3, we used the `brics` Java library[3].

In the walkthrough, notice that our tool gives propositions to the user so that the user does not have to enter the answers manually. The user may choose how many propositions are to be displayed (default is 9). To obtain these propositions, we use the following procedure. Remember that for each tree t for which we need to obtain the output, Algorithm 3 builds an NFA A that recognizes the set of all possible outputs for t (see Section 5.8). We check for the existence of an accepted word w_0 in A, and compute the intersection A_1 between A and an automaton recognizing all words except w_0. We then have two cases. Either A_1 is empty, and therefore we know the output for tree t is w_0. In that case, we do not need to interact with the user, and can continue on to the next tree. Otherwise, A_1 recognizes some word $w_1 \neq w_0$, which we display as a proposition to the user (alongside w_0). We then obtain A_2 as the intersection between A and an automaton recognizing all words except w_0 and w_1. We continue this procedure until we have 9 propositions (or whichever number the user entered), or when the intersected automaton becomes empty.

Concerning support for the String data type, we use ideas from Section 5.9 and reused our code from Algorithm 3 to infer outputs. Technically, we replace the String data type with an abstract class with two case classes, foo, and bar, that must be printed as “foo” and “bar” respectively. We then obtain an ADT without Strings, on which we apply the implementation of Algorithm 3 described above. We handle the Int and Boolean data types similarly, each with two different values *which are not prefix of each other* (we refer to the proof of theorem 5.8.1.1).

5.11 Evaluation

Although this work is mostly theoretical, we now depict through some benchmarks how many and which kind of questions our system is able to ask (Figure 5.5).

[3] http://www.brics.dk/automaton/

Name	Test set size *Total*	The output was: inferred *total*	asked *total*	asked with... nothing	a hint	suggestions
Grammar (Sec. 5.2)	116	102	14	6	6	2
Html tags (Ex. 2, 8, 9)	35	28	7	4	2	1
Html tags+attributes	60	52	8	2	4	2
Html xml+attributes	193	179	14	5	3	6
Binary (01001x)	15	12	3	1	2	0
Binary (11x)	15	12	3	3	0	0
Binary (ababx)	15	11	4	3	0	1
Binary (01001)	15	10	5	3	0	2
Binary (aababababbab)	15	9	6	3	0	3
$A_x(B_y(F_z))$ 1	3	0	3	1	2	0
$A_x(B_y(F_z))$ 2	14	8	6	3	3	0
$A_x(B_y(F_z))$ 4	84	67	17	8	4	5
$A_x(B_y(F_z))$ 8	584	552	32	19	5	8
$A_x(B_y(F_z))$ 16	4368	4305	63	32	16	15

Figure 5.5 – Comparison of the number of questions asked for different benchmarks.

The first column is the name of the benchmark. The first two appear in Section 5.2 and in the examples. The third is a variation of the second where we add attributes as well, rendered "^.foo := "bar"". The fourth is the same but rendered in XML instead of tags. Note that because we do not support duplication, we need to have a finite number of tags for XML.

The four rows "binary" illustrate how the number and type of questions may vary only depending on the user's answers. We represent binary numbers as either `Empty` or `Zero(x)` or `One(x)` where `x` is a binary number. We put in parenthesis what a user willing to print `Zero(One(Zero(Zero(One(Empty)))))` would have in mind. The second and the third "discard" `Zero` when printing. The fourth one prints `Empty` as empty, `Zero(x)` as `{x}ab` and `One(x)` as `a{x}b`, which result in an ambiguity not resolved until asking a 3-digit number.

The last five rows of Figure 5.5 also illustrate how the number of asked questions grows linearly, whereas the number of elements in the test set grows cubically. These five rows represent a set of classes of type A taking as argument a class of type B, which themselves take as argument a class of type F. We report on the statistics by varying the number of concrete classes between 1, 2, 4, 8 and 16 (see proof of Lemma 5.7.2.2)

The second column is the size of the test set. For the last five rows, the test set contains a cubic number of elements. The third column is the number of answers our tool was able to "infer" based on previously "asked" questions, whose total number is in the fourth column. The fourth column plus the third one thus equal the second one.

Columns five, six and seven decompose the fourth column into the questions which were either asked without any indication, or with a hint of type "[...]foo[...]" (because the arguments were known), or with explicit suggestions where the user just had to enter a number for the choice (see Section 5.10).

5.12 Related Work

Our approach of proactively learning transducers by example, or tree-to-string programs, can be viewed as a particular case of Programming-by-Example. Programming-by-example, also named inductive programming [Polozov and Gulwani, 2015] or test-driven synthesis [Perelman et al., 2014], is gaining more and more attention, notably thanks to Flash Fill in Excel 2013 [Gulwani, 2012a]. Subsequent work demonstrated that these techniques could widely be applicable not only to strings, but when extracting documents [Le and Gulwani, 2014], normalizing text [Kini and Gulwani, 2015] and number transformations [Singh and Gulwani, 2012b]. However, most state-of-the-art programming-by-example techniques rely on the fact that examples are unambiguous and/or that the example provider can check the validity of the final program [Angluin, 1987] [Yessenov et al., 2013] [Feser et al., 2015]. The scope of their algorithms may be larger but they do not guarantee formal result such as polynomial time or non-ambiguity, and often require the user to come up with the examples by himself. More generally, synthesizing recursive functions has recently gained an interest among computer scientists from repairing fragments [Koukoutos et al., 2016] to very precise types [Polikarpova et al., 2016], even by formalizing programming-by-example [Frankle et al., 2016].

Recently, research has pointed out that solving ambiguities is a key to make programming by example accessible, trustful and reduce the number of errors[Mayer et al., 2015][Hottelier et al., 2014]. The power of interaction is already well known in more statistical approaches, e.g. machine learning [Zhang and Chaudhuri, 2015], although recent machine-learning based formatting techniques could benefit from more interaction, because they acknowledge some anomalies [Parr and Vinju, 2016]. In [Gvero et al., 2011] and even [Gvero et al., 2013], the authors solve ambiguities by presenting different code snippets, obtained from synthesizing expressions of an expected type and from other sources of information. Nonetheless, the user has to choose between hard-to-read *code snippets*. Instead of asking which transducer is correct, we ask for what is the right output. Asking sub-examples at run-time proved to be a successful strategy when synthesizing recursive functions [Albarghouthi et al., 2013]. To deal with ambiguous samples, they developed a SATURATE rule to ask for inputs covering the inferred program. In our case, however, such coverage rule still yield the ambiguity raised in example 9, leaving the chance of finding the right program to heuristics.

Researchers have investigated fundamental properties of tree-to-string or tree-to-word transducers [Alur and Cerný, 2010], including expressiveness of even more complex classes

than we consider [Alur and D'Antoni, 2012], but none of them proposed a practical learning algorithm for such transducers. The situation is analogous for Macro Tree Transducers [Bahr and Day, 2013] [Engelfriet and Maneth, 2002]. Lemay [Lemay et al., 2010] explores the synthesis of top-down tree-to-tree transducers using an algorithm similar to L^* for automata [Angluin, 1987] and tree automata [Besombes and Marion, 2004]. These learning algorithms require the user to be in possession of a set of examples that uniquely defines the top-down tree transducer. We instead are able to incrementally ask for examples which resolve ambiguities, although our transducers are single-state. There are also probabilistic tree-to-string transducers [Graehl and Knight, 2004], but they require the use of a corpus and are not adapted to synthesizing small-size code portions with a few examples.

A Gold-style learning algorithm [Laurence et al., 2014, Laurence, 2014, Lemay et al., 2010] was created for sequential tree-to-string transducers. It runs in polynomial-time, but has a drawback: it requires the input/output examples to form a *characteristic sample* for the transducer which is being learned. The transducer which is being learned is however not known in advance. As such, it is not clear in practice how to construct such a characteristic sample. When the input/output examples do not form a characteristic sample, the algorithm might fail, and the user of the algorithm has no indication on which input/output examples should be added to obtain a characteristic sample.

In the case when trees to be printed are programming abstract syntax trees, our work is the dual of the mixfix parsing problem [Jouannaud et al., 1992]. Mixfix parsing takes strings to parse and the wrapping constants to print the trees, and produces the shape of the tree for each string. Our approach requires the shape of the trees and strings of some trees, and produces the wrapping constants to print the trees.

5.12.1 Equivalence of top-down tree-to-string transducers

Since tree test sets uniquely define the behavior of tree-to-string transducers, they can be used for checking tree-to-word transducers equivalence. Checking equivalence of sequential (order-preserving, non-duplicating) tree-to-string transducers can already be solved in polynomial time [Staworko et al., 2009], even when they are duplicating, and not necessarily order-preserving [Maneth and Seidl, 2007].

It was also shown [Helmut Seidl et al., 2015] that checking equivalence of deterministic top-down macro tree-to-string transducers (duplication is allowed, storing strings in registers to output them later is allowed) is decidable. Complexity-wise, this result gives a co-randomized polynomial time algorithm for linear (non-duplicating) tree-to-string transducers. This complexity result was recently improved in [Boiret and Palenta, 2016], where it was proved that checking equivalence of linear tree-to-string transducers can be done in polynomial time.

5.12.2 Test sets

The polynomial time algorithms of [Staworko et al., 2009, Boiret and Palenta, 2016] exploit a connection between the problem of checking equivalence of sequential top-down tree-to-string transducers and the problem of checking equivalence of morphisms over context-free languages [Staworko et al., 2009].

This latter problem was shown to be solvable in polynomial time [Plandowski, 1994, Plandowski, 1995] using test sets. More specifically, this work shows that each context-free language L has a (finite) test set whose size is $O(n^6)$ (originally "finite" in [Albert and Lawrence, 1985, Guba, 1986] and then "exponential" in [Albert et al., 1982]), where n is the size of the grammar. They also provide a lower bound on the sizes of the test sets of context-free languages, by exposing a family of grammars for which the size of the smallest test is $O(n^3)$.

As a result, when checking the equivalence of two morphisms f and g over a context-free language L, it is enough to check the equivalence on the test set of L whose size is polynomial. This result translates (as described in [Staworko et al., 2009]) to checking equivalence between sequential top-down tree-to-string transducers in the following sense. When checking the equivalence of two such transducers P_1 and P_2, it is enough to do so for a finite number of trees, which correspond to the test set of a particular context-free language. This language can be constructed from P_1 and P_2 in time $|P_1||P_2|$.

Remark. Theorem 5.7.1.1 also helps improve the bound for checking equivalence of 1STS with states, using the known reduction from equivalence of 1STS with states to morphisms equivalence over a context-free language (reduction similar to Lemma 5.5.1.3, see [Staworko et al., 2009, Laurence, 2014]).

5.13 Conclusion

We have presented a synthesis algorithms that can learn from example tree-to-string functions, with the input tree as the only argument. This includes functions such as pretty printers. It is crucial that our algorithm can automatically construct a sufficient finite set of input trees, resulting in an interactive synthesis approach that in which the user needs to answer only a linear number of questions in the grammar size. Furthermore, the interaction process driven by our algorithm guarantees that there is no ambiguity: the recursive function of the expected form is unique for a given set of input-output examples. Moreover, we have analyzed the structure of word equations that the algorithm needs to solve and shown that they have a special structure enabling them to be solved in deterministic polynomial time, which results in overall polynomial running time of our synthesizer. Our results make a case that providing tests for tree-to-string functions is a viable alternative to writing the recursive programs directly, an alternative that is particularly appealing for non-expert users.

Page 120 until this page 160 contained the content of the paper "Proactive Synthesis of Recursive Tree-to-String Functions from Examples". We will now add further insights.

C. Epilogue to Proactive Synthesis

C.1 Strong Theoretical Results

In this paper, we proved two strong theoretical results. The first concerning the cubic size of test sets is of historical importance, because it might apply to domains other than tree-to-strings. I remembered talking to my wife about it as the "Mayer-Hamza" theorem, and she would further enjoy the fact that she was married to me because the theorem thus also beared her (married) name. However, I still feel grateful for the original work of Plandowski [Karhumaki et al., 1995] because, without him, we would have never find out this result.

It turns out that we could have found the second theoretical result independently of the first. The linear questioning does not arise from the size of the test set, it is a rather independent property. We did not expect it, to be honest. When we began writing the paper, we did not have this result yet. I then remember conjecturing that we do not need all the elements of the test set, and trying to look for the reason. After we found a way to remove the fourth equation after three of them (see Theorem 5.8.1.1), we suddenly realized that it would imply a linear complexity, and we were amazed. Still, we spent several days finishing the proof on all the cases, which was not obvious at all.

C.2 Discussion

We now answer the questions we asked in the introduction (see page 9) with respect to the work of this chapter.

Question	Prosy
Is it faster/easier to conduct tasks using programming-by-example?	Yes/No
Do users need to see the generated program?	No
Do users need to take on the generated program?	Yes
Is the paraphrased version of the program useful?	N/A
Is it more reliable when the program is shown?	No
Is it more reliable when the computer asks questions?	**Yes**

The comparative summary for all the papers is available on page 163.

Compared to standard programming without help, it is faster to answer the questions than to write the program oneself. However, we did not measure its performance against

what a full-grown integrated development environment toolkit could bring. Users do not need to see the generated program because it can be trusted mathematically. However, they might want to take it and incorporate it in their existing codebase. Moreover, as we can produce a (unfinished) printer as soon as we have examples covering the whole program, the complete questioning is necessary to gain the trust of the user.

6 Discussion and Future Work

"La programmation par l'exemple est l'une des rares technologies pouvant faire chuter le mur de Berlin qui a toujours séparé les programmeurs des utilisateurs."

"Programming-by-example is one of the few technologies with the potential for breaking down the Berlin Wall that has always separated programmers from users."
Henry Lieberman

Here, we quantitatively and qualitatively analyze the findings of the previous chapters. We summarize the answers to the questions of the introduction for each chapter, and cross-compare them. Then we compare the utility of the tools developed within each chapter, with respect to the three dimensions of interactivity, programmability, and exemplarity.

6.1 Contributions and Empirical Answers

We summarize the answer to the questions of the introduction (page 9) for the papers mentioned in this thesis (pages 44, 75, 106, 161). These questions formulate what could be the needs of users and how the four papers bring different answers, depending on their context. The answers in bold represent the results with strong evidence, such as a proof or a statistically significant user study.

	Game	StriSynth	FlashProg	Prosy
Is it faster/easier to conduct tasks using programming-by-example?	Yes/No	Yes/No	Yes	Yes/No
Do users need to see the generated program?	Yes	Yes	Yes	No
Do users need to take on the generated program?	Yes	No	No	Yes
Is the paraphrased version of the program useful?	Conjectured	Yes	Yes	N/A
Is it more reliable when the program is shown?	N/A	Yes	**Yes**	No
Is it more reliable when the computer asks questions?	N/A	N/A	**Yes**	**Yes**

We note that, although programming-by-example looks better in most of the works, it would still be worth integrating it in a true programming language, especially for users looking to reuse the generated programs. We identified the needs for seeing and modifying the generated program; we also identified that questioning, or interactivity or pro-activeness from the computer, is a key role for the enhancement of programming-by-example. From the results on paraphrasing, it appears that generating comments on what the program is doing is as important as generating the program itself. This is consistent with many software development methods that require developers to write documentation or tests about the same time as writing their code (e.g. Rapid application development[1], Waterfall development[2])

6.2 Comparison of the Four Papers on the Dimensions of "Interactivity", "Programmability", and "Exemplarity'

We compare the works of the four previous chapters on three dimensions that are directly linked with the title of this thesis (see Figure 6.1).

- Interactivity: Is there feedback, or is there an effort from the computer to understand the user's goal?
- Programmability: Is the program modifiable? Is the programming language expressive?
- Exemplarity: Are examples well suited for the task? Do they reflect the goal?

[1] wikipedia.org/wiki/Rapid_application_development
[2] wikipedia.org/wiki/Waterfall_model

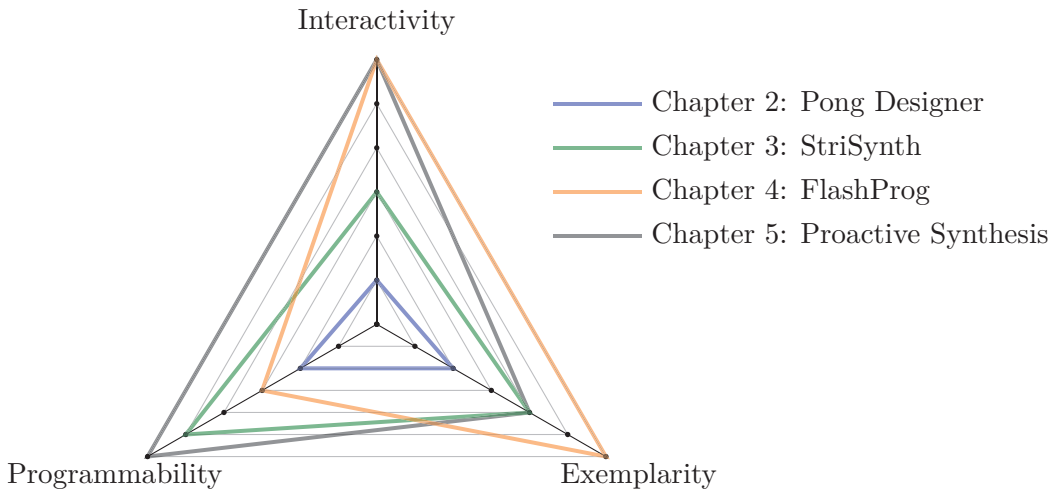


Figure 6.1 – Comparison of the four works in this thesis according to interactivity, programmability, and exemplarity.

For each dimension, we set up a qualitative scale of 0 (completely no, the center) to 6 (yes, absolutely), and we assigned numbers based on the following remarks:

"Interactivity"

Is there feedback, or is there an effort from the computer to understand the user's goal?

- Pong Designer: Besides the interaction with the finger, every interaction comes from the user, there is no feedback from the computer in case of ambiguity, so we gave the rating 1/6.
- StriSynth: With the read-eval-print loop (REPL) of StriSynth, there is always some feedback printed by the computer about what it was able to infer, this explains our rating of 3/6.
- FlashProg: With the iterated questioning, the feedback is very high, and because we proved that it reduced the number of errors and augmented the trust, we put a rating of 6/6.
- Prosy: The algorithms work at best with interaction from the beginning, they offer strong guarantees for the complexity, and the suggestions are pre-computed, we rated it 6/6

"Programmability"

Is the program modifiable? Is the programming language expressive?

- Pong Designer: Users can change the constants of the program statements and choose alternatives. However, they cannot add or write anything: Even if the underlying DSL is Turing-complete, it is not possible to write any program from the interface. Hence, we rated it 2/6.

- StriSynth: As we embedded StriSynth in Scala, users can chain and reuse the programs very easily. However, they cannot modify them directly, they can only add new examples; nonetheless, the export script to PowerShell makes it possible to edit a raw version of the program. We rated it 5/6.

- FlashProg: A user can choose alternatives in the program view, as she would do for Pong Designer. And, choosing alternatives adds examples, hence it does not display discarded alternatives anymore. In addition, programs can be chained using examples and by manipulating the output. It is however not possible to export or modify the program. We rated it 3/6

- Prosy: Prosy produces a complete program that can be further edited and maintained. With the help of an integrated development editor (IDE), the program can be maintained by updating examples. We rated it 6/6.

"Exemplarity"

Are examples well suited for the task? Do they reflect the goal?

- Pong Designer: As observed in the Epilogue, it is not obvious what an example of a game rule means for new persons. We rated it 2/6.

- StriSynth: For transforming strings, examples are straightforward and much better suited for the task. For partitioning and filtering, the way of providing examples would also match users' goals. However, there is no way to specify examples for chaining operations, and currently the syntax is everything but appealing. We rated it 4/6.

- FlashProg: The examples consisting of painting the text are popular and straightforward to understand, and they truly reflect the need. We rated it 6/6.

- Prosy: Telling the computer how to print the trees it provides is one way to program pretty-printers, but we do not have evidence that providing examples is more suited to the task than writing the program directly if the IDE could create beforehand the program structure. We rated it 4/6.

6.3 Future Work

We have several ideas that could serve as future work. First, we could start to improve on the topics related to each of these papers.

- Pong Designer: As a side effect of Pong designer, a way to record doodles, such as those created by Wandida (Wandida.com) could be created. Currently, doodles require a toolchain such as filming, cutting, squeezing animations to sound, mounting and exporting. With an engine such as the one Pong Designer provides, editing videos recorded by using this toolchain would be much easier.

- StriSynth: A way to push StriSynth to the limits would be to natively integrate automatic assistants in OS in order to dealing with the renaming and the moving of files. Reusing new and much faster learning engines ([Singh, 2016]) would probably increase the chances of such technology to be adopted.

- FlashProg: The technology behind FlashProg has been renamed Microsoft Prose [Gulwani, 2015], and it is currently serving many teams in Microsoft. The real benefit from this thesis would be to export the interactivity features to products using Microsoft Prose. For example, system administrators would enjoy having augmented versions of Convert-String [PowerShell Microsoft team, 2015] and ConvertFrom-String [PowerShell Microsoft Team, 2014, PowerShell Microsoft Team, 2015], where the interaction helps them to choose the right extraction program.

- Prosy: For the future, it would be interesting to see how we could avoid generating the $O(n^3)$ set and then prune the questions. Indeed, in our approach, the test set of size $O(n^3)$ is already factorized because it comes from paths in a graph. Being able to learn programs with state would be useful, for example to learn indentation. It would also be useful to discover how to learn recursive functions that deal with duplicated elements (e.g., opening and closing XML tags) and out-of-order elements.

  We might also to ask even less questions if we could ask them in the right order. For example, at the end of the sandwich story (page 118), we briefly mention that testing for two and three coins enables us to determine the result for one coin.

  Finally, asking questions about trees larger than elements in the test set could reduce the number of questions to ask. We conjecture, for example, that only two questions would suffice for correctly printing trees, whatever the number of nodes there are. Of course, users might spend more time writing the answer to one such big question than writing answers for every small question our algorithms ask.

Based on the previous qualitative analysis, it appears that it becomes desirable to have hybrid programming engines that not only provide the standard functionalities of IDEs

but that also provide functionalities for developing programs in native and comfortable user-interfaces by using programming-by-example engines.

For example, instead of building tools on top of a programming language, why not have the tool itself to write and maintain a program? Many editors lack the flexibility to create and reuse abstractions and constraints, and programming languages usually cannot be modified by modifying their output. There is room for bridging the gap between the two. We could imagine empowering users with an interface as nice as Microsoft PowerPoint, but as flexible and programmatic as LaTeX; or an interface as nice as Google Sites but as programmable as PhP; or an interface as nice as Word, but with commands that have the power of LaTeX; or an online slide editor as nice as Reveal.js but as editable as JavaScript. Chugh [Chugh et al., 2016] already started the work of having an interface as nice as Inkscape for editing SVGs but as powerful as Lisp, hence this is becoming reality.

7 Conclusion

"Dans la vie, rien n'est à craindre,
tout est à comprendre."

"In life, there is nothing to fear,
everything is to be understood."
Marie Curie

Throughout this thesis, we have reviewed experiences investigating how to improve programming-by-example. We analyzed a programming-by-example-based game engine for tablets, a toolkit for managing files by example, an engine for extracting data based on examples, and strong algorithms for learning recursive tree-to-string functions. We have compared these works together, and we have reached the conclusion that programming-by-example is desirable for many tasks, and that its inherent ambiguity can be resolved through various interaction models. Especially, we acknowledge that questioning (otherwise named conversational clarification, disambiguation or feedback) provably increases the performance of programming-by-demonstration and augments the confidence of users in the result. Furthermore, we discovered that showing the program helps users to gain trust, especially when the computer paraphrases it in plain English.

We hope that this five-year work will serve as an example that influences the future of programming-by-example, urging researchers and developers to create interactive communication techniques between computers and users. We expect such advances to contribute further to bridging the gap between users' goals and computers' ability to understand them. We wish that this bridging eventually opens a new door to people who would be interested by programming, if only its first step consisted in providing examples and receiving feedback about them.

Bibliography

[Abelson and McKinney, 2010] Abelson, H. and McKinney, A. (2010). AppInventor: App Inventor for Android. http://appinventor.mit.edu/explore/.

[AgentSheets, Inc., 2017] AgentSheets, Inc. (2017). AgentCubes Online : Teach Programming using 3d Games. https://www.agentcubesonline.com/.

[Albarghouthi et al., 2013] Albarghouthi, A., Gulwani, S., and Kincaid, Z. (2013). Recursive Program Synthesis. In *Computer Aided Verification*, pages 934–950. Springer, Berlin, Heidelberg.

[Albert et al., 1982] Albert, J., Culik, K., and Karhumäki, J. (1982). Test sets for context free languages and algebraic systems of equations over a free monoid. *Information and Control*, 52(2):172–186.

[Albert and Lawrence, 1985] Albert, M. H. and Lawrence, J. (1985). A proof of Ehrenfeucht's Conjecture. *Theoretical Computer Science*, 41:121–123.

[Alur et al., 2013] Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–17. IEEE.

[Alur and Cerný, 2010] Alur, R. and Cerný, P. (2010). Expressiveness of streaming string transducers. In Lodaya, K. and Mahajan, M., editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Alur and D'Antoni, 2012] Alur, R. and D'Antoni, L. (2012). Streaming tree transducers. In *Automata, Languages, and Programming*, pages 42–53. Springer.

[Amershi et al., 2009] Amershi, S., Fogarty, J., Kapoor, A., and Tan, D. S. (2009). Overview based example selection in end user interactive concept learning. In *UIST*, pages 247–256.

[Amershi et al., 2011] Amershi, S., Fogarty, J., Kapoor, A., and Tan, D. S. (2011). Effective End-User Interaction with Machine Learning. In *AAAI*.

[Androutsopoulos et al., 1995] Androutsopoulos, I., Ritchie, G. D., and Thanisch, P. (1995). Natural language interfaces to databases-an introduction. *arXiv preprint cmp-lg/9503016*.

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106.

[Atomic Energy of Canada Limited, 1985] Atomic Energy of Canada Limited (1985). Therac-25. https://en.wikipedia.org/wiki/Therac-25.

[Bahr and Day, 2013] Bahr, P. and Day, L. E. (2013). Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM.

[Barowy et al., 2015] Barowy, D., Gulwani, S., Hart, T., and Zorn, B. (2015). FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *PLDI*.

[Besombes and Marion, 2004] Besombes, J. and Marion, J.-Y. (2004). Learning tree languages from positive examples and membership queries. In *International Conference on Algorithmic Learning Theory*, pages 440–453. Springer.

[Bishop et al., 1998] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M. (1998). Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46 –53.

[Bodík et al., 2010] Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., and Rodarmor, C. (2010). Programming with angelic nondeterminism. In *POPL*, pages 339–352.

[Boiret, 2016] Boiret, A. (2016). Normal Form on Linear Tree-to-word Transducers. In *10th International Conference on Language and Automata Theory and Applications*.

[Boiret and Palenta, 2016] Boiret, A. and Palenta, R. (2016). Deciding Equivalence of Linear Tree-to-Word Transducers in Polynomial Time. In *International Conference on Developments in Language Theory*, pages 355–367. Springer.

[Burckhardt et al., 2013] Burckhardt, S., Fahndrich, M., de Halleux, P., McDirmid, S., Moskal, M., Tillmann, N., and Kato, J. (2013). It's alive! continuous feedback in UI programming. In *ACM SIGPLAN Notices*, volume 48, pages 95–104. ACM.

[Chandra et al., 2011] Chandra, S., Torlak, E., Barman, S., and Bodik, R. (2011). Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA. ACM.

[Cheema et al., 2012] Cheema, S., Gulwani, S., and LaViola, J. (2012). QuickDraw: improving drawing experience for geometric diagrams. In *Proc. of the SIGCHI conference*, pages 1037–1064, NY, USA. ACM.

[Chugh et al., 2016] Chugh, R., Hempel, B., Spradlin, M., and Albers, J. (2016). Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–354. ACM.

[Cooper et al., 2012] Cooper, A., Reimann, R., and Cronin, D. (2012). *About face 3: the essentials of interaction design*. Wiley.

[Cousot et al., 2001] Cousot, P., Cousot, R., Feret, J., Miné, A., Rival, X., Blanchet, B., Monniaux, D., and Mauborgne, L. (2001). The Astrée Static Analyzer:. http://www.astree.ens.fr/.

[Cypher, 1991] Cypher, A. (1991). EAGER: Programming Repetitive Tasks by Example. In *CHI*, pages 33–39.

[Cypher and Halbert, 1993] Cypher, A. and Halbert, D. (1993). *Watch what I Do: Programming by Demonstration*. MIT Press.

[Distefano and Calcagno, 2009] Distefano, D. and Calcagno, C. (2009). Monoidics' Infer Static Analyzer. https://www.crunchbase.com/organization/monoidics.

[Duncan and Kay, 2013] Duncan, S. and Kay, R. (2013). GameMaker: Fast, cross-platform games development. http://www.yoyogames.com/gamemaker/studio.

[Elliott, 2001] Elliott, C. (2001). Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, pages 41–69.

[Engelfriet and Maneth, 2002] Engelfriet, J. and Maneth, S. (2002). Output string languages of compositions of deterministic macro tree transducers. *Journal of Computer and System Sciences*, 64(2):350–395.

[European Space Agency, 1996] European Space Agency (1996). Ariane 5 Explosion. https://fr.wikipedia.org/wiki/Vol_501_d

[Feser et al., 2015] Feser, J. K., Chaudhuri, S., and Dillig, I. (2015). Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 229–239, New York, NY, USA. ACM.

[Frankle et al., 2016] Frankle, J., Osera, P.-M., Walker, D., and Zdancewic, S. (2016). Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA. ACM.

[Galenson et al., 2014] Galenson, J., Reames, P., Bodík, R., Hartmann, B., and Sen, K. (2014). CodeHint: dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663.

[Gao et al., 2015] Gao, T., Dontcheva, M., Adar, E., Liu, Z., and Karahalios, K. G. (2015). DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 489–500. ACM.

[Graehl and Knight, 2004] Graehl, J. and Knight, K. (2004). Training tree transducers. Technical report, DTIC Document.

[Guba, 1986] Guba, V. S. (1986). Equivalence of infinite systems of equations in free groups and semigroups to finite subsystems. *Mathematical Notes*, 40(3):688–690.

[Gulwani, 2011] Gulwani, S. (2011). Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA. ACM.

[Gulwani, 2012a] Gulwani, S. (2012a). Synthesis from Examples. In *WAMBSE Special Issue, Infosys Labs Briefings*, volume 10(2).

[Gulwani, 2012b] Gulwani, S. (2012b). Synthesis from Examples: Interaction Models and Algorithms. In *SYNASC*, pages 8–14.

[Gulwani, 2013] Gulwani, S. (2013). Flash Fill Microsoft Excel 2013 feature. http://research.microsoft.com/en-us/um/people/sumitg/flashfill.html.

[Gulwani, 2015] Gulwani, S. (2015). Microsoft Prose SDK. https://microsoft.github.io/prose/.

[Gulwani et al., 2012] Gulwani, S., Harris, W. R., and Singh, R. (2012). Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105.

[Gulwani et al., 2015a] Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. (2015a). Inductive Programming Meets the Real World. *Commun. ACM*, 58(11):90–99.

[Gulwani and Marron, 2014] Gulwani, S. and Marron, M. (2014). NLyze: Interactive Programming by Natural Language for Spreadsheet Data Analysis and Manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 803–814, New York, NY, USA. ACM.

[Gulwani et al., 2015b] Gulwani, S., Mayer, M., Niksic, F., and Piskac, R. (2015b). Strisynth: synthesis for live programming. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 701–704. IEEE Press.

[Gvero and Kuncak, 2015] Gvero, T. and Kuncak, V. (2015). Interactive Synthesis Using Free-form Queries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 689–692, Piscataway, NJ, USA. IEEE Press.

[Gvero et al., 2013] Gvero, T., Kuncak, V., Kuraj, I., and Piskac, R. (2013). Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 27–38, New York, NY, USA. ACM.

[Gvero et al., 2011] Gvero, T., Kuncak, V., and Piskac, R. (2011). Interactive Synthesis of Code Snippets. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 418–423, Berlin, Heidelberg. Springer-Verlag.

[Helmut Seidl et al., 2015] Helmut Seidl, Sebastian Maneth, and Gregor Kemper (2015). Equivalence of deterministic top-down tree-to-string transducers is decidable. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 943–962. IEEE.

[Hottelier et al., 2014] Hottelier, T., Bodik, R., and Ryokai, K. (2014). Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*.

[Jansson, 2000] Jansson, P. (2000). *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden.

[Jeż, 2017] Jeż, A. (2017). Word equations in linear space. *arXiv preprint arXiv:1702.00736*.

[Jha et al., 2010] Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM.

[Jouannaud et al., 1992] Jouannaud, J.-P., Kirchner, C., Kirchner, H., and Megrelis, A. (1992). Programming with equalities, subsorts, overloading, and parametrization in OBJ. *The Journal of Logic Programming*, 12(3):257–279.

[Jourdan et al., 2015] Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X., and Pichardie, D. (2015). A Formally-Verified C Static Analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA. ACM.

[Kandel et al., 2011] Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. (2011). Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM.

[Karhumaki et al., 1995] Karhumaki, J., Plandowski, W., and Rytter, W. (1995). Polynomial size test sets for context-free languages. *Journal of Computer and System Sciences*, 50(1):11–19.

[Kini and Gulwani, 2015] Kini, D. and Gulwani, S. (2015). FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 776–783, Buenos Aires, Argentina. AAAI Press.

[Köksal et al., 2012] Köksal, A. S., Kuncak, V., and Suter, P. (2012). Constraints as control. In *POPL*, pages 151–164, NY, USA. ACM.

[Kneuss et al., 2015] Kneuss, E., Koukoutos, M., and Kuncak, V. (2015). Deductive program repair. In *International Conference on Computer Aided Verification*, pages 217–233. Springer.

[Kneuss et al., 2013] Kneuss, E., Kuraj, I., Kuncak, V., and Suter, P. (2013). Synthesis modulo recursive functions. In *Acm Sigplan Notices*, volume 48, pages 407–426. ACM.

[Ko and Myers, 2008] Ko, A. J. and Myers, B. A. (2008). Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proc. of the 30th ICSE*, pages 301–310, NY, USA. ACM.

[Koukoutos et al., 2016] Koukoutos, M., Kneuss, E., and Kuncak, V. (2016). An Update on Deductive Synthesis and Repair in the Leon Tool. *arXiv preprint arXiv:1611.07625*.

[Kuncak, 2015] Kuncak, V. (2015). Developing Verified Software Using Leon. In *NFM*, pages 12–15.

[Kuncak et al., 2010a] Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2010a). Comfusy: A Tool for Complete Functional Synthesis (Tool Presentation). In *CAV, Proceedings*, volume 6174, pages 430–433. Springer-Verlag Berlin.

[Kuncak et al., 2010b] Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2010b). Complete Functional Synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA. ACM.

[Kuncak et al., 2012] Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2012). Software synthesis procedures. *Communications of the ACM*, 55(2):103–111.

[Kuncak, 2007] Kuncak, V. V. J. (2007). *Modular data structure verification*. Thesis, Massachusetts Institute of Technology.

[Landauer and Hirakawa, 1995] Landauer, J. and Hirakawa, M. (1995). Visual AWK: a model for text processing by demonstration. In *Visual Languages, IEEE Symposium on*, pages 267–267. IEEE Computer Society.

[Lau, 2009] Lau, T. (2009). Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65.

[Lau et al., 2000] Lau, T., Domingos, P., and Weld, D. S. (2000). Version Space Algebra and its Application to Programming by Demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 527–534, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Lau et al., 2003a] Lau, T., Domingos, P., and Weld, D. S. (2003a). Learning programs from traces using version space algebra. In *Proc. of the 2nd K-CAP*, pages 36–43, NY, USA. ACM.

[Lau and others, 2008] Lau, T. and others (2008). Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*.

[Lau et al., 2001] Lau, T., Wolfman, S. A., Domingos, P., and Weld, D. S. (2001). Learning repetitive text-editing procedures with SMARTedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209–226.

[Lau et al., 2003b] Lau, T. A., Wolfman, S. A., Domingos, P., and Weld, D. S. (2003b). Programming by Demonstration Using Version Space Algebra. *Machine Learning*, 53(1-2):111–156.

[Laurence, 2014] Laurence, G. (2014). *Normalisation et Apprentissage de Transductions d'Arbres en Mots*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I.

[Laurence et al., 2014] Laurence, G., Lemay, A., Niehren, J., Staworko, S., and Tommasi, M. (2014). Learning sequential tree-to-word transducers. In *International Conference on Language and Automata Theory and Applications*, pages 490–502. Springer.

[Le and Gulwani, 2014] Le, V. and Gulwani, S. (2014). FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 55. ACM.

[Le et al., 2013] Le, V., Gulwani, S., and Su, Z. (2013). SmartSynth: synthesizing smartphone automation scripts from natural language. In Chu, H.-H., Huang, P., Choudhury, R. R., and Zhao, F., editors, *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 193–206. ACM.

[Le et al., 2017] Le, V., Perelman, D., Polozov, O., Raza, M., Udupa, A., and Gulwani, S. (2017). Interactive Program Synthesis. *arXiv:1703.03539 [cs]*. arXiv: 1703.03539.

[Le et al., 2015] Le, V., Sun, C., and Su, Z. (2015). Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399, New York, NY, USA. ACM.

[Lemay et al., 2010] Lemay, A., Maneth, S., and Niehren, J. (2010). A learning algorithm for top-down XML transformations. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '10, pages 285–296, New York, NY, USA. ACM.

[Leroy, 2006] Leroy, X. (2006). Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA. ACM.

[Leung et al., 2015] Leung, A., Sarracino, J., and Lerner, S. (2015). Interactive Parser Synthesis by Example. In *PLDI*.

[Lieberman, 1993] Lieberman, H. (1993). Mondrian: a teachable graphical editor. In *Proc. of the INTERCHI*, pages 144–, Amsterdam, The Netherlands. IOS Press.

[Lieberman, 2000] Lieberman, H. (2000). Programming by example (introduction). *Communications of the ACM*, 43(3):72–74.

[Lieberman, 2001] Lieberman, H. (2001). *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann.

[Lothaire, 1997] Lothaire, M. (1997). *Combinatorics on words*, volume 17. Cambridge university press.

[Manber and Myers, 1993] Manber, U. and Myers, E. W. (1993). Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948.

[Maneth and Seidl, 2007] Maneth, S. and Seidl, H. (2007). Deciding Equivalence of Top-Down XML Transformations in Polynomial Time. In *PLAN-X*, pages 73–79.

[Mayer and Hamza, 2016] Mayer, M. and Hamza, J. (2016). Optimal Test Sets for Context-Free Languages. *arXiv preprint arXiv:1611.06703*.

[Mayer et al., 2015] Mayer, M., Soares, G., Grechkin, M., Le, V., Marron, M., Polozov, A., Singh, R., Zorn, B., and Gulwani, S. (2015). User interaction models for disambiguation in programming by example. In *28th ACM User Interface Software and Technology Symposium*.

[Mazurak and Zdancewic, 2007] Mazurak, K. and Zdancewic, S. (2007). A BASH: finding bugs in bash scripts. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 105–114. ACM.

[McDaniel, 2001] McDaniel, R. (2001). *Your wish is my command*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[McDaniel and Myers, 1999] McDaniel, R. G. and Myers, B. A. (1999). Getting More Out of Programming-by-Demonstration. In *CHI*, pages 442–449.

[Menon et al., 2013] Menon, A. K., Tamuz, O., Gulwani, S., Lampson, B. W., and Kalai, A. (2013). A Machine Learning Framework for Programming by Example. In *ICML*, pages 187–195.

[Miller et al., 2013] Miller, H., Haller, P., Burmako, E., and Odersky, M. (2013). Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA. ACM.

[Miller and Myers, 2001] Miller, R. C. and Myers, B. A. (2001). Outlier Finding: Focusing User Attention on Possible Errors. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 81–90, New York, NY, USA. ACM.

[Miller and Myers, 2002] Miller, R. C. and Myers, B. A. (2002). LAPIS: Smart editing with text structure. In *CHI'02 Extended Abstracts on Human Factors in Computing Systems*, pages 496–497. ACM.

[Ministère de la Défense Française, 2011] Ministère de la Défense Française (2011). Louvois: Logiciel unique à vocation interarmées de la solde. http://www.lemonde.fr/societe/article/2017/02/02/paie-des-militaires-les-rates-du-logiciel-louvois-peseront-jusqu-en-2021_5073204_3224.html.

[Mitchell, 1982] Mitchell, T. M. (1982). Generalization as search. *Artificial intelligence*, 18(2):203–226.

[Myers, 1987] Myers, B. A. (1987). Creating Dynamic Interaction Techniques by Demonstration. In *Proceedings of the SIGCHI/GI Conference on Human Factors in Computing Systems and Graphics Interface*, CHI '87, pages 271–278, New York, NY, USA. ACM.

[Myers and Buxton, 1986] Myers, B. A. and Buxton, W. (1986). Creating highly-interactive and graphical user interfaces by demonstration. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 249–258, New York, NY, USA. ACM.

[Myers et al., 1993] Myers, B. A., McDaniel, R. G., and Kosbie, D. S. (1993). Marquise: creating complete user interfaces by demonstration. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 293–300, New York, NY, USA. ACM. Train and show mode.

[Myers et al., 2004] Myers, B. A., Pane, J. F., and Ko, A. (2004). Natural Programming Languages and Environments. *Commun. ACM*, 47(9):47–52.

[Nasilowski, 2011] Nasilowski, S. (2011). Codea: Create Anything on your iPad with Codea. http://twolivesleft.com/Codea/.

[Osera and Zdancewic, 2015] Osera, P.-M. and Zdancewic, S. (2015). Type-and-example-directed program synthesis. In *PLDI*.

[Parr and Vinju, 2016] Parr, T. and Vinju, J. (2016). Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151. ACM.

[Perelman et al., 2014] Perelman, D., Gulwani, S., Grossman, D., and Provost, P. (2014). Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 43. ACM.

[Piskac et al., 2014] Piskac, R., Wies, T., and Zufferey, D. (2014). GRASShopper: Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 124–139. Springer, Berlin, Heidelberg.

[Plandowski, 1994] Plandowski, W. (1994). Testing equivalence of morphisms on context-free languages. In *European Symposium on Algorithms*, pages 460–470. Springer.

[Plandowski, 1995] Plandowski, W. (1995). The complexity of the morphism equivalence problem for context-free languages. *Ph. D. thesis, Dept. of Mathematics, Informatics and Mechanics*.

[Plandowski, 1999] Plandowski, W. (1999). Satisfiability of word equations with constants is in PSPACE. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 495–500. IEEE.

[Polikarpova et al., 2016] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. (2016). Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA. ACM.

[Polozov and Gulwani, 2015] Polozov, O. and Gulwani, S. (2015). FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 107–126, New York, NY, USA. ACM.

[Popescu et al., 2004] Popescu, A.-M., Armanasu, A., Etzioni, O., Ko, D., and Yates, A. (2004). Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics*, page 141. Association for Computational Linguistics.

[PowerShell Microsoft Team, 2014] PowerShell Microsoft Team (2014). ConvertFrom-String: Example-based text parsing. http://blogs.msdn.com/b/powershell/archive/2014/10/31/convertfrom-string-example-based-text-parsing.aspx.

[PowerShell Microsoft team, 2015] PowerShell Microsoft team (2015). Convert-String. https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/convert-string.

[PowerShell Microsoft Team, 2015] PowerShell Microsoft Team (2015). ConvertFrom-String Powershell Feature in Windows Threshold. http://research.microsoft.com/en-us/um/people/sumitg/flashextract.html.

[Raza et al., 2014] Raza, M., Gulwani, S., and Milic-Frayling, N. (2014). Programming by Example Using Least General Generalizations. In *AAAI*, pages 283–290.

[Resnick et al., 2009] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., and Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11):60–67.

[Reynolds et al., 2015] Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., and Barrett, C. (2015). Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 198–216. Springer, Cham.

[Ruvini, 2004] Ruvini, J.-D. (2004). The Challenges of Implicit Programming by Example. *IUI04*.

[Sal, 2012] Sal, S. (2012). Khan Academy. http://www.khanacademy.org.

[Scaffidi et al., 2008] Scaffidi, C., Myers, B., and Shaw, M. (2008). Topes: reusable abstractions for validating data. In *Proceedings of the 30th international conference on Software engineering*, pages 1–10. ACM.

[Scirra, 2013] Scirra, L. (2013). Construct 2: Create games. Effortlessly. https://www.scirra.com/construct2.

[Shacham et al., 2009] Shacham, O., Vechev, M., and Yahav, E. (2009). Chameleon: adaptive selection of collections. *OOPSLA*, 44(6):408–418.

[Shen et al., 2014] Shen, Y., Chakrabarti, K., Chaudhuri, S., Ding, B., and Novik, L. (2014). Discovering queries based on example tuples. In *SIGMOD*, pages 493–504.

[Simonyi et al., 2006] Simonyi, C., Christerson, M., and Clifford, S. (2006). Intentional software. *OOPSLA Not.*, 41(10):451–464.

[Singh, 2016] Singh, R. (2016). Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827.

[Singh and Gulwani, 2012a] Singh, R. and Gulwani, S. (2012a). Learning Semantic String Transformations from Examples. *PVLDB*, 5(8):740–751.

[Singh and Gulwani, 2012b] Singh, R. and Gulwani, S. (2012b). Synthesizing number transformations from input-output examples. In *Proc. of the 24th CAV conference*, pages 634–651, Berlin, Heidelberg. Springer-Verlag.

[Singh and Gulwani, 2015] Singh, R. and Gulwani, S. (2015). Predicting a Correct Program in Programming by Example. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 398–414. Springer, Cham.

[Smith et al., 2000] Smith, D. C., Cypher, A., and Tesler, L. (2000). Programming by Example: Novice Programming Comes of Age. *Commun. ACM*, 43(3):75–81.

[Sánchez-Ruíz and Jamba, 2008] Sánchez-Ruíz, A. J. and Jamba, L. A. (2008). FunFonts: introducing 4th and 5th graders to programming using Squeak. In *Proc. of the 46th Annual Southeast Regional Conference on XX*, pages 24–29, NY, USA. ACM.

[Solar-Lezama, 2008] Solar-Lezama, A. (2008). *Program Synthesis by Sketching*. PhD thesis, UC Berkeley.

[Solar-Lezama et al., 2007] Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S. (2007). Sketching stencils. *ACM SIGPLAN Notices*, 42(6):167–178.

[Srivastava et al., 2010] Srivastava, S., Gulwani, S., and Foster, J. S. (2010). From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA. ACM.

[Stack Overflow, 2009] Stack Overflow (2009). What is the most difficult/challenging regular expression you have ever written? https://stackoverflow.com/q/800813/1287856.

[Stack Overflow, 2011] Stack Overflow (2011). Auto increment a variable in regex. https://stackoverflow.com/q/4615137/1287856.

[Staworko et al., 2009] Staworko, S., Laurence, G., Lemay, A., and Niehren, J. (2009). Equivalence of deterministic nested word to word transducers. In *International Symposium on Fundamentals of Computation Theory*, pages 310–322. Springer.

[Stolee and Fristoe, 2011] Stolee, K. T. and Fristoe, T. (2011). Expressing computer science concepts through Kodu game lab. In *Proc. of the 42nd ACM SIGCSE*, NY, USA. ACM.

[Super User, 2011] Super User (2011). How to batch combine JPEG's from folders into PDF's? https://superuser.com/q/362433/245250.

[Suter, 2012] Suter, P. P. H. (2012). *Programming with Specifications*. PhD thesis, EPFL.

[Synopsys, 2016] Synopsys (2016). Coverity: Static Code Analysis. http://www.coverity.com/.

[Tillmann et al., 2011] Tillmann, N., Moskal, M., Halleux, J. d., and Fahndrich, M. (2011). TouchDevelop: programming cloud-connected mobile devices via touchscreen. *Proc. of the 10th OOPSLA*, ONWARD.

[Victor, 2012a] Victor, B. (2012a). Inventing on Principle. http://vimeo.com/36579366.

[Victor, 2012b] Victor, B. (2012b). Learnable Programming. http://worrydream.com/LearnableProgramming/.

[Vigyan Singhal, 1999] Vigyan Singhal (1999). JasperGold Formal Verification Platform. https://www.crunchbase.com/organization/jasper-design-automation

[Wang and Cook, 2004] Wang, G. and Cook, P. (2004). ChucK: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815.

[Wang et al., 2017] Wang, S. I., Ginn, S., Liang, P., and Manning, C. D. (2017). Naturalizing a Programming Language via Interactive Learning. *arXiv preprint arXiv:1704.06956*.

[Wei et al., 2014] Wei, Y., Hamadi, Y., Gulwani, S., and Raghothaman, M. (2014). C# code snippets on-demand. http://www.robmiles.com/journal/2014/2/11/c-code-snippets-on-demand.

[Witten and Mo, 1993] Witten, I. H. and Mo, D. (1993). TELS: Learning Text Editing Tasks from Examples. In Cypher, A., editor, *Watch What I Do*, pages 183–203. MIT Press.

[Yessenov et al., 2013] Yessenov, K., Tulsiani, S., Menon, A., Miller, R. C., Gulwani, S., Lampson, B., and Kalai, A. (2013). A colorful approach to text processing by example. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 495–504. ACM.

[Zhang and Chaudhuri, 2015] Zhang, C. and Chaudhuri, K. (2015). Active learning from weak and strong labelers. In *Advances in Neural Information Processing Systems*, pages 703–711.

[Zhang and Sun, 2013] Zhang, S. and Sun, Y. (2013). Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234.

A Take-aways

To put this thesis briefly, one can refer to these take-aways written by the author. They are available in English (right) and in French (left).

"Tous les programmes ont des limites:
Un jour, quelqu'un qui aura besoin d'une option
ne la trouvera pas dans le programme.
Un pouvoir illimité vient avec un accès illimité en écriture au programme,
surtout pour qui aura besoin d'options.

"All programs have limitations:
Finally, someone needing
a feature will not find it in the program.
An unlimited power comes with an unlimited modifiability to the program,
especially for anyone needing features."

"Deux organisations font leur travail
sans poser de questions, bien qu'il faudrait:
Les assassins et les ordinateurs.

"Two organizations do their job without
asking questions, although they should:
Assassins and computers."

"Les ordis savent résoudre des problèmes.
Leur problème le plus difficile est de trouver
les problèmes que les gens veulent résoudre.

"Computers are good at solving problems.
Their hardest problem is to find the
problems people want to solve."

"Mieux vaut prévenir que guérir.
Mieux vaut vérifier que déboguer.
Mieux vaut communiquer qu'ordonner."

"Prevention is better than cure.
Verification is better than debugging.
Communication is better than ordering."

"Quel est actuellement le point commun
principal entre la programmation
et la gestion d'une équipe?
C'est le point sur le i."

"What is currently the main common point
between programming
and managing a team?
It's the point on the i."

"La question n'est pas comment les scientifiques peuvent rendre les ordinateurs plus productifs mais comment les ordinateurs peuvent rendre les scientifiques plus productifs

"The question is not how scientists can make computers more productive, but how computers can make scientists more productive."

"Les entreprises ont besoin de programmes vérifiés. Les utilisateurs ont besoin de programmes clarifiés."

"Businesses need verified programs. End-users need clarified programs."

"**Ordinateur:** Vos désirs sont mes ordres.
Humain: Génial, je veux un jeu de Pong avec des animations 3D !
Ordinateur: Erreur de syntaxe. Merci d'écrire vos désirs dans un langage de programmation non ambigü."

"**Computer:** Your wish is my command.
Human: Great! I would like a Pong game with 3D animations!
Computer: Syntax error. Please write your wish in an unambiguous programming language."

"*. . . [Après 3772 questions]*
Ordinateur: Qu'entends-tu par "raquette" ?
Humain: Je n'aurais jamais dû essayer de programmer un jeu de pong par la voix. . . "

"*. . . [After 3772 questions]*
Computer: What do you mean by "paddle"?
Human: I knew I should have never tried programming a pong game by voice . . . "

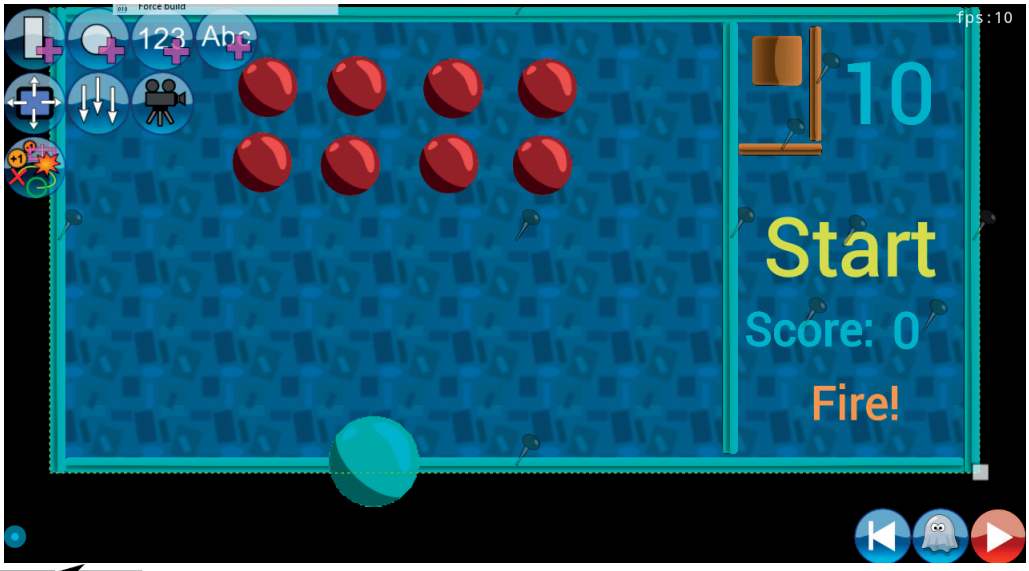
Mikaël Mayer

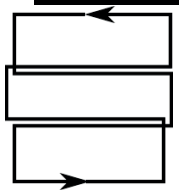
B Pong Designer Tutorials

B.1 Space Invaders Tutorial

In this appendix not published with the paper of chapter 2 page 11, we explain how to use Pong Designer to create a Space-Invaders game.

The purpose of a space-invaders game is to destroy the invaders that are descending from the sky. Programming such a game involves a little bit more logic than usual. It demonstrates a not so straightforward use of conditional statements to describe the moves of the enemy.





Enemies are slowly traveling from right to left, and at time intervals, they velocity is inversed and they step down a little. After 10 steps downwards, instead of stepping down, they step up until they reach their initial position.

As we do not yet support path-by-example paradigms, we will demonstrate how to program this trajectory by graphically specifying how the speed vector changes and the relative movement.

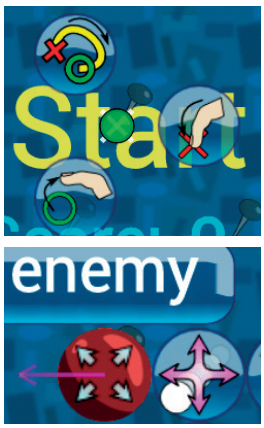
A bouncing block will control the regularity of this movement, regularly decreasing a number. Depending on the number's value, the game will move the enemy either up or down, and each time reverse its speed.

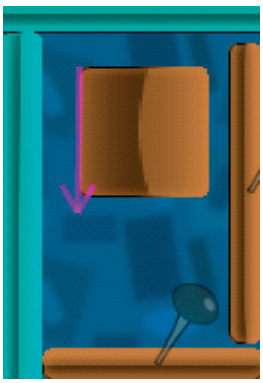
The player controls a circular paddle constrained on the bottom of the screen. If they press "Fire", a projectile appears in front of them and shoots upwards. If the projectile hits the enemy, we remove the enemy, and the player can shoot again.

Each time the projectile hits an enemy, the score increases. From time to time, a random projectile from the sky goes through the enemies, and if it hits the player, the player looses a point.

We start with a layout containing all the necessary shapes and only one enemy. First, we would like the game to start when we press on "start". This requires the enemy and the block decreasing the counter to start to move.

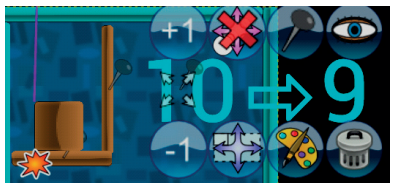
- Press on the event menu, press on the "start" label. Even if no events occurred yet, it displays three possible events that could occur on this label. Select the green circle corresponding to a "finger up" event. Change the velocity of the moving block to point downwards, and the speed of the enemy to point to the left, and hide the start label. Accept the rule.



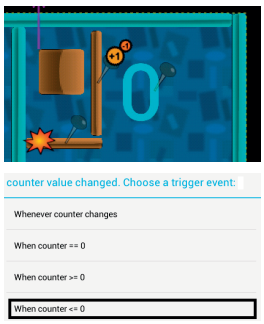


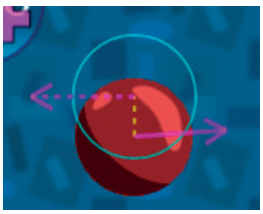
Second, we would like to set up the moving block to decrease the counter, and by the same way to set up the counter change to make the enemy change its direction. Besides, when this counter is positive, the enemy should step down a little, and when it is negative, the enemy should step up a little. When the counter reaches -10, it goes back to 10.

1. Start the game, wait for the collision of the trigger with the orange wall. Pause the game, press on the event menu, select the collision. Change the counter on the right from 10 to 9. Accept the rule.



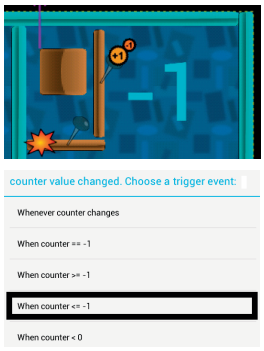
2. Select the counter that now displays 9. Change its value to 0. Click on the event menu. Press on the counter, and choose the event when its value is greater or equal to 0. Turn the speed of the enemy by 180 degrees, and make it step down a little. Accept the rule.

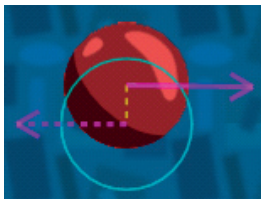




3. Start the game for a while. Watch the counter decreasing once. When it reaches -1, pause the game. It is also possible to change it manually without waiting.

4. Press on the event menu, select the event that occurred to produce -1. Select the condition "when the value is ≤ -1". Turn the speed of the enemy by 180 degrees, and make it step up a little. Accept the rule.

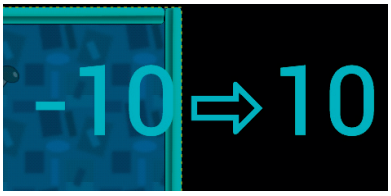




5. Start the game for a while. Watch the counter decreasing to -3. Unfortunately, the enemy does not go up any further. By pausing the game and selecting the rule, we would see that the system inferred absolute positions. By explicitly switching between the lines of code or providing a new input/output example, we can solve the problem.

6. Relaunch the simulation. When the counter reaches -10, pause the game.

   Press on the event menu, select the event that occurred to produce -10. Select the condition "when the value equals -10". Change the value from -10 to 10 (see below). Accept the rule.



Below, we show the kind of code our system produced for the last three steps.

```
counter
WhenIntegerChanges(counter) { (newValue) =>
  if(newValue <= -11) {
    ennemy.y = ennemy.prev_y + -20 // <-|->
    ennemy.angle = ennemy.prev_angle - 180 // <-|->
  } else {
    if(newValue <= -10) {
      ennemy.y = ennemy.prev_y + -20 // <-|->
      ennemy.angle = 90 // <-|->
      counter.value = Score1.prev_value - counter.prev_value
    } else {
      if(newValue <= -1) {
        ennemy.y = ennemy.prev_y + -20 // <-|->
        ennemy.angle = ennemy.prev_angle - 180 // <-|->
```

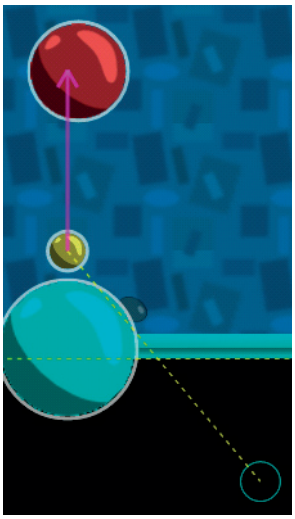
To cover all cases, the generator automatically splits all integer conditions into disjoint intervals. Therefore, the code in the if-statement over the intervals $[-\infty, -11]$ and $[-9, -1]$ describes the enemy going up, whereas the code in the if-statement over the interval $[0, +\infty[$ describes the enemy going down. The special singleton $\{-10\}$ contains the change of the counter to 10.

Furthermore, the ambiguity of the line of code transforming -10 to 10 is a problem solved by subtracting the current value of the counter from the score, which is currently equal to zero. Although correct for the moment, this formula will be wrong. One can manually modify it at this stage using arrows, or one can correct it later when the game will be more complex.

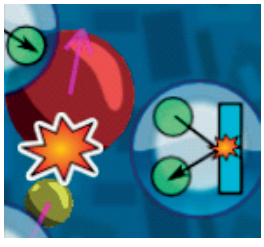
We next wish to make the player to follow the finger horizontally, as for the brick breaker game. We proceed as follows. Start the game. While watching the enemy going up and down, move the finger to drag the player's ball to the right. Pause the game. Press on the event menu, select the move event that just occurred, drag the ball to the right, and accept the rule.

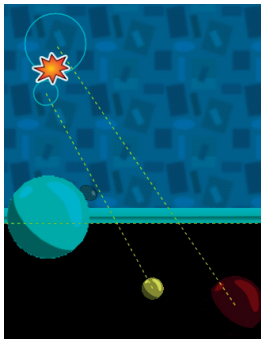
We next enable the player to shoot at the enemy when "Fire" is pressed. For that, we first create a small orange bullet anywhere outside the screen. Then we add the logic.

1. Start the game. Press the “Fire” button. Pause the game. Create a small orange bullet anywhere outside the screen. Press on the event menu, select the press on the “Fire” button. Move the bullet to the top of the player, sets its speed to the top. The system will infer that this circle’s new position should be relative to the player. Also, set the visibility of the “fire” button to invisible, since the player is not using more than a bullet a time.

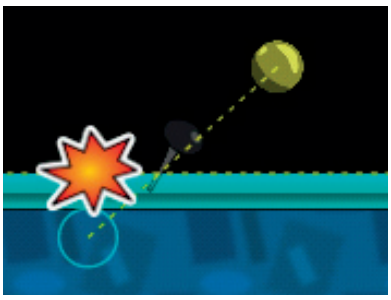


2. Start the game. The bullet hits the enemy. Pause the game, press on the event menu, select the collision, and change the game such that the score increases, the enemy and the bullet disappear, the score augments, and the fire button becomes visible.





3. What happens if the bullet does not hit any enemy? Rewind the game just before the circle hits the enemy, and move the enemy away. Start the game. After the bullet hits the top wall, pause the game. Select the event menu, select the collision, move the bullet away, set its speed to zero, and make the “Fire” button to reappear.



4. Let us add more enemies. Go back to the beginning. Duplicate enemies to make an array. For that, select an enemy and create a new circle. The system automatically duplicates rules and parameters.

Using techniques explained in the Breakout style game, it is straightforward to add a victory label, a game over, to have a specific enemy to fire other bullets at the player, etc.

In the sequel, we illustrate the expressiveness of Pong Designer for describing integer computations by showing how to compute the Fibonacci sequence. This also suggests that this programming paradigm can introduce programming concepts, regardless whether these concepts are related to games.

B.2 Fibonacci Sequence Tutorial

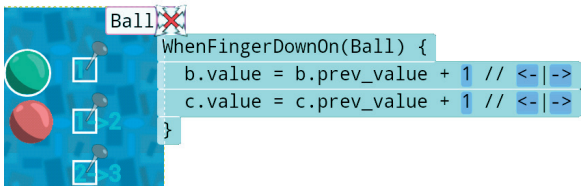
A user usually programs a Fibonacci sequence by explicitly providing the calculation steps. However, the Fibonacci sequence follows a simple recognizable pattern. Our game engine is able to recognize this pattern after a few input/output examples, and it computes the expected program.

Although it appears to use a sledgehammer to kill a fly, demonstrating the behavior instead of providing the code has several advantages:

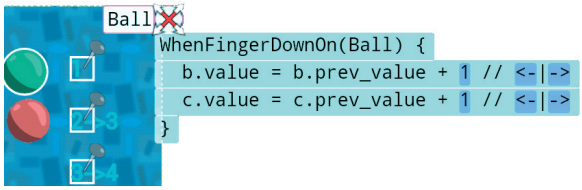
- The user is sure that the generated function complies with the provided examples.
- He can directly test the generated function on other inputs to check the generalization made by the system.
- He can demonstrate other non-trivial sequences such as the reverse Fibonacci sequence (see step 13) and indicate wrong behaviors.
- Finally, the system could be able to ask for disambiguation by providing differentiating examples in a future version.

The implementation of this "game" results in an interactive Fibonacci sequence. The interaction is the following: when the user presses the green ball that acts as a trigger, the game computes the next numbers in the Fibonacci sequence. When the user presses on the red ball, it computes the sequence in the reverse order.

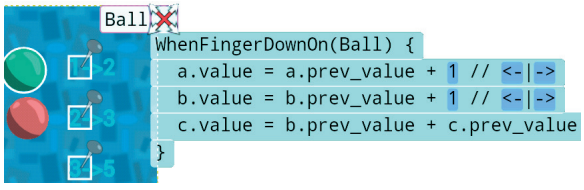
1. Insert three score counters into the game, and two ball shapes: one green and one red.
2. Set their respective values to 1, 1, 2.
3. Press the event menu, and press on the green ball. Select the finger down event.
4. Change the numbers to 1, 2, 3.
5. Select OK.
6. Open the rule.



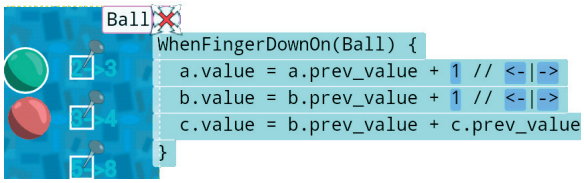
7. Apply the rule once. The numbers now become 1,2,3, but their next values 1,3,4 are wrong.



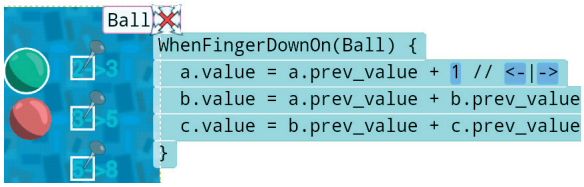
8. Modify the next values to 2,3,5, and press OK. The system immediately update the rule.



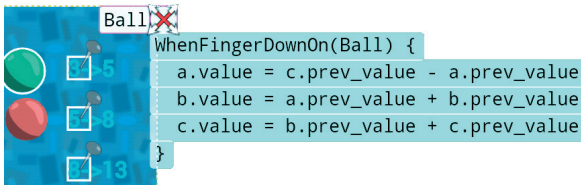
9. Apply the rule once. The numbers now become 2,3,5, but their next values 3,4,8 are still wrong.



10. Modify the next values to 3,5,8, and press OK. The system immediately update the rule. The formula for computing the second number is now correct.



11. Apply the rule once again, and change the 4,8,13 to 5,8,13, and accept it.



12. Run the game, press two times on the ball. The game correctly computes the Fibonacci numbers.

13. To create the rule for the reverse computation, the steps could be similar, but we will use a faster way. Press on the red ball and pause the game. Press on the event menu, select the press on the red ball, and change the numbers from 13,21,34 to 8,13,21. Accept the rule.

14. Launch the game. Because of the few possibilities, the system inferred the correct rule for the reverse Fibonacci sequence.

C User Study for Composite Scripting

The following pages 197-205 contain the user study for the work on Composite Scripting (See Chapter 3 page 47, especially section 3.6.1 page 68). We had two variants of this user study, one starting with exercises with our tool StriSynth and then continuing with Powershell, and the other doing it with Powershell first.

StriSynth

User Study A

1 Overview

StriSynth is a "programming by example" language for string transformation. By giving examples of the transformations you would like, StriSynth will generate a script (in this case a PowerShell script) that generalizes the pattern you have given.

This study will be used to determine which sorts of problems are most appropriate for StriSynth from the user's perspective. We will give you an introduction to StriSynth, highlighting both it's strengths and limitations. Then, you will complete a series of scripting tasks designed to explore how you prefer to use StriSynth. We expect this study to take approximately an hour.

Please do not close the terminal windows until you are completely finished, we are collecting data on your session and cannot save everything we need if you close the window in the middle of the study. You may disconnect from the RDP, or walk away from your computer for a time, as long as the PowerShell terminal remains open.

There are three sets of tasks - the first two will each be exclusive to either StriSynth or PowerShell. The third set you will not need to code, you will only be asked to choose between use StriSynth or PowerShell for each task.

By completing this study you grant us the rights to analyze any scripts or commands you write for this study. We will collect no personally identifying information on you for this study. You may decline to participate or complete the study at any time.

2 PowerShell Tutorial

PowerShell is an object based scripting language.

If you would like to retrieve the list of all the files in a directory, you can use 'ls'. This command will return a List (array) of objects, which you can then manipulate. For example, for this study you will often need only the name attribute from the objects. (ls | select -ExpandProperty Name) will return a List(String) of names. This list can be piped into StriSynth generated scripts, explained in the next section. To get the contents of the file in PowerShell use (Get-Contents), which will return List(String), where each element is a line from the file.

Here is a reference that gives powershell commands, equivalent bash commands, and an English explanation of the command. `http://cecs.wright.edu/~pmateti/Courses/233/Labs/Scripting/bashVsPowerShellTable.html`

3 StriSynth Tutorial

StriSynth is a "programming by example" language for string manipulation. To program in StriSynth you need to provide one or more examples of input to output. The StriSynth engine will then synthesize a script based on your given examples.

A tutorial video is available at `https://www.youtube.com/watch?v=GVvTdwulWPY`

Here is a in-depth tutorial that gives StriSynth commands and an English explanation of the command. `http://santolucito.github.io/strisynth/`

4 Tasks

You have been given login information to a remote Windows machine. Use the Remote Desktop viewer software of your choice to login. Once connected, open a PowerShell window to begin.

There will be a directory that matches your user identification number on this form. Within this directory you will find directories for each set of tasks (set1, set2). Set3 does not require coding. Within each of these directories, you will find directories for each task (set1/task1, set1/task2,...). Please complete all sets and tasks in numerical order.

You will need to be in the StringSolver directory to build scripts with StriSynth (sbt console). Once you have finished using StriSynth to make a script, you can use it in the the appropriate directory.

4.1 Set 1 - StriSynth

Please complete all these tasks with StriSynth.

If you feel you are not able to come to a solution for any of these problems after 8min, please move on. It is not important that you find the solutions to these problems, but rather that you gain a sense of what it is like to work with this tool.

Note that you are free to use the Internet, including copying and pasting. We hope to create a simulation of your typical decision procedure for finding a solution.

After finishing each task, please **make a note** how much time you took. You will be asked to enter this time into a survey given at the end of the study. Do not take more than 25 minutes on this set.

4.1.1 Linking to items

Create a script (solution.ps1) to generate a link to all the documents in the folder. That is, `"test.doc"` should become `"<A HREF="test.doc">test</A>"`.

4.1.2 Printing pdfs

You are given a list of files of coma-separated file names in the file list.txt. Create a script (solution.ps1) that will print all .pdf files using the imaginary 'print <filename>' command. Do not print any other files extensions.

4.1.3 Merging a book

You are given the pages of three books, Algorithms, Graphics, and Math. Create a script (solution.ps1) to merge all the pages (existing .pdf files) to a new .pdf with the name of the book using the command 'merge <list of in files> <merged filename>. Math1.pdf and Math2.pdf should be combined to make Math.pdf using the command 'merge Math1.pdf Math2.pdf Math.pdf'.

4.2 Set 2 - PowerShell

Please complete all these tasks with PowerShell.

If you feel you are not able to come to a solution for any of these problems after 8min, please move on. It is not important that you find the solutions to these problems, but rather that you gain a sense of what it is like to work with this tool.

Note that you are free to use the Internet, including copying and pasting. We hope to create a simulation of your typical decision procedure for finding a solution.

After finishing each task, please **make a note** how much time you took. You will be asked to enter this time into a survey given at the end of the study. Do not take more than 25 minutes on this set.

4.2.1 Linking to items

Create a script (solution.ps1) to generate a link to all the documents in the folder. That is, `"test.doc"` should become `"<A HREF="test.doc">test</A>"`.

4.2.2 Printing pdfs

You are given a list of files of coma-separated file names in the file list.txt. Create a script (solution.ps1) that will print all .pdf files using the imaginary 'print <filename>' command. Do not print any other files extensions.

4.2.3 Merging a book

You are given the pages of three books, Algorithms, Graphics, and Math. Create a script (solution.ps1) to merge all the pages (existing .pdf files) to a new .pdf with the name of the book using the command 'merge <list of in files> <merged filename>. Math1.pdf and Math2.pdf should be combined to make Math.pdf using the command 'merge Math1.pdf Math2.pdf Math.pdf'.

4.3 Set 3 - Your Choice

This set will be completed concurrently with the survey. You will be asked to rank your preference for tool to use to complete each task. Your options are StriSynth, PowerShell, and any other scripting language (bash, perl, sed, etc). Both approaches are possible for every task, however you will not be asked to actually code the solution. We just want to know which tool you would choose for each task if you were to code a solution.

Note that you would be free to use the Internet, including copying and pasting. We hope that you will imagine this as a simulation of your typical decision procedure for finding a solution.

4.3.1 Automated Emails

Please create a script (solution.ps1) to send emails to 'president@yale.edu'. Send as many emails as there are lines in the input file.

The command to send an email is `"Send-Email <address> \# <subject> \# <message>"`. The message of the email will be one line taken from the file 'in.txt' and the subject will just be the string 'test'. For example, if 'in.txt' contains the `"foobar"`, you should generate the command...

`"Send-Email president@yale.edu # test # foobar"`

4.3.2 Health Data

Write a script to extract data from a patient health records.

The input format is below. The numbers indicate name; age; diseases; weight; height; and blood pressure.

```
Patient #091234; Alice Smith; 24; Genital Herpes ; 71.2; 180; 130/80
Patient #4155; Bob Brown; Asthma; 9; 35.8; 100; 120/78
Patient #18535; Evelyn Doyle; HPV; 19,64.4,150,120/79
```

For all patients that are 10 years or older, you need to extract the patients' diseases. Since this will be a publicly available database, you need to be sure to use the patient id rather than the name. The output format is below.

```
#091234 has Genital Herpes
#18535 has Asthma
```

4.3.3 Reorder commands

You have a program, p.out, that you want to run on many files. p.out will read two files and write to two other files. The commands have already been constructed, but the ordering of the arguments has changed for p.out. If a command looked this this before the change

```
./p.out in1.txt in2.txt out1.txt out2.txt;
```

now it should look like

```
./p.out in1.txt out1.txt in2.txt out2.txt;
```

The input output files are all .txt file, but otherwise have no pattern. The list of commands is given you to as a file shown below.

```
./p.out foo.txt bar.txt f.txt b.txt;
./p.out a.txt b.txt c.txt d.txt;
./p.out ying.txt yang.txt ting.txt tang.txt;
```

4.3.4 Report Building

You are given a text file (in.txt) that contains a bank account history. Please generate a new organized report in the below format. To get the contents of the file in PowerShell use 'Get-Contents', which will return List(String), where each element is a line from the file. So, if the input report looks like...

```
20061120_Yale +100
20150119_MSR +100
20100315_MSR +20
19980905_Yale -10
```

the output report (out.txt) should look like...

```
Yale-report +100 -10
MSR-report +100 +20
```

4.3.5 Extract Filenames

Write a script to make a list of the files in a folder. The list should contain the filename without the extension. For example, the following directory...

```
foo.txt
bar.pdf
foo.bar.doc
```

will yield `[foo,bar,foo.bar]`

4.3.6 Initials

You will need to convert a number of "Full Name" data fields to a common format. Here are the input formats.

```
FirstName MiddleName LastName
FirstName M. LastName
LastName, FirstName MiddleName
LastName, FirstName M.
FirstName LastName
LastName, FirstName
```

The target format is the user's initials and a unique counter (`John Abe Doe becomes J.A.D.-01`). The order of the counter doesn't matter, only that each item has a unique value.

4.3.7 Move

Move all the `.png` files to the `imgs/` subdirectory.

4.3.8 Art Project

Write a script that will work as part of a digital art project.

You will collect data from the location services platform, 3Triangle, to be sent to a visualizer. The 3Triangle API provides the following data.

```
last checkins-(41.316070, -72.943445),(41.311090, -72.932008); Username-james9867; Device-Windows
last checkins-(41.316470, -72.943040); Username-jane1995; Device-Windows
last checkins-(41.316588, -72.943467),(41.311348, -72.932065),(41.315090, -72.932008); Username-joey24; Dev
```

Create a list of all the locations that have been recorded by Android phones, and a list of all the locations that have been recorded by Windows phones

```
(41.316070, -72.943445),(41.311090, -72.932008), (41.316470, -72.943040)
```

4.3.9 Linking to items

Create a script (solution.ps1) to generate a link to all the documents in the folder. That is, `"test.doc"` should become `"<A HREF="test.doc">test</A>"`.

4.3.10 Printing pdfs

You are given a list of files of coma-separated file names in the file list.txt. Create a script (solution.ps1) that will print all .pdf files using the imaginary 'print <filename>' command. Do not print any other files extensions.

4.3.11 Merging a book

You are given the pages of three books, Algorithms, Graphics, and Math. Create a script (solution.ps1) to merge all the pages (existing .pdf files) to a new .pdf with the name of the book using the command 'merge <list of in files> <merged filename>. Math1.pdf and Math2.pdf should be combined to make Math.pdf using the command 'merge Math1.pdf Math2.pdf Math.pdf'.

5 End of Study

Now that you have completed all the tasks please save your terminal history with the following command. In the window you have been using for PowerShell, run `"h -count 500 > ~/history.txt"`. Your StringSolver command history has already been automatically captured.

5.1 Writeup

Please fill out the short survey at the following address `https://yalesurvey.qualtrics.com/SE/?SID=SV_afv1UNqr1LLtAMZ`.

You may now log out of the Remote Desktop connection. Thank you for participating in this study. If you have any questions or would like to know more about this research, please contact your recruiter.

D The Three Coins Proof

It was useful to rely on automated tools for verification and proof during the work of Chapter 5, page 109.

One of this automated proofs is particularly interesting. It proves the fact stated by the sandwich story on page 112 that testing up to three coins is enough. Reproducing the proof as a formally verified program would be 16 pages long in this thesis format[1]. The complexity could be reduced if trivial operations, such as the associativity of concatenation, were built-in by default. I used the Leon[2] tool to prove this theorem.

In the sequel, I present a still formal but readable version of the proof after an introduction.

D.1 Code Coverage Is Not Enough

Let us consider the following type hierarchy and the most simple template f to render it:

```
abstract class List
case class Cons(e: List) extends List
case class Nil() extends List

def f(l: List) = l match {
  case c: Cons ⇒ C1 + f(c.e) + C2
  case n: Nil ⇒ C3
}
```

If we were to learn f by example, it would not be sufficient to have input examples covering the entire program, in order to guess the value of the free variables C_1, C_2 and C_3. Indeed, a code coverage procedure might produce the outputs Cons(Nil()) and maybe even Nil(). However, the following functions are equal on these two terms and produce “”

[1] github.com/MikaelMayer/string-proofs/blob/master/StringTheorems.scala
[2] https://github.com/epfl-lara/leon

and "ab", but differ afterwards because f_1 produces "aabb" whereas f_2 produces "abab".

```
def f1(l: List) = l match {
  case c: Cons ⇒ "a" + f(c.e) + "b"
  case n: Nil  ⇒ ""
}
def f2(l: List) = l match {
  case c: Cons ⇒ "ab" + f(c.e) + ""
  case n: Nil  ⇒ ""
}
```

D.2 Characterizing Unary Lists Printers

We now prove the following statement, which could be used to finish an active learning phase for a hierarchy of unary lists by asking at most three questions. Given two functions of the shape given by f in appendix D.1, they are equal if and only if they are equal on $Exs = \{\mathsf{Nil}(), \mathsf{Cons}(\mathsf{Nil}()), \mathsf{Cons}(\mathsf{Cons}(\mathsf{Nil}()))\}$. In the sandwich metaphor of Chapter 5, page 118, this means that we need to test the machines with up to three coins.

Proof. For that, let us first define a few lemmas on strings (we use the multiplication notation for the concatenation of strings).

Lemma 1. (Equation split) If A, B, C, D are strings, then:

$$AB = CD \wedge |A| = |C| \Leftrightarrow A = C \wedge B = D$$

Lemma 2. (Inegality-propagate) If p, A, q, B are strings, then:

$$(AB = CD \wedge |A| < |C|) \Leftrightarrow (AB = CD \wedge |B| < |D|)$$

Lemma 3. (Prefix-introduce) If p, A, q, B are strings, then:

$$(|p|` < |q| \wedge pA = qB) \Leftrightarrow (\exists k.q = pk \wedge A = kB)$$

Lemma 4. (Suffix-introduce) If p, A, q, B are strings, then:

$$(|p|` < |q| \wedge Ap = Bq) \Leftrightarrow (\exists k.q = kp \wedge A = Bk)$$

Lemma 5. (constructive-commutation) If C, q, p are strings, then

$$Cq = pC \Leftrightarrow$$
$$\exists k_1, k_2 : \text{string}, n \in \mathbb{N}.p = k_1k_2, q = k_2k_1, C = k_1(k_2k_1)^n$$

Lemma 6. (commutative-recurrence) If A, B are strings and n an integer such that $AB = BA$, then $(AB)^n = A^nB^n$

Lemma 7. (associativity) If A, B are strings and n an integer, then $(AB)^n A = A(BA)^n$

Let us have two functions f and g obtained from the template `f` in appendix D.1, the first one being parameterized by the constants A, B and C and the second by D, E and F.

The fact that f and g are equal on Exs yields the three equations:

$$C = F \tag{D.1}$$

$$ACB = DFE \tag{D.2}$$

$$AACBB = DDFEE \tag{D.3}$$

If $|A| = |D|$ it follows that $|B| = |E|$ and from lemma 1 we can deduce that the functions are equal.

Without loss of generality, let us suppose that $|A| < |D|$. From equation D.2 and lemma 2, we obtain that $|B| > |E|$. Hence from lemma 3 and lemma 4, we obtain the existence of two constants p and q such that $D = Ap$ and $B = qE$. Hence equations D.2 and D.3 can be rewritten and simplified as:

$$Cq = pC \tag{D.4}$$

$$ACqEq = pApCE \tag{D.5}$$

Applying lemma 5 to equation D.4 produces constants k_1, k_2 and n as in the lemma making euqation D.4 to hold and rewriting equation D.5 to:

$$Ak_1(k_2k_1)^n k_2k_1Ek_2k_1 = k_1k_2Ak_1k_2k_1(k_2k_1)^n E \tag{D.6}$$

If $n = 0$, then

$$Ak_1k_2k_1Ek_2k_1 = k_1k_2Ak_1k_2k_1E \tag{D.7}$$

and it follows from the fact that $|Ak_1k_2| = |k_1k_2A|$, $|Ek_2k_1| = |k_2k_1E|$ and lemma 1 used

twice that:

$$Ak_1k_2 = k_1k_2A \wedge Ek_2k_1 = k_2k_1E \tag{D.8}$$

A similar reasoning for $n > 0$ would yield the same equation D.8.

To conclude, we remark that, if we abbreviate an input $\mathsf{Cons}(\mathsf{Cons}(\ldots\mathsf{Nil}()))$ by $\mathsf{Cons}^m(\mathsf{Nil}())$ where m is the number of Cons in the expression, invoking the lemma 6 and 7:

$$\begin{aligned}
f(\mathsf{Cons}^m(\mathsf{Nil}())) &= A^mCB^m \\
&= A^mk_1(k_2k_1)^n(qE)^m \\
&= A^mk_1(k_2k_1)^n(k_2k_1E)^m \\
&= A^mk_1(k_2k_1)^n(k_2k_1)^mE^m \\
&= A^mk_1(k_2k_1)^n(k_2k_1)^mE^m \\
&= A^mk_1(k_2k_1)^m(k_2k_1)^nE^m \\
&= A^m(k_1k_2)^mk_1(k_2k_1)^nE^m \\
&= A^m(k_1k_2)^mCE^m \\
&= A^m(k_1k_2)^mFE^m \\
&= (Ak_1k_2)^mFE^m \\
&= D^mFE^m \\
&= g(\mathsf{Cons}^m(\mathsf{Nil}()))
\end{aligned}$$

So we proved that f and g are equivalent. □

E Proactive Synthesis Proofs

E.1 Injectivity of τ_Σ

Lemma 5.5.1.1. *For any ranked alphabet Σ, the function $[\![\tau_\Sigma]\!]$ is injective.*

Proof. Assume that $[\![\tau_\Sigma]\!]$ is not injective, and let $t = f(t_1, \ldots, t_n)$ and $t' = f'(t'_1, \ldots, t'_m)$ be two trees with $t \neq t'$, such that $\tau_\Sigma(t) = \tau_\Sigma(t')$. We pick t and t' satisfying those conditions such that $\tau_\Sigma(t)$ has the smallest possible length.

By definition of τ_Σ, we have

$$\tau_\Sigma(t) = (f, 0)\tau_\Sigma(t_1)(f, 1)\cdots\tau_\Sigma(t_n)(f, n)$$

and

$$\tau_\Sigma(t') = (f', 0)\tau_\Sigma(t'_1)(f', 1)\cdots\tau_\Sigma(t'_m)(f', m).$$

Since $\tau_\Sigma(t) = \tau_\Sigma(t')$, we deduce that $(f, 0) = (f', 0)$ and $f = f'$, meaning that t and t' have the same root.

Thus, $t = f(t_1, \ldots, t_n)$, and $t' = f(t'_1, \ldots, t'_n)$.

To conclude, we consider two cases. If for all $i \in \{1, \ldots, n\}$, $\tau_\Sigma(t_i) = \tau_\Sigma(t'_i)$, we have $t_i = t'_i$, as the length of $\tau_\Sigma(t_i)$ is strictly smaller than $\tau_\Sigma(t)$. This ensures that $t = t'$, and we obtain a contradiction.

On the other hand, if there exists $i \in \{1, \ldots, n\}$, $\tau_\Sigma(t_i) \neq \tau_\Sigma(t'_i)$, consider the smallest such i. We then have:

$$\tau_\Sigma(t) = (f, 0)\tau_\Sigma(t_1)(f, 1)\cdots\tau_\Sigma(t_{i-1})(f, i-1)\tau_\Sigma(t_i)(f, i)\cdots\tau_\Sigma(t_n)(f, n)$$

and

$$\tau_\Sigma(t') = (f,0)\tau_\Sigma(t_1)(f,1)\cdots\tau_\Sigma(t_{i-1})(f,i-1)\tau_\Sigma(t'_i)(f,i)\cdots\tau_\Sigma(t'_n)(f,n).$$

Since $\tau_\Sigma(t) = \tau_\Sigma(t')$, and the prefixes are identical up until $(f, i-1)$, we deduce

$$\tau_\Sigma(t_i)(f,i)\cdots\tau_\Sigma(t_n)(f,n) = \tau_\Sigma(t'_i)(f,i)\cdots\tau_\Sigma(t'_n)(f,n).$$

We finally consider three subcases, with respect to this last equation.

- If $\tau_\Sigma(t_i)$ and $\tau_\Sigma(t'_i)$ have the same length, we deduce $\tau_\Sigma(t_i) = \tau_\Sigma(t'_i)$, contradicting our assumption.
- If $\tau_\Sigma(t_i)$ is strictly shorter than $\tau_\Sigma(t'_i)$, we deduce that $\tau_\Sigma(t_i)(f,i)$ is a prefix of $\tau_\Sigma(t'_i)$. This is not possible as $\tau_\Sigma(t'_i)$ must be well parenthesized if $(f,0)$ is seen as an open parenthesis, and (f,i) as a closing parenthesis (by definition of τ_Σ).
- The case where $\tau_\Sigma(t_i)$ is strictly longer than $\tau_\Sigma(t'_i)$ is symmetrical to the previous one.

□

E.2 Proof of NP-completeness

Theorem 5.6.1.1. *Given a sample $\mathcal{S}$, checking whether there exists a* 1STS *τ such that for all $(t,w) \in \mathcal{S}$, $\tau(t) = w$ is an* NP*-complete problem.*

Proof. In general, we can check for the existence of τ in NP using the following idea. Every input/output example from the sample gives constraints on the constants of τ. Therefore, to check for the existence of τ, it is sufficient to non-deterministically guess constants which are subwords of the given output examples. We can then verify in polynomial-time whether the guessed constants form a 1STS τ which is consistent with the sample $\mathcal{S}$.

To prove NP-hardness, we consider a formula φ, instance of the one-in-three positive SAT. The formula φ has no negated variables, and is satisfiable if there exists an assignment to the boolean variables such that for each clause of φ, exactly one variable evaluates to true.

Formally, let $\mathbb{X}$ be a set of variables and let $\varphi \equiv C_1 \wedge \cdots \wedge C_n$ such that for every $i \in \{1,\ldots,n\}$, $C_i \equiv \mathsf{OneInThree}(x^i_1, x^i_2, x^i_3)$ with $x^i_1, x^i_2, x^i_3 \in \mathbb{X}$.

Let $\Sigma = \mathsf{nil}^{(0)} \cup \{x^{(1)} \mid x \in \mathbb{X}\}$. Let $\Gamma = \{a, \#\}$. Then, for every clause $\mathsf{OneInThree}(x_1^i, x_2^i, x_3^i)$, we define $\mathcal{S}(x_1^i(x_2^i(x_3^i(\mathsf{nil})))) = a\#$. Finally, we define $\mathcal{S}(\mathsf{nil}) = \#$.

We now prove the following equivalence. There exists a $\mathsf{1STS}$ τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ if and only if φ is satisfiable.

($\Rightarrow$) Let $\tau = (\Sigma, \Gamma, \delta)$ be a $\mathsf{1STS}$ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ By definition of $\mathcal{S}$, we know $\tau(\mathsf{nil}) = \#$ and and for all $i \in \{1, \ldots, n\}$, $\tau(x_1^i(x_2^i(x_3^i(\mathsf{nil})))) = a\#$.

Moreover, if for all $x \in \mathbb{X}$, we denote $\delta(x) = (\mathsf{left}(x), \mathsf{right}(x))$, with $\mathsf{left}(x), \mathsf{right}(x) \in \Gamma^*$. Then, by definition of τ, we have, for $i \in \{1, \ldots, n\}$:

$$\begin{aligned}\tau(x_1^i(x_2^i(x_3^i(\mathsf{nil})))) &= \mathsf{left}(x_1^i)\mathsf{left}(x_2^i)\mathsf{left}(x_3^i)\tau(\mathsf{nil})\mathsf{right}(x_3^i)\mathsf{right}(x_2^i)\mathsf{right}(x_1^i)\\ &= \mathsf{left}(x_1^i)\mathsf{left}(x_2^i)\mathsf{left}(x_3^i)\#\mathsf{right}(x_3^i)\mathsf{right}(x_2^i)\mathsf{right}(x_1^i)\end{aligned}$$

We deduce that $\mathsf{left}(x_1^i)\mathsf{left}(x_2^i)\mathsf{left}(x_3^i) = a$ and $\mathsf{right}(x_3^i)\mathsf{right}(x_2^i)\mathsf{right}(x_1^i) = \varepsilon$.

Thus, exactly one of $\mathsf{left}(x_1^i)$, $\mathsf{left}(x_2^i)$, $\mathsf{left}(x_3^i)$ must be equal to $a \in \Gamma$, while the other two must be equal to ε. Then, φ is satisfiable using the boolean assignment that maps a variable $x \in \mathbb{X}$ to $\top$ if $\mathsf{left}(x) = a$, and to $\bot$ if $\mathsf{left}(x) = \varepsilon$.

($\Leftarrow$) Conversely, assume there exists a satisfying assignment $\mu : \mathbb{X} \to \{\bot, \top\}$ for φ. Then, we define the $\mathsf{1STS}$ $\tau = (\Sigma, \Gamma, \delta)$ where $\delta(\mathsf{nil}) = \#$ and for all $x \in \mathbb{X}$ $\delta(x) = (a, \varepsilon)$ if $\mu(x) = \top$, and $\delta(x) = (\varepsilon, \varepsilon)$ if $\mu(x) = \bot$. We then have $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$. □

Remark. The NP-completeness proof of [Laurence, 2014] could not apply here, because the transducers are more general. Namely, they are allowed to have multiple states in their setting.

E.3 Cubic Lower Bound

Lemma 5.7.2.2. *There exists a sequence of domains $D_1, D_2, \ldots$ such that for every $n \geq 1$, the smallest tree test set of D_n has at least n^3 elements, and the size of D_n is linear in n. Furthermore, this lower bound holds even with the extra assumption that all states of the domain are initial.*

Proof. Our proof is inspired by the lower bound proof for test sets of context-free languages [Plandowski, 1994, Plandowski, 1995]. However, that lower bound did not work for context-free grammars with the extra assumption that all non-terminal symbols are starting symbols. Therefore, their proof cannot be applied for domains where all states are initial. Our contribution is a variant which shows that, even when all states of the domain are initial, the lower bound still holds and the minimal test-set has a cubic size.

For $n \geq 1$, we first define the domain $D_n = (\Sigma, Q, I = Q, \delta)$ containing linear trees (lists) of depth 1 to 3, and using n different symbols of each level. Formally, we have (we use a functional notation for δ, as δ is here deterministic):

- $\Sigma = \{{A_j}^{(1)}, {B_j}^{(1)}, {F_j}^{(0)} \mid 1 \leq j \leq n\}$,
- $Q = \{q_2, q_1, q_0\}$,
- $\delta({A_j}^{(1)}, q_2) = (q_1)$,
- $\delta({B_j}^{(1)}, q_1) = (q_0)$,
- $\delta({F_j}^{(0)}, q_0) = ()$.

D_n recognizes $n^3 + n^2 + n$ trees. Our goal is to prove, by contradiction, that D_n does not have a tree test set $T \subset D_n$ of size less than n^3. Let $t = A_x(B_y(F_z)) \in D_n \setminus T$ for some arbitrary $x, y, z \in [1, n]$. t exists when the size of T is strictly less than n^3. We construct two 1STSs τ_1 and τ_2 such that $[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T}$ but $[\![\tau_1]\!](t) \neq [\![\tau_2]\!](t)$, contradicting the fact that T is a tree test set.

Let $\Gamma = \{p, q\}$ be an alphabet, and $\tau_1 = (\Sigma, \Gamma, \delta_1)$, $\tau_2 = (\Sigma, \Gamma, \delta_2)$, where

- $\delta_1(A_j) = (\varepsilon, pq)$ if $j = x$,
- $\delta_1(A_j) = (\varepsilon, \varepsilon)$ otherwise,
- $\delta_1(B_j) = (\varepsilon, \varepsilon)$ if $j = y$,
- $\delta_1(B_j) = (p, q)$ otherwise,
- $\delta_1(F_j) = (qp)$ if $j = z$,
- $\delta_1(F_j) = (\varepsilon)$ otherwise,

and

- $\delta_2(A_j) = (pq, \varepsilon)$ if $j = x$,
- $\delta_2(A_j) = (\varepsilon, \varepsilon)$ otherwise,
- $\delta_2(B_j) = (\varepsilon, \varepsilon)$ if $j = y$,
- $\delta_2(B_j) = (p, q)$ otherwise,
- $\delta_2(F_j) = (qp)$ if $j = z$,
- $\delta_2(F_j) = (\varepsilon)$ otherwise.

We can verify that $[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T}$, but $\tau_1(t) \neq \tau_2(t)$, as $\tau_1(t) = pqqp$ and $\tau_2(t) = qppq$.

We conclude that the only tree test set of D_n is D_n itself, which contains $n^3 = (\frac{|\Sigma|}{3})^3$ words. Moreover, the (syntactic) size of D_n is $O(n)$. □

We note that the above transducers δ_1 and δ_2 have the same output on all F_k and on all $B_j(F_k)$. Therefore, even if we interactively ask questions as for Theorem 5.8.1.1, these questions will not be able to resolve the ambiguity which will appear only at the specific $A_x(B_y(F_z))$.

E.4 Construction of $\Phi_3(G)$

To construct $\Phi_3(G)$ for a linear context-free grammar $G = (N, \Sigma, R, S)$, we precompute in time $O(|N|^2|R|)$, for each pair of vertices (A, B), the optimal path from A to B in $\mathsf{graph}(G)$. Then for each possible choice of at most 3 edges $e_1 = (A_1, r_1, B_1), \dots e_n = (A_n, r_n, B_n)$, with $0 \leq n \leq 3$, we construct the path $P = P_1 e_1 \dots P_n e_n P_{n+1}$ where each P_i is the optimal path from A_{i-1} to B_i (if it exists) with $A_0 = S$ and $B_{n+1} = \bot$ by convention. We then add the word corresponding to P to our result.

To conclude, since the length of each optimal path is bounded by $|N|$, we can construct $\Phi_3(G)$ in time $O(|N| \cdot |R|^3)$.

E.5 Proof of Theorem 5.9.1.1

Theorem 5.9.1.1. *If the domain $D = (\Sigma, Q, I, \delta)$ is such that for every $f \in \Sigma$ of arity $k > 0$, there exist trees in $t_1, \dots, t_k \in D$ such that $f(t_1, \dots, t_k) \in D$ and each t_i contains at least one v, then there exists a tree test set of D' of linear size $O(|\Sigma| \cdot A)$ where A is the maximal arity of a symbol of Σ.*

Proof. Let D be a domain with the property above. First, remark that for some $s, u \in \Gamma^*$, if $t' \in D'$ and t' contains v_s, then all the trees obtained from t' by replacing v_s by v_u are also in D'.

We build the linear tree test set T' as follows.

Let us associate to every symbol $f^{(k)} \in \Sigma$ and to every position in the arity $i \in [1, k]$ the tree t_i as provided by the hypothesis. Let us take one of its $t'_i \in D'$ (denoted $f^i[s]$) such that t'_i contains at least one v_s for some $s \in \Gamma^*$, the other v being mapped to any other constant. For all other $u \in \Gamma^*$, remark that $f^i[u] \in D'$.

Define T' containing all "default" trees and all their variations, changing one "#" to "?":

$$\mathrm{T}' = \bigcup_{f^{(k)} \in \Sigma} \left(\{ f^{(k)}(f^{i}[\text{“\#”}]_{i \in [1,k]}) \} \cup \{ f^{(k)}(f^{i}\left[\begin{cases} \text{“?”} & \text{if } i = j \\ \text{“\#”} & \text{else} \end{cases} \right]_{i \in [1,k]}) | j \in [1,k] \} \right)$$

.

Note that T' is included in D', since the original tree was in D'.

The size of the set T' is at most $2|\Sigma| \cdot A$. We will now prove that T' is a tree test set for D'. Indeed, suppose that we have two transducers τ_1, and τ_2, and that they are equal on T'. We will show the strong result that they have the *same constants*.

Let $f^{(k)} \in \Sigma$. Since $f^{i}[\text{“\#”}], f^{i}[\text{“?”}] \in \mathrm{T}'$ for $i \in [1,k]$, we can write

$$[\![\tau_1]\!](f^{i}[\text{“\#”}]) = [\![\tau_2]\!](f^{i}[\text{“\#”}]) \quad \wedge \quad [\![\tau_1]\!](f^{i}[\text{“?”}]) = [\![\tau_2]\!](f^{i}[\text{“?”}])$$

which, for some w_i and z_i can be rewritten to:

$$w_i \cdot \text{“\#”} \cdot z_i = w_i' \cdot \text{“\#”} \cdot z_i' \quad \wedge \quad w_i \cdot \text{“?”} \cdot z_i = w_i' \cdot \text{“?”} \cdot z_i'$$

We apply Lemma 5.9.1.1 to conclude that $w_i = w_i'$ and $z_i = z_i'$, so we can remove the primes.

By hypothesis, we have that:

$$[\![\tau_1]\!](f^{(k)}(f^{i}[\text{“\#”}]_{i \in [1,k]})) = [\![\tau_2]\!](f^{(k)}(f^{i}[\text{“\#”}]_{i \in [1,k]}))$$

which we can rewrite to:

$$u_0 w_1 \text{“\#”} z_1 \cdot u_1 \cdots w_k \text{“\#”} z_k \cdot u_k = u_0' w_1 \text{“\#”} z_1 \cdot u_1' \cdots w_k \text{“\#”} z_k \cdot u_k'$$

We also have k other similar equalities by changing any of the “#” by a “?” at the same place in the two sides of the equation. Using Lemma 5.9.1.1, and by changing the first “#” to “?”, we obtain that $u_0 w_1 = u_0' w_1$ so $u_0 = u_0'$. After simplifying the equations, it remains that:

$$u_1 \cdots w_k \text{“\#”} z_k \cdot u_k = u_1' \cdots w_k \text{“\#”} z_k \cdot u_k'$$

so we can continuously apply the lemma to obtain that $u_1 = u_1', \ldots u_k = u_k'$. Hence T' is a tree test set of linear size. □

Mikaël Mayer

Born July 1st, 1986
Sex: M
Nationality: French

StackOverflow ID 1287856
Github ID MikaelMayer
Google Scholar ID 56QdR7kAAAAJ

Education

Sep. 2012 – Aug. 2017	**PhD in Computer Science –** EPFL, Lausanne, Switzerland "Interactive Programming by Example"	Viktor Kuncak
Sep. 2008 – Jan. 2010	**M.Sc. in Computer Science** – EPFL, Lausanne, Switzerland "Complete Program Synthesis for Linear Arithmetics"	Viktor Kuncak Ruzica Piskac
Aug. 2005 – Jan. 2010 (Aug. 2008)	**École Polytechnique – X (Promotion 2005)** – Palaiseau, France M. Sc. Diplôme de l'École Polytechnique, specialized in Computer Science M. Sc. Ingénieur de l'École Polytechnique	
Sep. 2003 – April. 2005	Classe Préparatoire MPSI – MP* – Lycée Kléber, Strasbourg, France	
Jun. 2003	Baccalaureate in Sciences (Summa Cum Laude, mathematics) – Belfort, France	

Employment History

Sep. 2012 – Aug. 2017	**Teaching Assistant** – EPFL, Lausanne, Switzerland *Functional Programming in Scala, Compiler Construction* *Parallel and concurrent programming, Complex Analysis IV*	Viktor Kuncak *Martin Odersky* *Boris Buffoni*
Jun. 2014 – Jun. 2015	**Microsoft Research Scientist** – (remotely) Belfort, France *User interaction models for programming by example*	Sumit Gulwani
Feb. 2012 – Sep. 2012	**Upbraining Mobile Software Engineer** - Belfort, France *R&D director*	Christine Mayer
Feb. 2010 – Feb. 2012	**LDS Voluntary Humanitarian Mission** – France and Switzerland *HR manager, team coaching, service projects, family strengthening*	Michel Carter Kent Murdock
Sep. 2009 – Dec. 2009	**Biologically Inspired Robotic Group** – Lausanne, Switzerland *Semester project, Dynamic modeling in C++*	Alexander Spröwitz Auke Ijspeert
Apr. 2008 – Aug. 2008	**Google Software Engineer Intern** – Dublin, Ireland *Memory leaks detection, auto-complete for Google Help*	Marcos Campal
Jul. 2007 – Aug. 2007	**Andes Fertiles Humanitarian Mission** – La Paz / Viacha, Bolivia *Water-pumping windmill design and construction*	Virginie Genevois
Oct. 2005 – Apr. 2006	**La main à la pâte** - National Education Program for Primary schools *Experimental science school teacher assistant*	Nadine Sire
Feb 2004	**Research internship** – Institut Charles Sadron, Strasbourg, France *Crushing radius measures of giant phospholipids vesicle. (Caml)*	André Schröder

Institutional Responsibilities

Sep. 2015 – Dec. 2015	**Study Advisor**– Master in Computer Science	Sylviane Dal Mas
Sep. 2016 – Dec 2016	**Study Advisor**– Master in Computer Science	Sylviane Dal Mas

Approved Research Projects

Jun. 2014 – Jun. 2015 **Microsoft Research Scientist** – (remotely) Belfort, France — Sumit Gulwani
User interaction models for programming by example

Apr. 2008 – Aug. 2008 **Google Software Engineer Intern** – Dublin, Ireland — Marcos Campal
Memory leaks detection, auto-complete for Google Help

Supervision of Junior Researchers

Feb. 2013 – Jul. 2013 **Lomig Megard**, *Pong Designer Theoretical Report*
EPFL, Master semester project

Feb. 2014 – Jul. 2014 **Lomig Megard**, *Programming matrix rules by example in a game development system*
EPFL, Master thesis

Feb. 2014 – Jul. 2014 **Thomas Dupriez**, *Webpages bidirectionally Synchronized With Programs*
ENS Cachan, Master semester project

Teaching Activities

Feb.. 2013 – June. 2013 **Complex Analysis - 2nd year of bachelor** — Boris Buffoni

Sep. 2013 – Dec. 2013 **Compiler Construction Teaching Assistant** — Viktor Kuncak

Feb.. 2014 – June. 2014 **Complex Analysis - 2nd year of bachelor** — Boris Buffoni

Sept. 2015 – Dec. 2015 **Functional Programming Principles in Scala** — Martin Odersky, Viktor Kuncak

Feb.. 2015 – June. 2015 **Concurrent and Parallel Programming** — Martin Odersky, Viktor Kuncak

Sept. 2016 – Dec. 2016 **Functional Programming Principles in Scala** — Martin Odersky, Viktor Kuncak

Membership in panels, boards, etc., and individual scientific reviewing activities

Jul. 2016 **Reviewer - International Conference of Mathematic Education** Hamburg, Germany

Conference Organizations

Jul. 2017 **Publicity Chair of Computer-Aided Verification 2017** Heidelberg, Germany

Rewards, Scholarships

Jan. 2010 **"Elca Reward" for the best master grade** in Computer Science (5.77 / 6)

Sep. 2009 **Scolarship of Excellence** – EPFL, Lausanne, Suisse

Jul. 2003 **1st national certificate of merit** for the Concours Général des Lycées in Mathematics

Career breaks

Feb. 2010/12 **LDS Voluntary Humanitarian Mission**
This mission taught me some of the most important lessons in life, as I helped others to reinforce their self-esteem, to strengthen their families, and especially I was sharing my life with a colleague 24h/24.

Major scientific achievements

Proactive synthesizer of tree-to-string functions (2017) https://goo.gl/NwORok
I Implemented algorithms for learning tree-to-string functions by example in polynomial time with a linear number of questions. I wrote the parser, participated in the algorithm elaboration, and made the interaction failure-free. The underlying theoretical work may have a great impact on how to design programming-by-example features in IDEs.

WebBuilder (2016) http://leondev.epfl.ch
I integrated the work of Thomas Dupriez into a web interface. I created several presentations (https://github.com/MikaelMayer/leon-web-builder), including slides demonstrating formal lambda calculus with text-to-speech and highlighting of some parts. WebBuilder was very popular at Open doors and many other demos.

Leon (2016) http://leon.epfl.ch
I added programming-by-example features to synthesize and repair pretty-printers using a custom optimized string solver. I rewrote the front-end in ScalaJs that helped further development. The demo about converting matrices to string, html and LaTeX had a great impact on IC Research Days 2016.

Microsoft Prose Playground (2015) https://prose-playground.cloudapp.net
I developed English-like structured translation of programs, disambiguation at program fragment level, and the global disambiguation engine. I wrote a compositional engine to chain many extraction processes, and adapted the engine for extracting from web pages and from 2D tables. I also developed other less academic but useful parts such as undo/redo, typed AJAX calls, preview of the structure, algorithms to infer the structure of extraction, preview in the scroll bar, highlighting on mouse hover of parts of programs, the output result table, export features, and documentation.
We got the first prize at Microsoft TechFest for the technology and interface. Bill Gates himself said that we need more technologies like this.

StringSolver (2014) https://github.com/MikaelMayer/StringSolver
Open-source JVM implementation of the algorithms beyond Flash Fill from Microsoft, with a plug-in mechanism, and support for date-number conversion, file counters and ellipsis (partial output). Before StringSolver, people would use FlashFill only on Excel, but with no access a source code implementing it. StringSolver also later helped to show the effectiveness of Microsoft Prose, as it took me 5 months to develop what in Microsoft Prose we would do in 2.

Pong Designer (2014) https://play.google.com/store/apps/details?id=ch.epfl.lara.synthesis.kingpong
A game engine where we program games while playing them. I programmed almost: Interface and internals, object hierarchy, reversible time engine, pre-built shapes, drawing, sound recording, camera filming, code display, fix engine, new rule engine. I propagated some ideas of this work in Prose (Program view, Disambiguation), which had a great success.

Complete Functional Synthesis (2010) https://github.com/epfl-lara/comfusy
I designed and implemented the algorithms in Scala for synthesizing a program from linear arithmetic specifications. I wrote an extension to some particular cases of non-linearism. This is my most quoted work until now, around 100 citations. It was the first time that someone exhibited complete algorithms for synthesis, with some real use-cases and very accessible syntax.