Program Synthesis for Empowering End Users and Stress-Testing Compilers

By

VU MINH LE
B.Eng. (University of Technology, Vietnam National University) 2006
M.Sc. (University of California, Davis) 2011

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Sumit Gulwani

Premkumar Devanbu

Committee in Charge

2015

Vu Minh Le
September 2015
Computer Science

Program Synthesis for Empowering End Users
and Stress-Testing Compilers

Abstract

Since 2012, the number of people with access to computing devices has exploded. This trend is due to these devices' falling prices, and their increased functionalities and programmability. Two challenges arise from this trend: (1) How to assist the increasing number of device owners, most of whom are non-programmers, yet hope to take full advantage of their devices, and (2) how to make critical software, which is relied upon by other software running on these devices, more reliable? Our vision is that *program synthesis* is the solution for the above arising challenges. This dissertation describes various program synthesis techniques to synthesize programs from natural languages, input/output examples, and existing programs to tackle these challenges.

We present SmartSynth, a natural language interface that synthesizes smartphone programs from natural language descriptions. SmartSynth enables users to automate their repetitive tasks by leveraging various phone sensors. It is the first system that combines the advances in both natural language processing (NLP) and program synthesis: it uses NLP techniques to parse a given command, and applies program synthesis techniques to resolve parsing ambiguities. We have adapted and extended SmartSynth's algorithms to integrate it into TouchDevelop, a popular touch-based programming environments for end users.

We develop FlashExtract, a system that extracts data from semi-structured document (such as text files, webpages, and spreadsheets) using examples. Natural language does not work well in this domain because the tasks are complicated and users usually do not know how to perform them. In FlashExtract, users only need to highlight some sample regions to be extracted, the system will learn a program to select similar regions

automatically. They can also provide nested examples to extract structured, hierarchical data. FlashExtract has been shipped as the cmdlet ConvertFrom-String of PowerShell in Microsoft Windows 10 and as the Custom Fields feature in Microsoft Azure Operational Insights.

While it is important to make programming accessible to end users, it is also vital to improve the correctness of critical software because they impact other software products. We introduce Equivalence Modulo Inputs (EMI), a novel, general methodology to synthesize valid compiler test programs from existing test cases. Given a test program and a set of its inputs, we profile the program's execution over the inputs. We then generate new test variants by randomly removing code that is unexecuted under the provided inputs. The expectation is that these new variants should behave exactly the same as the original program under the same inputs; any observed discrepancy indicates a compiler bug. Our technique is simple to realize yet very effective. In total, we have reported more than 400 bugs in GCC and LLVM, most of which have already been fixed. Many compiler vendors have adopted EMI to test their compilers.

We also believe that cross-disciplinary solutions can increase the impact of program synthesis techniques. We therefore further explore program synthesis in the following three dimensions. First, we introduce two novel user interaction models to help users resolve ambiguities in programming-by-example systems like FlashExtract. One model allows users to effectively navigate among the large set of programs consistent with the provided examples. The other model uses active learning to ask questions to help differentiate the outputs of these programs. Second, we present a guided, advanced mutation strategy based on Bayesian optimization to synthesize EMI variants more effectively. Our improved technique supports both code deletions and insertions in the unexecuted regions, and uses Markov Chain Monte Carlo (MCMC) optimization to guide the synthesis process. Finally, we apply program synthesis to find bugs in a new domain: the link-time optimizer components in compilers.

To Minhon and Koala.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am very fortunate to have Zhendong and Sumit as my PhD advisors. They gave me many valuable lessons to make me what I am today. Each year working with them is a different adventure. I could not possibly be happier. To me, they are the best advisors in the world.

I am grateful to my friends at UC Davis, especially Mehrdad Afshari, Earl Barr, Sven Koehler, Chengnian Sun, Martin Velez, Thanh Vo, and Daniel Zinn, for all the discussions we have about virtually everything. I have learned a lot from you.

My special thank to my second family in Davis: Vu Nguyen, Ngoc Nham, Linh Huynh, Binh Pham, Trang Le, Phuong Nguyen, Thu Ha, Tuan Anh Pham, Linh Trinh, and many others. Six wonderful years in Davis are full with memories with you. I will miss you all.

Finally, and most importantly, I would like to thank my wife, Quyen Nguyen, who has been always with me through ups and downs of life. I also like to thank my parents, Thang Le and Luong Nguyen, and Quyen's parents, Long Luu and Hien Le, for their unending encouragement and support. I love you all.

Chapter 1

Introduction

We have entered a new era for computing in recent years with the sharp increase in availability and popularity of multi-functional mobile devices. These devices are equipped with more and more sensors, which radically multiplies the ways people interact with them. Unfortunately, most people who have or will have access to these devices are non-programmers who are unable to make full effective use of their devices. The application market is unlikely to meet the personalized needs of device users. The economic model in the application market favors the presence of only those applications that would be used "as-is" by a large number of people. This has created a significant opportunity to cater to the needs of the long tail of people looking for *personalized* applications that only they, or very few other people, would want to use.

At the same time, the massive amount of data generated by users, applications, and devices have resulted in the big data revolution. However, such data are usually embedded in documents of various types such as text/log files, spreadsheets, and webpages. Although these documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data model, they are not suitable for data manipulation tasks. As a result, one challenge in the data revolution is to bring unstructured data embedded in such documents into structured format, from which big data algorithms can be applied. The task, often referred as data cleaning or data wrangling, is tedious and time consuming. Studies show that data scientists spend 50-80% of their time for this

task [Times, 2014]. There are some early efforts in solving this bottleneck of the big-data workflow [Kandel et al., 2011, Metaweb Technologies, Inc., 2015], but they only focus on CSV (Comma Separated Values) files. These techniques also require users to have some knowledge in programming. This is not the case for business users, who have to perform data cleaning tasks occasionally. We need a more general and user-friendly solution to extract data from various sources and to help users without any programming background complete their tasks.

The abundant of devices and applications have put more pressure to critical software, such as compilers, on which all these applications depend. Despite of decades of extensive research and development, production compilers still contain bugs, and in fact quite a few. What makes compiler bugs distinct from other application bugs is that compiler bugs often manifest only indirectly as application failures, which makes them harder to recognize. When compiler bugs occur, they frustrate programmers and may lead to unintended application behavior and disasters, especially in safety-critical domains. It is therefore critical to make compilers more robust.

Our vision is that ***program synthesis*** is the solution for the above arising challenges. We have developed various program synthesis techniques to synthesize programs from natural languages, input/output examples, and existing programs to tackle these challenges. In particular, we develop SmartSynth, a system that synthesizes smartphone scripts from natural language descriptions. SmartSynth lets users exploit various phone sensors and functionalities to automate their repetitive tasks. We have integrated SmartSynth into TouchDevelop [Tillmann et al., 2011]. Our solution to the data cleansing problem is FlashExtract, a system that synthesizes data extraction programs from input/output examples to extract data from semi-structure documents (*e.g.*, text files, webpages, spreadsheets). FlashExtract has been shipped as part of PowerShell in Microsoft Windows 10 and as the Custom Fields feature in Microsoft Azure Operational Insights. Finally, we introduce Equivalence Modulo Inputs (EMI), a novel and general methodology to synthesize valid compiler test programs from existing programs, as part of our efforts to improve the quality of compilers. EMI has helped improved the quality

of compilers significantly. In total, we have reported more than 400 confirmed bugs in GCC and LLVM, most of which have already been fixed. Many compiler vendors have adopted EMI to test their compilers.

SmartSynth: Synthesizing Smartphone Programs from Natural Language In the first part of this dissertation, we study the problem of developing smartphone automation scripts, those that execute some tasks automatically under certain conditions (*e.g.*, the phone turns off network connections automatically when its battery is low, the phone sends the user's spouse a text saying "Kids are dropped" when he is at the location of their kids' school). The abundance of such scripts on popular smartphone platforms like Locale [Two forty four a.m. LLC., 2013], Tasker [Crafty Apps, 2015], and On{x} [Microsoft Research, 2013] motivate our problem domain.

The current state of the art in end-user programming of smartphones is far from satisfactory. General-purpose languages such as Java, C#, and Objective C are clearly not suitable for most end user. Visual programming systems such as App Inventor [MIT Center for Mobile Learning, 2015] and Tasker [Crafty Apps, 2015], which visualize programming constructs into building blocks, still require end users to consciously think about programming like typical programmers. Digital assistants like VoiceActions [Google, 2013] and Siri [Apple, 2013] only support limited smartphone features.

Our goal is to be (1) *natural* so that users can interact with the system through natural language (NL) and (2) *general* so that, unlike Siri, our system allow arbitrary combination of smartphone features. To this end, we introduce SmartSynth, a new end-to-end *NL-based programming system* for synthesizing smartphone scripts. The key *conceptual novelty* of SmartSynth is the methodology to decompose the problem into: (1) designing a domain-specific language (DSL) to capture automation scripts users commonly need, and (2) combining natural language processing (NLP) and program synthesis [Gulwani, 2010] techniques to synthesize scripts in the DSL from NL. Our *technical novelties* include: (1) a carefully designed DSL that incorporates standard constructs from smartphone programming platforms and balances its expressivity and the feasibility for automatic script synthesis from NL descriptions, (2)

techniques adapted from the NLP community to translate English descriptions into relevant components (*i.e.* script constructs) and a partial set of dataflow relations among the components, and (3) techniques inspired by the program synthesis community to complete the partial set of dataflow relations via type-based program synthesis [Gulwani et al., 2011, Mandelin et al., 2005, Perelman et al., 2012] and construct the desired scripts via reverse parsing [Aho et al., 2006].

FlashExtract: Synthesizing Data-Extraction Program from Examples Existing programmatic solutions to data extraction have three key limitations. First, the solutions are domain-specific and require knowledge/expertise in different technologies for different document types. Second, they require understanding of the entire underlying document structure including the data fields that the end user is not interested in extracting and their organization (some of which may not even be visible in the presentation layer as in case of webpages). Third, and most significantly, they require knowledge of programming. The first two aspects create challenges for even programmers, while the third aspect puts these solutions out of reach of the vast majority of end users who lack programming skills. As a result, users are either unable to leverage access to rich data or have to resort to manual copy-paste, which is both time-consuming and error prone.

In the second part of this dissertation, we address the problem of developing a uniform end-user friendly interface to support data extraction from semi-structured documents of various types. Our methodology includes two key novel aspects: a uniform user interaction model across different document types, and a generic inductive program synthesis framework.

Uniform and End-user Friendly Interaction Model Our extraction interface supports data extraction via examples. The user initiates the process by providing a nested hierarchical definition of the data that he/she wants to extract using standard structure and sequence constructs. The user then provides examples of the various data fields and their relationships with each other. An interesting aspect is that this model is independent of the underlying document type. This is based on our observation that different document types share one thing in common: a two-dimensional presentation

layer. We allow users to provide examples by highlighting two-dimensional regions on these documents. These regions indicate either the fields that the user wants to extract or structure/record boundaries around related fields.

Inductive Program Synthesis Framework To enable data extraction from examples, we leverage inductive program synthesizers that can synthesize scripts from examples in an underlying domain-specific language (DSL). The key technical contribution is an inductive program synthesis framework that allows easy development of inductive synthesizers from mere definition of DSLs (for data extraction from various document types). We describe an expressive algebra containing four core operators: map, filter, merge, and pair. For each of these operators, we define its sound and complete generic inductive synthesis algorithms parameterized by the operator's arguments. As a result, the synthesis designer simply needs to define a DSL with two features: (a) It should be expressive enough to provide appropriate abstractions for data extraction for the underlying document type, (b) It should be built out of the operators provided by our core algebra. The synthesis algorithm is provided for free by our framework. This is a significant advance in the area of programming by examples, wherein current literature [Gulwani et al., 2012, Gulwani, 2012] is limited to domain-specific synthesizers.

EMI: Synthesizing Compiler Test Programs from Existing Programs In the third part of this dissertation, we introduce a simple, yet broadly applicable concept for validating compilers. Our vision is to take existing real-world code and transform it in a novel, systematic way to produce different, but equivalent variants of the original code.

Equivalence Modulo Inputs We introduce *equivalence modulo inputs* (EMI) for a practical, concrete realization of the vision. EMI crucially leverages the connection between dynamic program execution and static compilation to synthesize program variants. These EMI variants can specifically target compiler optimization phases, and stress-test them to reveal latent compiler bugs.

The key insight behind EMI is to exploit the interplay between *dynamically executing* a program P on a subset of inputs and *statically compiling* P to work on all inputs. More concretely, given a program P and a set of input values I from its domain, the input set I

induces a natural collection of programs $\mathcal{C}$ such that every program $Q \in \mathcal{C}$ is equivalent to P modulo I: $\forall i \in I, Q(i) = P(i)$. The collection $\mathcal{C}$ can then be used to perform differential testing [McKeeman, 1998] of any compiler $Comp$: If $Comp(P)(i) \neq Comp(Q)(i)$ for some $i \in I$ and $Q \in \mathcal{C}$, $Comp$ has a miscompilation.

EMI is effective because although an EMI variant Q is only equivalent to P modulo the input set I, the compiler has to perform all its (static) analysis and optimizations to produce correct code for Q over *all inputs*. Moreover, P's EMI variants, while semantically equivalent *w.r.t.* I, can have quite different static data- and control-flow. Since data- and control-flow information critically affects which optimizations are enabled and how they are applied, the EMI variants not only help exercise the optimizer differently, but also demand the exact same output on I from the generated code by these different optimization strategies.

Orion Given a program P and an input set I, the space of P's EMI variants *w.r.t.* I is vast, and difficult or impossible to compute. Thus, for realistic use, we need a practical instantiation of EMI. We propose a "profile and mutate" strategy to systematically synthesize a subset of a program's EMI variants. In particular, given a program P and input set I, we profile executions of program P over the input set I, and derive (a subset of) P's EMI variants (*w.r.t.* I) by stochastically pruning, inserting, or modifying P's unexecuted code on I. These variants should clearly behave exactly the same on the same input set I as the original program P. We then feed these variants to any given compiler. Any detected deviant behavior on I indicates a bug in the compiler.

We have implemented our "profile and mutate" strategy for C compilers and focused on *pruning unexecuted code*. We have extensively evaluated our tool, Orion[1], in testing three widely-used C compilers—namely GCC, LLVM, and ICC—with extremely positive results. We have used Orion to generate variants for real-world projects, existing compiler test suites, and much more extensively for test cases generated by Csmith [Yang et al., 2011]. In eleven months, we have reported, for GCC and LLVM alone, 147 confirmed, *unique* bugs.

[1] Orion was a giant huntsman in Greek mythology.

Other Dimensions of Program Synthesis The last part of this dissertation is dedicated to discussing other dimensions of program synthesis to demonstrate the *breath* and *depth* of this technique.

New User Interaction Models: Program Navigation and Conversational We investigate two user interaction models to improve the *usability* of systems that synthesize programs from examples such as FlashExtract. The first model, called Program Navigation, allows users to navigate among all programs synthesized by the underlying synthesis engine (as opposed to displaying only the top-ranked program) and to pick one that is intended. We also paraphrase these programs in English for easy readability. The second model, called Conversational Clarification, is based on active learning. It asks questions to users to resolve ambiguities in the their specification with respect to the available test data. We generate these questions from the discrepancies in outputs of our synthesized programs (note that these programs agree on the output of the examples but may disagree on the rest of the file). We have implemented the above two user interaction models in a generic user interface framework called FlashProg. The FlashProg framework support several synthesis engines related to data manipulation, such as FlashFill [Gulwani, 2011], FlashRelate [Barowy et al., 2015], FlashExtract [Le and Gulwani, 2014].

New Technique: Guided Stochastic Program Synthesis We propose novel, effective techniques to address limitations in Orion, our first realization of EMI. Besides deletion (the only mutation supported in Orion), we support *code insertion* into unexecuted program regions. Because we can potentially insert an unlimited number of statements into these regions, we can generate an enormous number of variants. More importantly, the synthesized variants have substantial different control- and data-flow, therefore helping exercise the compiler much more thoroughly. Our experimental results show that the increased variation and complexity are crucial in revealing more compiler bugs.

Furthermore, we introduce a novel method to guide the synthesis process to uncover deep bugs. We formulate our bug finding process as an optimization problem whose goal is to *maximize* the difference between a variant and the seed program. By synthesizing substantially diverse variants, we aim at testing more optimization strategies that may

not be exercised otherwise. We realize this process using Markov Chain Monte Carlo (MCMC) techniques, which help effectively sample the program space to allow diverse programs. Our evaluation results show that our realization Athena is very effective in finding deep bugs that require long sequences of sophisticated mutations on the seed program. Our results also demonstrate that most of these bugs could not be discovered by Orion, which only uses a much simpler, blind mutation strategy.

New Domain: Stress-Testing Link-Time Optimizers Finally, we illustrate the general *applicability* of program synthesis by applying it to finding bugs in a new domain: link-time optimizers. We propose Proteus[2], a randomized differential testing technique to stress-test link-time optimizers, and the first extensive effort to stress-test LTO in GCC and LLVM. Two key challenges are to find LTO-relevant test programs (which typically involve multiple compilation units), and to reduce the bug-triggering test programs effectively.

To tackle the first challenge, we use Csmith to generate *single-file* test programs. We then automatically transform each test program in two semantics-preserving manners. First, we extend Orion to *inject arbitrary function calls to unexecuted code regions* to increase function-level interprocedural dependencies. The increased dependencies stress-test LTO more thoroughly. Second, we split each test program into separate compilation units. We compile each of these compilation units at a random optimization level, and finally link the object files also at a random optimization level. As for the second challenge, we develop an effective procedure to *reduce multiple-file test programs* that trigger bugs. Our key observation is that the bug-triggering property of our splitting function is preserved under reduction, which allows us to perform reduction on the original single-file test program. Indeed, after reduction, we split a reduced test program into separate compilation units, and check for bug-triggering behavior. This approach works very well in practice. Most of our tests were reduced within several hours. In comparison, existing reduction techniques take days or weeks, or never terminate, and produce invalid reduced tests.

[2]Proteus is a Greek sea god who can *assume different forms*.

Contributions We make the following contributions:

- We introduce SmartSynth, an end-to-end system that synthesizes smartphone automation scripts from natural language descriptions. The underlying algorithm of SmartSynth combines advances in both natural language processing and program synthesis. Our evaluation on 50 different tasks collected from smartphone help forums shows that SmartSynth is effective. It can synthesize the intended scripts in real time for over 90% of the 640 descriptions collected via a user study.

- We present FlashExtract, a system that lets users extract data from semi-structure documents using examples. FlashExtract eliminates the need to learn domain-specific scripting technologies and understand the document's internal data model. We also introduce a rich algebra of operators for data extraction DSLs and a modular inductive synthesis strategy for each of these operators. This allows rapid development of data-extraction synthesizers. In our evaluation that involves extracting data from 75 real-world documents, FlashExtract only needs 2.36 examples to extract a field, and it completes within 0.84 seconds on average.

- We present a generic framework called FlashProg that implements two novel user interaction models: (1) Program Navigation that lets users browse the large space of satisfying programs by selecting ranked alternatives for different program subexpressions, and (2) Conversational Clarification that perform conversations with users to resolve ambiguity. Our user study shows that both models significantly reduce the number of errors without any difference in completion time.

- We introduce the novel, general concept of equivalence modulo inputs (EMI) for systematic compiler validation. We propose the "profile and mutate" strategy to realize Orion, a practical implementation of EMI for testing C compilers. We report our extensive evaluation of Orion in finding numerous bugs (147 unique bugs) in GCC and LLVM.

- We implement a new EMI mutation strategy that allows inserting code into unexecuted regions. This helps synthesize more diverse variants that have substantially

different control- and data-flow. We propose a novel guided bug-finding process that uses MCMC sampling to find more diverse test variants to trigger deep bugs that otherwise could not be triggered using existing techniques. Our tool Athena has found 72 new bugs in GCC and LLVM in 19 months (68 have been fixed).

- We introduce Proteus, the first randomized differential testing technique to stress-test link-time optimizers. We propose a practical procedure to reduce LTO bugs that is efficient (*i.e.* significantly shortening reduction time) and effective (*i.e.* reliably rejecting invalid tests). We demonstrate that Proteus is effective. In 11 months of continuous testing, we have reported 37 bugs, among which 21 have been confirmed, and 11 have been fixed.

We structure the remainder of this dissertation as follows. Chapter 2 discusses program synthesis using natural language and its instantiation SmartSynth. Chapter 3 introduces program synthesis using input/output examples and its instantiation FlashExtract. Chapter 4 presents program synthesis using existing programs and its instantiation EMI. Chapter 5 discusses various extensions of our program synthesis techniques. We survey related work in Chapter 6 and conclude in Chapter 7.

Chapter 2

SmartSynth: Synthesizing Smartphone Programs from Natural Language

In this chapter, we study the problem of developing smartphone automation scripts, those that execute some tasks automatically under certain conditions (*e.g.*, the phone turns off network connections automatically when its battery is low, the phone sends the user's spouse a text saying "Kids are dropped" when he is at the location of their kids' school). The abundance of such scripts on popular smartphone platforms like Tasker [Crafty Apps, 2015] and Locale [Two forty four a.m. LLC., 2013] motivate our problem domain.

The current state of the art in end-user programming of smartphones is far from satisfactory. General-purpose languages such as Java, C#, and Objective C are clearly not suitable for most end user. The other alternatives are visual programming systems such as App Inventor [MIT Center for Mobile Learning, 2015] and Tasker [Crafty Apps, 2015], which visualize programming constructs into building blocks. In these systems, users program by identifying blocks and composing them visually. Although these systems are more user-friendly, they still require end users to consciously think about programming like typical programmers, such as introducing variables and deciding when/how to use conditionals, loops, or events.

Because of the deluge of mobile devices and their underlying systems, we believe in the promising direction toward *general* programming systems with *natural* interfaces for

meeting end users' needs. Two notable examples are VoiceActions [Google, 2013] and Siri [Apple, 2013] for controlling smartphones via speech. However, they only provide support for limited smartphone features. Siri, for example, only matches speech to pre-defined patterns. Our goal is to be (1) *natural* so that users can interact with the system through natural language (NL) and (2) *general* so that, unlike Siri, our system does not simply match NL descriptions to pre-defined patterns.

To this end, we introduce SmartSynth, a new end-to-end *NL-based programming system* for synthesizing smartphone scripts. The key ***conceptual novelty*** of SmartSynth is the methodology to decompose the problem into: (1) designing a domain-specific language (DSL) to capture automation scripts users commonly need, and (2) combining natural language processing (NLP) and program synthesis [Gulwani, 2010] techniques to synthesize scripts in the DSL from NL. Our ***technical novelties*** include: (1) a carefully designed DSL that incorporates standard constructs from smartphone programming platforms and balances its expressivity and the feasibility for automatic script synthesis from NL descriptions, (2) techniques adapted from the NLP community to translate English descriptions into relevant components (*i.e.* script constructs) and a partial set of dataflow relations among the components, and (3) techniques inspired by the program synthesis community to complete the partial set of dataflow relations via type-based program synthesis [Gulwani et al., 2011, Mandelin et al., 2005, Perelman et al., 2012] and construct the desired scripts via reverse parsing [Aho et al., 2006].

Indeed, we combine and refine recent advances in both the NLP and program synthesis areas. A recent, emerging trend in NLP is semantic understanding of natural languages. There have been recent, although limited, successes in translating structured English into database queries [Androutsopoulos, 1995] and translating NL statements into logic in specific domains [Kate et al., 2005, Finucane et al., 2010]. As for program synthesis, the traditional goal has been to discover new or complicated algorithms from *complete logical specifications*. There is recent, renewed interest in this area of synthesis because of (1) interesting applications (such as end-user programming [Gulwani et al., 2012] and intelligent tutoring systems [Gulwani, 2012]), and (2) the ability to deal with

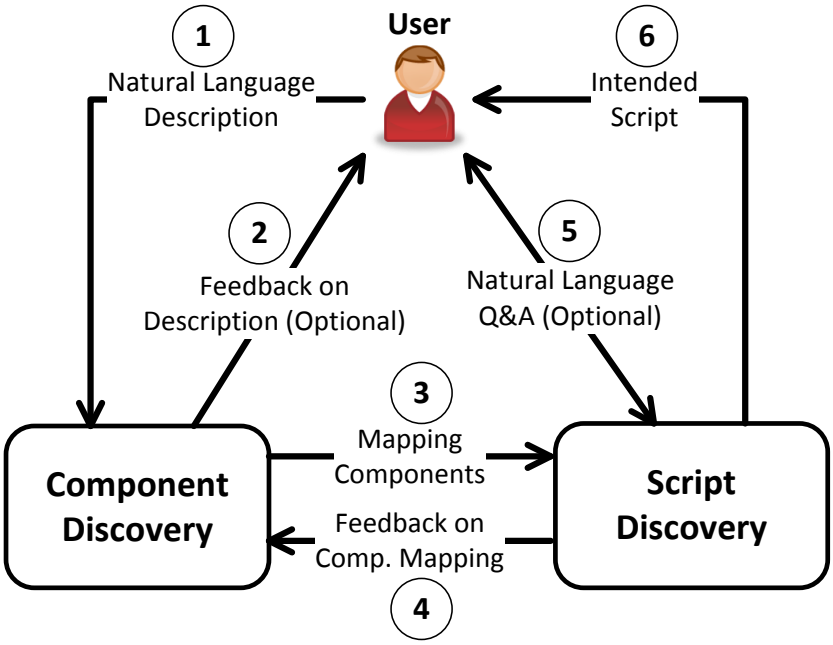


Figure 2.1: SmartSynth's system architecture.

under-specifications (such as examples [Gulwani et al., 2012], and a set of APIs or keywords [Gulwani et al., 2011, Little and Miller, 2007]). Our combination of NLP and program synthesis is synergistic since NLP is used to "partially understand" natural language and program synthesis is then used to refine this understanding and generate the intended script.

Figure 2.1 illustrates SmartSynth's process of synthesizing scripts. The user communicates her intent via natural language (Step 1). The "Component Discovery" box contains two algorithms that are inspired by the NLP area: (i) a mapping algorithm that maps the description into script components such as APIs and literals (Step 3), and (ii) an algorithm that asks the user to refine parts of the description that SmartSynth does not understand (Step 2). The "Script Discovery" box first uses an NLP-based algorithm to detect dataflow relations among the identified components. If these relations do not fully specify the dataflow relationships between components, SmartSynth invokes our algorithm (inspired by type-based synthesis) to complete the missing dataflow relations. If there are multiple equally high-ranked relations, SmartSynth initiates an interactive conversation with the user to resolve the ambiguities (Step 5). As the final step, it

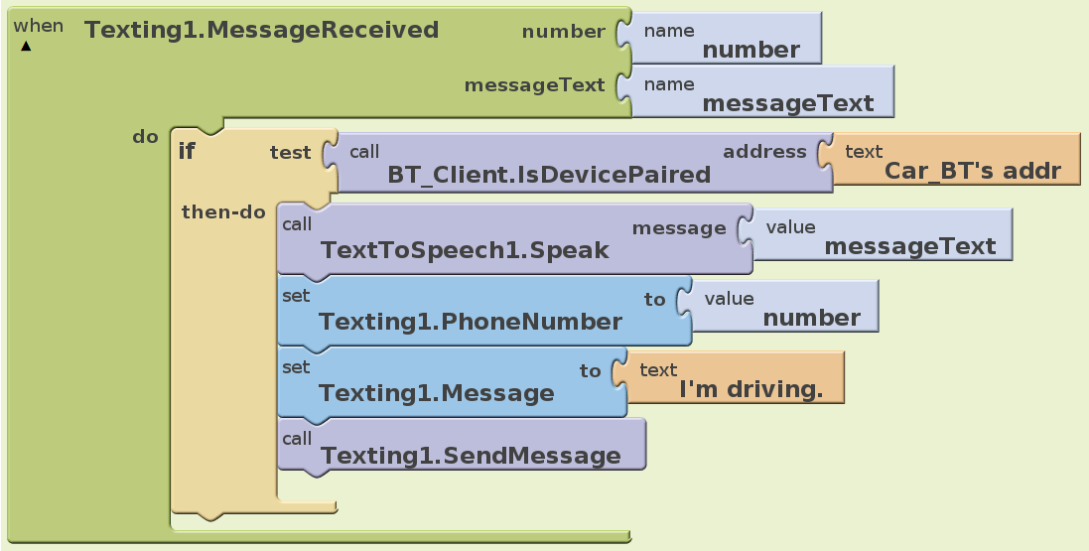


Figure 2.2: A visual program for Example 1 written in AppInventor.

constructs the intended script from the identified components and relations using an algorithm akin to reverse parsing.

An interesting feature of SmartSynth is the NLP-Synthesis feedback loop in Steps 3-4. Although our NLP algorithm has its own mapping feature set to perform component mapping, it might not precisely capture the mapping in some cases, due to irregularities of natural language. SmartSynth uses program synthesis technique to gain confidence about the quality of the generated script and as another metric for the quality of NLP mapping. In particular, SmartSynth repeatedly requests the NLP-based algorithm for the next likely interpretation of the description, if the current interpretation is deemed unlikely (*i.e.*, the interpretation does not translate to a high-confidence script).

2.1 Motivating Example

We motivate our system via the following running example, taken from a help forum for Tasker [Crafty Apps, 2015].

Example 1 *The user wants to create a script to do the following when she receives an SMS while driving: (1) read the content of the message and (2) send a message "I'm driving" to the sender. Since the user always connects her car's bluetooth to the phone when she is in the*

car, she uses this fact to denote that she is driving[1]*. One possible description of this script is:*

> *When I receive a new SMS, if the phone is connected to my car's bluetooth, it reads out loud the message content and replies the sender* "I'm driving.".

Users of both conventional and visual programming systems have to deal with all the low-level details and make several decisions, often unintuitive to them, during the process of creating a new program. In contrast, users of SmartSynth only need to give the system their problem description in NL, and interact with it, if necessary, in natural language. Below we compare traditional programming systems with SmartSynth using our example.

Identifying Components Users of conventional and visual programming systems have to conceptualize their ideas into script components (*i.e.*, script constructs such as APIs), potentially under several refinement steps [Wirth, 1971]. This is a non-trivial task as it assumes that users should understand all script constructs' specifications. As shown in Figure 2.2, users of AppInventor have to conceptualize and identify various components while transforming the idea in Example 1 into a running script.

In comparison, SmartSynth automatically decomposes the description into disjoint text chunks and matches each chunk to its supported components (details in Section 2.3.1). For the description in Example 1, SmartSynth is able to decompose and match the description to the components shown in Table 2.1. When any chunk of the description cannot be mapped to a component, SmartSynth will ask the user to refine it.

However, as is often the case in NLP, mapping a chunk to components can be ambiguous. For example, we can map *"if the phone is connected to"* to any of the three candidates shown in Table 2.1. Similarly, we can map *"replies"* to either `SendMessage` or `SendEmail`. One can look at API arguments to resolve these ambiguities. In the first case, since the argument is a bluetooth device, it is very likely that the mapping component is `IsConnectedToBTDevice`. But this approach does not work as well for the second case, where the ambiguity can only be resolved by considering the global context that may indicate receipt or sending of an SMS.

[1] *This is a clever workaround to avoid using the GPS sensor, which drains the battery power very quickly.*

Description	Possible Mappings
When I receive a new SMS	`MessageReceived`

if the phone is connected to	`IsConnectedToBTDevice`
	`IsConnectedToWifiNetwork`
	`IsConnectedToDataService`

my car's bluetooth	`Car_BT`
reads out loud	`Speak`
the message content	`MessageReceived.Text`$_O$
replies	`SendMessage` `SendEmail`
the sender	`MessageReceived.Number`$_O$
"I'm driving"	`"I'm driving"`

Table 2.1: Possible mappings from text chunks to components in Ex. 1. `Text`$_O$ and `Number`$_O$ are return values of `MessageReceived`.

SmartSynth takes a different approach. Instead of manually encoding many disambiguation rules, it relies on the techniques inspired by type-based synthesis to automatically disambiguate mapping candidates. Specifically, from each mapping candidate that needs to be resolved, SmartSynth generates a script and assigns it a score indicating how likely the script is. The best mapping is associated with the script that has the highest score. In Example 1, SmartSynth is able to select the right mapping set (`IsConnectedToBTDevice`, `SendMessage` and the others) because together they form the highest ranked script.

Synthesizing Scripts Besides identifying all necessary components, users of conventional systems also need to understand the components' low-level details in order to assemble them meaningfully.

To create the script in Figure 2.2, the users must understand that `MessageReceived` is an event and returns a phone number and a message content, which may be stored in two temporary variables. They need to understand that `IsDevicePaired` is the guard

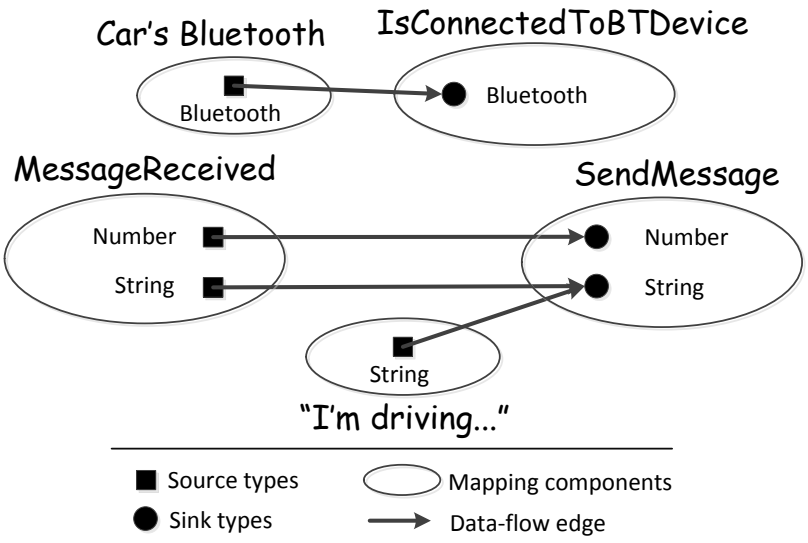


Figure 2.3: The graph showing all possible dataflow relations among components identified by the Component Discovery algorithms.

of the conditional and it must be linked with the car's bluetooth. Also, the APIs `Speak` and `SendMessage` must be configured with arguments of types `Text` and `Number` & `Text` respectively. Finally, the users need to pass the temporary variables/literals to those APIs and arrange them in the correct order.

In contrast, users of SmartSynth are not required to understand those low-level details because the system knows the signatures of all those APIs. The challenge is to generate additional script constructs such as loops, conditionals and assignments to combine these components together into a script reflecting the user's intent. SmartSynth solves this challenge in the following steps (details in Section 2.3.2). First, it builds a special data structure that represents all possible dataflow relations among the components. We call each of them a (dataflow) *relation*. Figure 2.3 shows the data structure for the right component mapping set. An edge in this figure represents a relation, which specifies a possible dataflow from a value (source) to an API's parameter (sink). The value from a source might be assigned to multiple sinks (if they have the same type), and a sink might receive a value from different sources (also of the same type). For example, the sink denoting the message content in `SendMessage` can be assigned to the argument

Return value or Literal	API Parameter
`Car_BT`	`IsConnectedToBT-Device.Bluetooth`$_I$
`MessageReceived.Text`$_O$	`ReadText.Text`$_I$
`MessageReceived.Number`$_O$	`SendMessage.Number`$_I$
`"I'm driving"`	`SendMessage.Text`$_I$

Table 2.2: Relations detected from Example 1's description. Subscripts I/O characterize the field of an API as a parameter or a return value.

from either the received message of `MessageReceived`, or the string literal "I'm driving" because they have the same type `String`.

Next, SmartSynth uses classic NLP techniques [Jurafsky and Martin, 2008] to detect likely relations from the NL description. These relations must be derived from the set of all possible relations embedded in the graph. Table 2.2 shows the relations that SmartSynth has detected from the description. A row in this table represents a relation defining which source is assigned to a sink. For example, the last row states that the message content to be sent by `SendMessage` is the string literal "I'm driving". SmartSynth generates and returns the final script to the user if it is able to detect all necessary relations from the NL description (Figure 2.4).

```
when (number, content) := MessageReceived()
  if (IsConnectedToBTDevice(Car_BT) then
    Speak(content);
    SendMessage(number, "I'm driving");
```

Figure 2.4: The script for Example 1.

However, since users can give free-form descriptions, SmartSynth may often detect only a *subset* of the necessary relations. In these cases where the intent is under-specified, SmartSynth uses techniques inspired by type-based synthesis to find the missing relations. It performs searching over the dataflow graph for missing relations and uses a special

ranking scheme to prioritize more likely relations. SmartSynth then uses these newly discovered relations to generate the most likely script and returns it to the user.

As an example, suppose that the user had given a slightly different description *"... and send back..."* (instead of *"... and replies the sender..."*) and also suppose that SmartSynth does not model this mapping and thus is unable to extract the third relation between $\texttt{MessageReceived.Number}_O$ and $\texttt{SendMessage.Number}_I$ in Table 2.2. Nonetheless, SmartSynth can discover this missing relation and generate the same script as in Figure 2.4.

2.2 Automation Script Language

In this chapter, we use a scripting language SmartScript (Figure 2.5) with representative features of the current smartphone generation. Since SmartScript is an intermediate language, we can port it to other mobile platforms via simple syntax-directed translation [Aho et al., 2006].

2.2.1 Language Features

We have designed SmartScript from an extensive study of the scripts from various smartphone help forums. This design process is an interesting exercise to balance the trade-offs between the expressiveness of SmartScript and the effectiveness of SmartSynth. The restrictions that we place in SmartScript (event and conditionals only appear at the top of the script) allow SmartSynth to perform type-based synthesis more effectively under the uncertainties in NL processing.

A script P in SmartScript represents a task that executes a sequence of actions under a certain condition. It has some parameters I (which will be entered by users when the script runs) and may be triggered by an event E. When the event occurs, it generates some variables, which might be converted by T into new types that are used in the condition C. The condition is then evaluated and if it holds, the body M is executed.

We use a few examples, each of which contains an NL description and the synthesized script, to illustrate our key language constructs.

Event and Conditional

Example 2 [Phone Locator] *When the phone receive a new text message, reply with my current location if the message content is "Secret code".*

```
when (number, text) := MessageReceived
  if (text = "Secret code") then
    text2 := LocationToString(CurrentLocation);
    SendMessage(number, text2);
```

API Composition

Example 3 [Picture Uploader] *Take a picture, add to it the current location and upload to Facebook.*

```
pic := TakePhoto();
text := LocationToString(CurrentLocation);
pic2 := AddTextToPhoto(pic, text);
UploadPhotoToFacebook(pic2);
```

Loops

Example 4 [Group Texting] *Send my current location to 111-1111 and 222-2222.*

```
text := LocationToString(CurrentLocation);
foreach number in {111-1111, 222-2222} do
  SendMessage(number, text);
od
```

$$
\begin{array}{rcl}
\text{Script } P & ::= & I\ E\ T\ C\ M \\
\text{Parameter } I & ::= & \mathbf{input}(\mathbf{i_1}, \ldots, \mathbf{i_n}) \ \mid \ \epsilon \\
\text{Event } E & ::= & (\underline{r_1}, \ldots, \underline{r_n}) := \mathbf{when}\ \underline{\texttt{Event}}() \ \mid \ \epsilon \\
\text{Conversions } T & ::= & F_1; \ldots; F_n; \\
\text{Condition } C & ::= & \mathbf{if}\ (\Pi_1 \wedge \cdots \wedge \Pi_n)\ \mathbf{then} \\
\text{Body } M & ::= & Stmt_1; \ldots; Stmt_n; \\
\text{Conversion } F & ::= & \mathbf{x} := \texttt{Convert}(a) \\
\text{Predicate } \Pi & ::= & \underline{\texttt{Predicate}}(a_1, \ldots, a_n) \\
\text{Statement } Stmt & ::= & S \ \mid \ \mathbf{foreach}\ \mathbf{x} \in a\ \ \mathbf{do}\ S_1; \ldots; S_n;\ \mathbf{od} \\
\text{Atomic Statement } S & ::= & A \ \mid \ F \\
\text{Action } A & ::= & (\underline{r_1}, \ldots, \underline{r_n}) := \underline{\texttt{Action}}(a_1, \ldots, a_n) \\
\text{Argument } a & ::= & \mathbf{x} \ \mid \ \mathbf{i} \ \mid \ \underline{r} \ \mid \ \underline{l}
\end{array}
$$

Figure 2.5: The syntax of automation language. : $\mathbf{x}$, $\mathbf{i}$, $\underline{\text{r}}$, and $\underline{\text{l}}$ refer to a temporary variable, an argument of the script, a return value, and a literal, respectively. The essential components identified by the NLP techniques are underlined.

2.2.2 Essential Components

Script components (constructs) in SmartScript are not equally important. A component is essential to the semantics of the script if we cannot reconstruct the script once we have removed the component from it. On the other hand, a component is not essential if we are still able to reconstruct the script if the component is removed. In order to successfully generate the intended script, SmartSynth must be able to identify all essential components from an NL description.

In Figure 2.5, we underline all SmartScript's essential components. An essential component in SmartScript is either an API or an entity. While APIs are pre-defined and related to smartphone functionalities, entities correspond to personal data. This mixture enables the easy creation of personalized automation scripts.

APIs We have created a representative set of 106 APIs that cover most of the available functionalities of the current generation of smartphones. We annotate each API with a set of weighted tags (used by the mapping algorithm to map text chunks to appropriate components), and a type signature (used by the synthesis algorithm to generate only type-safe scripts). The tag set is weighted since some tags are more indicative of a component than others.

We classify the APIs into the following categories to match SmartScript constructs and to assist the NLP engine in mapping descriptions to APIs:

- *Events.* These represent events that occur on the phone. For example, the event of receiving a new text message (`MessageReceived`), the event of receiving a phone call (`IncomingCall`). Events trigger script execution.

- *Predicates*. These denote conditions on the state of the phone. For example, there are text messages that the user has not read (`HasUnreadMessages`), the phone is in landscape orientation (`IsInLandscapeMode`). They also include comparison operators (*e.g.*, $<$, $>$, and $=$). Predicates restrict the condition under which the script is executed.

- *Actions.* These are normal phone operations, such as turning off wifi or bluetooth connection (`TurnWifiOff`, `TurnBluetoothOff`), muting the phone (`MutePhone`), taking a picture (`TakePhoto`), sending a text message (`SendMessage`). SmartSynth executes a sequence of actions in a script's body when its event is triggered and the predicates are satisfied.

Entities Entities represent values that are passed among APIs inside a script. An entity is one of the following:

- *Return values:* These are return values of our representative APIs. For example, the content of the received message ($\texttt{MessageReceived.Text}_O$), the caller of incoming call ($\texttt{IncomingCall.Number}_O$).

- *Literals:* These are personal data items from the user's phone (*e.g.*, a contact in her address book, her bluetooth headset, her music collection, her current location) or generic values (*e.g.*, a string "I'm driving", the time 10pm).

The purpose of identifying APIs is to form the script's skeleton, while that of capturing entities is to prepare for building the data-flow relations among these identified APIs in a later step. We discuss the details of our algorithm next.

2.3 Synthesis Algorithm

The key observation behind our synthesis algorithm is that a SmartScript script can be constructed from its constituent set of essential components and the dataflow relations among those components. This observation allows us to reduce our original problem of synthesizing a SmartScript script from an NL description to the following three sub-problems: (a) identifying essential components (Section 2.3.1), (b) identifying their dataflow relations (Section 2.3.2), and (c) constructing the script from the components and their dataflow relations (Section 21). We use techniques inspired from both the NLP and program synthesis communities to solve these sub-problems.

We first use an algorithm inspired by NLP techniques to map the given NL description to a set of essential components. We then use rule-based relation detection algorithm (a classic NLP technique) to extract dataflow relations among the identified components. Since this algorithm may not be able to extract all relations, we use a technique inspired by type-based synthesis to find the likely missing relations. Finally, we use the components and their relations to construct the script and return it to the user. We explain these algorithms in detail next.

2.3.1 Component Discovery

We assume that for every NL description, there exists a decomposition of that description into disjoint text chunks, where each of the text chunks can be mapped to a component. It is expected that there are many possible ways to decompose a given description into chunks. Furthermore, each of the chunks can be mapped to many possible components. Thus, the space of all possible mappings from an NL description to a set of components

is large. *The key high-level insight* of our component discovery algorithm is to consider all possible mappings (to gain high recall) in a ranked order (to gain high precision).

To this end, we have developed an effective data structure that efficiently stores all possible description decompositions, and the supportive and refutative features used to map these decompositions into components. We then find the best component mapping from this data structure.

Mapping Data Structure Our data structure is a weighted directed graph $\langle V, E \rangle$ where

- V is a set of vertices. Each vertex represents a *cut* between two consecutive words in the description.
- E is a set of edges. An edge (v_i, v_j) denotes a chunk of text that starts from the vertex v_i and ends at the vertex v_j.

For each edge in E, we need to calculate the confidence of mapping its text chunk to a particular component. We use mapping features to calculate this score.

Mapping Features A *feature* h is a function that returns the confidence score of mapping an edge in our data structure to a component. In other words, it indicates how likely a text chunk (represented by the edge) is mapped to a certain component, among all possible components.

Currently, our system uses the following features:

- *Regular expressions.* We use a set of regular expressions to extract entities. A chunk is likely mapped to an entity if it matches the corresponding regular expression. For each entity that is an API's return value, we create a set of common terms that people might call it. For example, we associate `MessageReceived.Text`$_O$ with "message content", "received content", and "received message". To extract literal entities (*e.g.*, the time "10pm", the number "123-4567"), we define regular expressions for all data types used in SmartSynth.
- *Bags-of-words.* We use the bags-of-words model [Jurafsky and Martin, 2008] to categorize a chunk as representing some APIs. A chunk is more likely mapped to

an API if it contains more words that are related to the API's tags. For example, it is more likely to map "send a text" to `SendMessage` than to map the single word "send" to the same API, because the tag set of `SendMessage` contains both "send" and "text".

- *Phrase length.* This feature gives a negative score to a mapping (between a chunk and a component) if the chunk is either too long or too short.

- *Punctuation.* A punctuation is an indication of the start or end of a chunk. Therefore, chunks that start or end with punctuations have higher scores.

- *Parse tree.* Although NLP parsers may not provide precise parse trees, they are still useful because these parse trees are partially correct, and we can exploit them to gain higher mapping confidence. For example, if a parse tree indicates that a text chunk is a noun phrase, it increases the confidence of mapping this chunk to entities. Also, if the main verb in a chunk matches a verb in an action API, it is more likely to map the chunk to this API. In our implementation, we use the Stanford NLP parser [Klein and Manning, 2003].

The above features give different scores for mapping a text chunk to a component. The *aggregate function* combines these scores into a single score. In our implementation, the aggregate function is a weighted sum of all feature scores. It is formally defined as follows:

$$f(e,c) = \sum_{h_i \in F} w_i * h_i(e,c)$$

where e is the edge that contains the text chunk, c is the component that e maps to, F is the feature set, w_i is the feature weight.

For each edge, we select the mapped component whose aggregate score is the highest. This score becomes the edge's weight in the data structure.

$$mapping(e) = \{\langle c, f(e,c)\rangle \mid c \in \underset{c \in C}{\operatorname{argmax}}\ f(e,c)\}$$

where argmax returns the components that maximize the value of the function f.

We store all related information in the data structure to make a *global* decision based on all the features. The next step is to extract the best mappings for the whole NL description from the data structure.

Finding the Best Mapping A path from the start node to the end node decomposes the descriptions into several disjoint text chunks. Each chunk is mapped to a component with a confidence score represented by the weight of its corresponding edge. The total confidence score of mapping the whole description to a component set is the total weight of the corresponding path. Thus, the best mapping (which has the highest total confidence score) corresponds to the longest path in the graph.

2.3.2 Script Discovery

Once SmartSynth has discovered the set of essential components, it uses the following ingredients to synthesize the intended script: (1) type signature of the components, (2) structural constraints imposed by SmartScript, (3) spatial locality of the components in the natural language description, and (4) an effective ranking scheme.

The overall process works as follows. Using typing information, SmartSynth builds a dataflow graph to capture all possible dataflow relations among the provided components. It uses a classic NLP technique to extract dataflow relations from the relative orders of the identified components. If the set of extracted relations is *incomplete*, SmartSynth utilizes the dataflow graph and a ranking scheme to identify the most likely missing relations. SmartSynth then constructs the intended script from the identified components and relations via a process akin to reverse parsing, and returns the script to the user. Next, we discuss these steps in detail.

Dataflow Relations Discovery

Dataflows are essential in determining the intended script. If components are building bricks, dataflows are the cement to glue the bricks together to form scripts. *The key insight* in this process is to use an NLP technique to detect a (partial) set of (high-confidence) dataflow relations followed by a type-based synthesis technique to complete this set using type signatures of components and a ranking scheme.

Definition 1 (Dataflow Relation) *A* dataflow relation *is an ordered pair of a* source *(*i.e.*, an API return value or a literal) and a* sink *(*i.e.*, an API parameter), representing a value assignment from the source to the sink.*

Dataflow Graph We build the graph $G_{\mathcal{C}} = \langle V, E \rangle$ that concisely captures all dataflow relations among the identified components $\mathcal{C}$ as follows:

- V is the set of all sources and sinks in $\mathcal{C}$.
- E is the set of all directed dataflow edges from sources to sinks, such that they are type-compatible. Each edge in E represents a dataflow relation between a source and a sink.

A source type τ_1 is type-compatible with a sink type τ_2 if they have the same type or τ_1 is a list of type τ_2 (we use the later case to generate loops).

The task of SmartSynth is to determine which relations are most relevant in $G_{\mathcal{C}}$. It does this in two steps. First, it detects dataflow relations among components from their relative locations in the description. When necessary, it uses type-based synthesis techniques to infer additional relations from the dataflow graph.

Detecting Partial Dataflow Relations We use rule-based relation detection algorithm, a standard NLP technique, to detect dataflow relations among the identified components. Since all of our APIs are pre-defined and structured (each API is annotated with its type signature), we are able to manually compile a set of rules to extract relations based on the order of how components appear in the sentence. Specifically, if an API component is followed by entities having types that match its signature, those entities are likely to be the API's arguments.

Table 2.3 shows three rules for detecting all relations in Example 1. The first rule states that if the entity following `IsConnectedToBTDevice` has type `BT`, there is a relation between those two.

We remark that these rules can be learned automatically using supervised learning [Jurafsky and Martin, 2008, Kate et al., 2005]. However, in a manageable domain

C_1	**typeof (C_2)**	**typeof (C_3)**	**Relations**
`IsConnectedToBTDevice`	`BT`		$\langle C_2, C_1.\texttt{BT}_I \rangle$
`ReadText`	`Text`		$\langle C_2, C_1.\texttt{Text}_I \rangle$
`SendMessage`	`Number`	`Text`	$\langle C_2, C_1.\texttt{Number}_I \rangle$ $\langle C_3, C_1.\texttt{Text}_I \rangle$

Table 2.3: Some rules to detect relations in Example 1. C_1, C_2, C_3 is a sequence of identified components.

like ours, expert rules work quite well. The Stanford NLP parser also uses hand-written rules to extract grammatical relations between words [de Marneffe et al., 2006].

Even though our algorithm works fairly effectively, it is unable to detect all relations within a description, as is typically the case with NLP algorithms. In these cases, SmartSynth relies on recent advances in modern program synthesis to complete the relation set and generates the final script. Without the synthesis algorithm, SmartSynth would halt here and ask the user to refine the description.

Completing the Relation Set The goal here is to discover missing relations to form a complete *dataflow set* that can be turned into a script in a later phase. We use a specialized ranking scheme to find the most likely complete dataflow set among all possible sets.

Definition 2 (Complete Dataflow Set) *A set $G_{\mathcal{C}}$ of dataflow relations among components in $\mathcal{C}$ is* complete *if every sink of a component in $\mathcal{C}$ has one and only one relation associated with it. This dataflow set fully specifies how arguments are passed to all the sinks in $\mathcal{C}$.*

Algorithm 1 describes the process of completing the dataflow set. At the high-level, SmartSynth performs search over the dataflow graph to find the best dataflow sets $\mathcal{R}$ for the sinks $\tilde{T}$ that have not been assigned any values (line 5). SmartSynth uses the ranking scheme (represented by `Cost`) to guide the search towards more likely relations (lines 12, 14). It enforces structural constraints imposed by SmartScript by eliminating dataflow sets that do not conform to the language (line 11). To speed up the discovery

Algorithm 1: Discovering the most likely complete dataflow set

Input : component set $\mathcal{C}$, partial relation set $\tilde{R}$

Output : the most likely complete dataflow set R_{top}

```
1  begin
2      G_C ← BuildDataFlowGraph(C)
3      T_all ← {t | t is sink in G_C.V}                    /* already concretized sinks */
4      T_conc ← {t | ⟨s,t⟩ ∈ R̃}
5      T̃ ← T_all \ T_conc                                  /* to be concretized sinks */
6      R ← {ArbitraryRelationSet(R̃)}
7      Q ← {⟨R̃, T̃⟩}
8      while Q ≠ {} do
           /* worklist is not empty */
9          ⟨R,T⟩ ← Q.Dequeue()
10         if MinCost(R) ≤ MaxCost(R) then                 /* explore this branch */
11             if T = ∅ & GenCode(R) ∈ SmartScript then
12                 if Cost(R) < MaxCost(R) then R ← {R}
13                 else if Cost(R) = MaxCost(R) then R ← R ∪ {R}
14             else                                        /* explore child spaces */
15                 t ← choose(T)
16                 foreach (s,t) ∈ G_C.E_v do
17                     Q.Enqueue(⟨R ∪ ⟨s,t⟩, T \ t⟩)
           /* else do nothing:  prune this branch */
18     if R = ∅ then R_top ← ⊥
19     else if R = {R} then R_top ← R
20     else R_top ← PerformConversation(R)
21     return R_top
```

process, our algorithm implements the general branch-and-bound algorithm. It ignores a search sub-space if the minimum cost (`MinCost`) of all dataflow sets in that space is greater than that of the current set (lines 10 and 20).

One important aspect of Algorithm 1 is our specialized ranking scheme, which helps prioritize the search process toward the script that most likely matches the user's intention. We next discuss the ranking scheme.

Ranking Scheme Our high-level insight behind the ranking scheme is to give preferences to those scripts (1) that have greater coupling among the components, (2) that are more generally applicable, or (3) that require fewer user inputs.

We define a set of (negative) cost metrics to realize the above insight, each of which defines a less likely or less preferred feature of a script. The `Cost` function is simply the sum of the costs given by each of them:

- *Unused Variable Cost.* This metric gives a unit cost to each unused variable generated by SmartSynth. It promotes scripts that have better coupling among components.
- *Parameter Cost.* This metric gives a unit cost for each of parameters the script uses. This is because users normally do not want to configure the script when it runs. They often want fully automatic scripts.
- *Domain-Specific Cost.* This metric contains expert rules that capture what people usually do in practice. For example, in the messaging domain, we punish the script if the replied message is the same as the content of the received message.

This ranking scheme work quite effectively for our problem domain. However, in the presence of a large amount of user data, we can build a probabilistic ranking model. We are investigating this option for the TouchDevelop domain from its thousands of scripts contributed by its users.

Script Construction

At this point, SmartSynth has found a set of components and a complete set of dataflow relations. It now uses Algorithm 2 to construct the intended script. This process is the reverse of the normal parsing process in a compiler. A compiler breaks down a program into components and dataflow relations to enable optimizations and low-level code generation. In contrast, the SmartSynth synthesizes the script structure from its

Algorithm 2: Script Construction

Input : component set $\mathcal{C}$, dataflow set R

Output : an automation script

begin

  `/* generate temporary variables */`

  foreach *source* $\tau \in \mathcal{C}$ **do**

    `SymTable`$[\tau] \leftarrow$ new variable

  `/* assign sources' variables to sinks */`

  foreach $(\tau_s, \tau_t) \in \mathcal{C}.E$ **do**

    `SymTable`$[\tau_t] \leftarrow$ `SymTable`$[\tau_s]$

  $I \leftarrow$ live input variables in $\mathcal{C}$

  $E \leftarrow \perp$

  if $\exists e \in \mathcal{C} : e$ *is event* **then** $E \leftarrow e$

  $C \leftarrow \bigwedge_i \Pi_i$ where Π_i is a predicate in $\mathcal{C}$

  $T \leftarrow$ conversion functions used by C

  $M \leftarrow$ the rest of actions and conversion functions in g

  `/* generate foreach loop */`

  foreach $(\tau_s, \tau_t) \in g.E_v$ s.t. $\tau_s = List\langle\tau_t\rangle$ **do**

    $s \leftarrow$ the subgraph reachable from τ_t

    `SymTable`$[\tau_t] \leftarrow$ new variable

    `/* replace` s `by foreach stmt */`

    $M \leftarrow M[(\texttt{foreach SymTable}[\tau_t] \in \texttt{SymTable}[\tau_s]\texttt{do } s \texttt{ od}) \ / \ s]$

  $P \leftarrow \langle I, E, T, C, M \rangle$

  return P

building pieces (*i.e.*, the components and their dataflow relations). During this synthesis process, SmartSynth needs to: (1) generate code to pass arguments to APIs, (2) generate conditionals, and (3) generate loops.

SmartSynth generates a temporary variable for each API return value, and, based on the dataflow set identified in the previous steps, passes these variables as arguments to

other APIs (line 2-5). Next, it extracts the event and conditionals from the event and predicate components to form the guard of the script (lines 8-9). The remaining action APIs form the body.

The *most interesting aspect* of the script construction process is loop generation. SmartSynth generates a looping construct if there is a relation from a collection to an API parameter (lines 12-16). For example, the user may say “send Joe and John my current location." Although the user does not explicitly mention the word “loop”, it would be her intention to use a loop. In particular, SmartSynth detects that there is a relation between the group {Joe, John} and $\texttt{SendMessage.Number}_I$, and generates a loop to send messages to both Joe and John.

Having described the two technical components (*i.e.*, Component Discovery and Script Discovery), we now ready to formally describe the rest of SmartSynth’s architecture.

2.3.3 SmartSynth

Algorithm 3 implements all the components in the architecture. It realizes our observation that a SmartScript script can be identified from its essential components and their dataflow relations. Specifically, it uses NLP techniques to find components from the provided NL description (line 4) and to detect some of relations (line 5). It then invokes the synthesis algorithm to complete the dataflow relations (line 10), and to synthesize the final script in line 14. If nothing goes wrong, those steps correspond to the simplest possible workflow in SmartSynth.

However, since it is dealing with end users and natural language, SmartSynth faces many uncertainties. For example, users might give irrelevant and/or ambiguous descriptions. We discuss SmartSynth’s error handling mechanism next.

Handling Descriptions with Unsupported Functionalities It is possible that users ask for functionalities unsupported by SmartSynth. If the given description is in-scope, there exists a meaningful decomposition of the description into several chunks, and SmartSynth is able to map each of these chunks to a component. The graph in this case is a *connected* graph. On the other hand, if one or more parts of the description are out-of-scope, these parts cannot be mapped to any component(s). The graph in this case

Algorithm 3: The overall algorithm of SmartSynth.

Input : Natural language description d

Output : Automation script P, or request to refine (parts of) d

```
1  begin
2      G_d ← BuildDataFlowGraph(d)
3      if G_d is connected then
4          C ← BestMapping(G_d)
5          R̃ ← DetectRelations(C)
6          while MappingScore(⟨C, R̃⟩) < δ do
               /* mapping score too low */
7              C ← NextBestMapping()
8              R̃ ← DetectRelations(C)
9          if C ≠ ∅ then                                   /* mapping exists */
10             R ← CompleteRelations(⟨C, R̃⟩)
11             if R > 1 then R ← QuestionAnswering(R)      /* equally ranked sets */
12             else R ← R.First()
13             return GenerateCode(⟨C, R⟩)
14         return AskForRefinement(d)                  /* refine the whole description */
15     else                                             /* refine nonmappable phrases */
16         l ← DisconnectedSubGraphs(G_d)
17         return AskForRefinement(l)
```

is *disconnected*. SmartSynth asks the user to refine the phrases corresponding to the disconnected subgraphs that cannot be mapped to anything (line 17 in Algorithm 3, step 2 in Figure 2.1). Figure 2.6 visualizes this case.

Handling Ambiguous NL Descriptions Although SmartSynth uses multiple mapping features to increase the precision of the component mapping process, there are cases where these features cannot promote the best component mapping properly. This is unavoidable because we cannot precisely model natural language. SmartSynth resolves such cases via the feedback loop on lines 6-8 in Algorithm 3 (and Steps 3-4 in Figure 2.1).

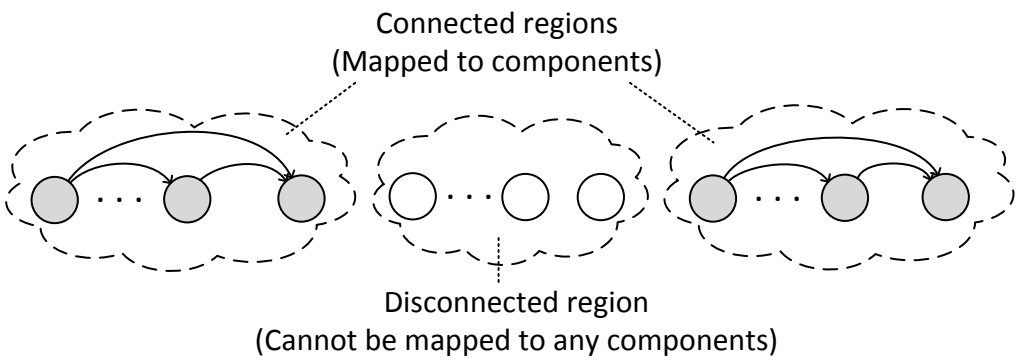


Figure 2.6: A region in the description that cannot be mapped to any component. SmartSynth reports its phrase to the user to clarify.

It evaluates the quality of the component mapping provided by the mapping algorithm and repeatedly generates other interpretations of the NL description, if the current one is unsatisfactory.

Handling Equally Likely Scripts When the ranking scheme cannot differentiate the top dataflow sets, SmartSynth has multiple equally ranked scripts at the top. In such cases, SmartSynth performs interactive conversation with the user to eliminate undesired scripts (line 12 in Algorithm 3, and Step 5 in Figure 2.1). It generates a sequence of *distinguishing* multiple-choice questions based on these candidates and presents them to the user, one question at a time. Her answers help SmartSynth capture more constraints and eliminate unintended dataflow sets.

The intuition behind the algorithm is that each question corresponds to a sink that receives different values in those dataflow sets, and the answer choices are those values that can be assigned to the sink. When the user selects an answer, SmartSynth eliminates those dataflow sets that do not correspond to the selected value.

Because the sink set is finite and known *a priori* (sinks are parameters of pre-defined APIs), we are able to build a database that stores an NL question for each of the sinks. Similarly, we define NL representations for all the sources. SmartSynth generates questions and answers by looking up the values of the corresponding sinks and sources in the database. For example, for the sink `SendMessage.Number`$_I$, SmartSynth generates

the question *"Who do you want to send the SMS to?"*. Suppose this sink is related to two sources, `MessageReceived.Number`$_O$ and `111-1111`. They will be presented as two answer choices *"(A) The sender of the received SMS"* and *"(B) The phone number 111-1111"*.

2.4 Evaluation

In this section, we evaluate the effectiveness of SmartSynth in discovering the right set of components and their dataflow relations. We also compare the overall performance of SmartSynth and the system that employs only NLP techniques.

2.4.1 Setup

We implemented SmartSynth in C#. We ran all our experiments on a machine with an Intel® i7 2.66 GHz CPU and 4 GB of RAM. SmartSynth uses the Stanford NLP parser [Klein and Manning, 2003] wrapped under a public web service [LeBlanc, 2015].

We compile the benchmark from task descriptions found in help forums of Tasker, App Inventor, and TouchDevelop. Among the 70 task descriptions that we found, seven cannot be expressed in our language SmartScript. Therefore, we remark that SmartScript was able to express 90% of the kind of real-life examples that it was designed for. All the inexpressible descriptions that we found required synchronization between different tasks, possibly through the use of some shared global variables or control-flow among events. To illustrate, we describe below one of them; the other ones are similar.

Example 5 [Inexpressible in SmartScript] *If there is an unread SMS and the user has not turned the screen on, remind her every 5 minutes.*

This script has two synchronized tasks, which may be synchronized using a global status variable denoting whether or not the screen has been turned on. We plan to extend our language to allow task synchronization as future work.

Of the remaining 63 task descriptions that are expressible in SmartScript, 13 are similar to another one in the set. Table 2.4 shows the distinct 50 tasks. To collect NL descriptions for those benchmark tasks, we conducted an online user study involving freshman students. We provided a one page document describing their task. We divided

the benchmark into two sets that are roughly equivalent in terms of complexity. The system randomly assigned a student to one set. For each problem in an assigned set, we gave the student its desired script in SmartScript as the specification, and asked the student to give several NL descriptions that match the given script.

Eleven students participated in the study, giving a total of 725 distinct descriptions for our 50 benchmark tasks. Among these given descriptions, 640 match their respective specifications. We used these 640 descriptions to evaluate SmartSynth.

2.4.2 Experimental Results

In Table 2.4, column "Com" shows, for each task, the total number of components that are identified from the description. The next two columns divide these components into APIs and entities. Column "Rel" corresponds to the number of dataflow relations for each task, and "DRel" shows how many relations on average the NLP-based algorithm can extract from various descriptions. The last two columns show the average parsing time by the Stanford NLP parser and the average total synthesis time under ten runs. Note that the parsing time includes the communication overhead to the web service provider. It dominates the total time by SmartSynth.

Component Discovery We first evaluate the effectiveness of SmartSynth in mapping descriptions into components. Figure 2.7 shows the precision of this process under increasingly sophisticated configurations. The baseline algorithm (regular expression for entities and bag-of-words for APIs) produces the right component set 64.3% of the time. The algorithm combining all features achieves a 77.1% precision (the fourth column). These results indicate that the considered features work harmoniously to improve the overall precision.

Component Discovery (with Feedback Loop) Since mapping from descriptions to components tends to be ambiguous, SmartSynth utilizes the feedback from the synthesis algorithm (see Section 2.3.3) to gain higher mapping confidence. The last column of Figure 2.7 shows that the aid from the feedback loop helps improve the mapping precision to 90.3%. This proves that the synthesis feedback is very effective in disambiguating

ID	Brief Description	Com	API	Ent	Rel	DRel	t_{NLP}	t_{total}
1	Reply current loc., read msg's number and content	12	6	6	7	4.13	3,151	3,325
2	Reply current loc. and batt. level upon receiving a code	11	5	6	7	3.75	2,705	3,206
3	Phone locator by SMS	10	5	5	8	4.6	4,376	5,737
4	Text a friend when come nearby	8	3	5	4	2	1,102	1,137
5	Silent at night but ring for important contacts	8	3	5	4	3.14	2,544	2,617
6	Speak weather in the morning	7	3	4	3	1.67	388	405
7	Take a picture, add the location and upload to Facebook	7	4	3	4	3	837	875
8	Auto response to SMS at night	7	3	4	4	3.61	2,071	2,134
9	Alert and turn off connections when battery is low	7	6	1	1	1	2,402	2,878
10	Text wife when nearly finish a long commute	6	2	4	3	2.94	1,439	1,481
11	Send SMS at a particular time	6	2	4	3	2.60	1,173	1,210
12	Send an email when leaving the office	6	2	4	3	2.67	1,077	1,108
13	Send a message when kids are dropped at school	6	2	4	3	2.88	1,644	1,689
14	Send to a list of friends current location every 15 minutes	6	3	3	4	4	788	817
15	Reply busy message if connected to car's bluetooth	6	3	3	3	2.19	2,723	2,811
16	Turn off GPS when connected to home wifi	5	2	3	1	1	737	779
17	Increase display timeout while running some apps	5	2	3	3	2	948	983
18	Auto answer calls when headset is connected	5	3	2	1	1	1,339	1,395
19	Block calls from a group of users	5	2	3	2	1.42	715	744
20	Connect to wifi when disconnected from car's bluetooth	5	2	3	2	2	1,234	1,292
21	Find direction to the place a photo is taken	5	3	2	3	2.07	759	787
22	Turn off wifi and GPS at night	5	3	2	2	2	653	688
23	Ring loudly for important contacts	5	2	3	3	3	1,090	1,128
24	Lower volume when a loud friend is calling	5	2	3	3	3	1,539	1,584
25	Take a picture and add current time to it	5	3	2	3	1.88	486	507
26	Handsfree texting	5	3	2	2	1.36	1,246	1,290
27	Broadcast the received message from a group	5	2	3	4	3	827	859
28	Read the received msg while connected to car's bluetooth	5	3	2	2	1.90	1,122	1,176
29	Mute the phone if is in the theater	4	2	2	1	0.89	467	485
30	Disable screen rotation at night	4	2	2	2	2	413	436
31	Mute the phone if is in meeting	4	2	2	1	0.92	344	359
32	Turn on data service for apps that require data	4	2	2	2	1.76	1,501	1,547
33	Turn on GPS when apps requiried GPS are run	4	2	2	2	1	752	781
34	Send current location to a friend via SMS	4	2	2	3	2	272	284
35	Send an SMS with a secret code to trigger an alarm	4	2	2	1	1	912	947
36	Repeat caller name	4	3	1	2	1	441	459
37	Alert when receive an email with important subject	4	2	2	1	1	759	788
38	No text while driving	4	2	2	2	1.55	1,824	1,879
39	Ask to take the call if connected to car's bluetooth	4	3	1	1	1	1,702	1,776
40	Launch Pandora when the headphone is plugged	3	2	1	1	1	310	327
41	Reduce the volume when headset is plugged in	3	2	1	1	1	361	379
42	Keep screen awake when using the keyboard	3	2	1	1	1	414	434
43	Maximize screen brightness for calls	3	2	1	1	1	406	424
44	Turn off ringer by turning the phone down	3	3	0	0	0	718	757
45	Lay the phone down to switch to speaker	3	3	0	0	0	792	827
46	Show direction from current loc. to previously saved loc.	3	2	1	2	2	661	696
47	Open the keyboard and start texting	2	2	0	0	0	199	211
48	Unplug headset to pause media player	2	2	0	0	0	219	231
49	Play alert sound when battery is full	2	2	0	0	0	221	233
50	Send a text message to a group	2	1	1	3	2	387	402

Table 2.4: Characteristics of the benchmarks extracted from smartphone help forums.

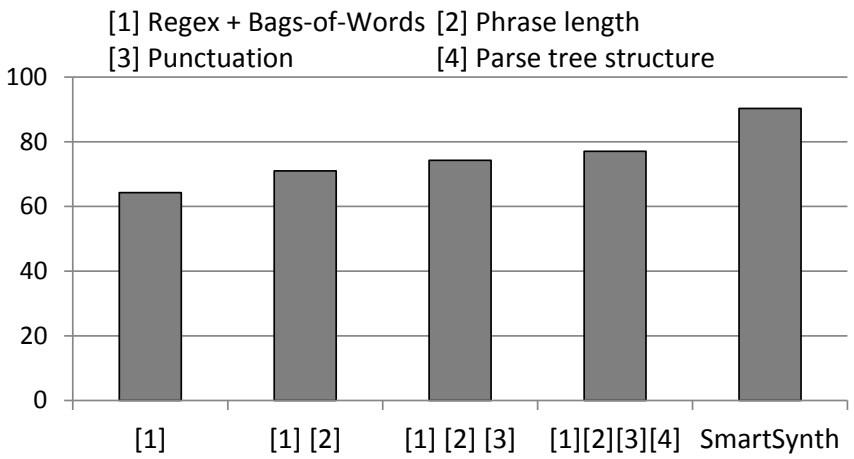


Figure 2.7: The result of mapping phrases to components under increasingly sophisticated configurations.

mapping candidates. Note that we measure precision by counting the number of sets that are mapped *entirely* correctly, while in other settings, *individual* components are counted. Their performance might be reduced when measured in our context.

Dataflow Relation Detection (with NLP Technique) We now evaluate the capability of SmartSynth in finding dataflow relations among APIs and entities under the ideal case, *i.e.* when the components are mapped correctly. Figure 2.8 shows some statistics on the number of dataflow relations that the rule-based relation detection algorithm can detect on benchmark tasks that are grouped by the number of relations.

These results indicate that the rule-based relation detection algorithm works quite effectively for tasks that have few relations. Specifically, it found almost all relations in the 1-relation tasks and nearly 86% relations in those having four. However, the rule-based algorithm faces problems when the tasks contain more relations. In particular, it could not detect nearly half of the relations in tasks having 7 or more relations. In order to resolve this, SmartSynth needs further support. Next, we measure the effectiveness of the synthesis-based algorithm in assisting the rule-based algorithm to complete those missing relations.

Dataflow Relation Completion (with Synthesis Technique) Figure 2.9 shows that in nearly a quarter of the cases, the rule-based algorithm failed to detect all the dataflow

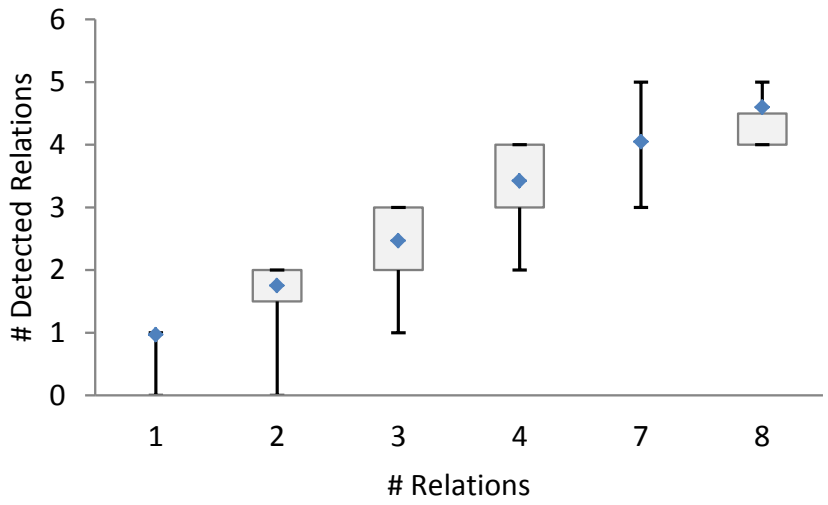


Figure 2.8: The number of dataflow relations detected by the rule-based algorithm for various tasks grouped by relation number.

relations in the solution set. Furthermore, the failure rate is proportional to the number of dataflow relations in the task descriptions. These indicates that pure NLP techniques do not scale well with complicated tasks. By employing the relation completion algorithm inspired by program synthesis area, SmartSynth was able to complete the dataflow sets and synthesize the desired scripts. In general, the synthesis-inspired algorithm is more useful with more complicated scripts, where the failure rate of detecting all relations using only NLP techniques is close to 100%.

Overall Performance Should SmartSynth only use NLP techniques and not employ the synthesis-inspired algorithm, it would suffer from the ambiguity of mapping descriptions to components and the uncertainty of extracting their relations. As a result, it might not be able to generate the user's intended script. Evaluated on our data, such system only returns the intended script 58.7% of the time (mostly with simple descriptions).

After we enable techniques inspired by program synthesis area, SmartSynth both improves its component identification (from the feedback loop) as well as its relation discovery (from additional relations found using the relation completion algorithm). SmartSynth achieves the 90% accuracy by combining the strength of the two research areas.

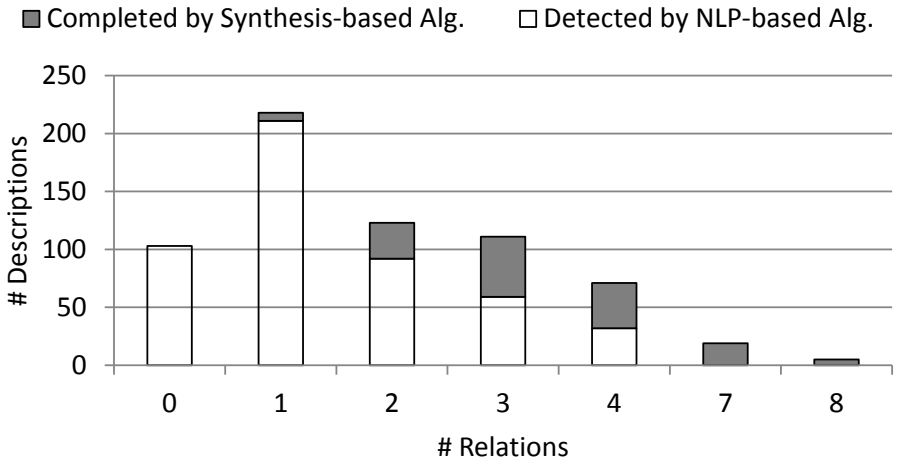


Figure 2.9: The number of descriptions whose dataflow relations are entirely detected by the rule-based algorithm (total 487) vs. those that require completion by the completion algorithm (total 153), grouped by the number of relations.

SmartSynth works quite well once it combines the strength of both NLP and synthesis communities. However, it might fail to generate the intended script sometimes. We discuss such cases next.

2.4.3 Limitations

Given the NL description is grammatically correct and in-scope, there are two possibilities for SmartSynth to fail in generating the intended script. The first possibility is when users ask for a script that is not expressible in SmartScript. When designing SmartScript, we carefully balanced the trade-offs between language expressiveness and synthesis effectiveness. We can extend SmartScript to cover more scripts, but this also expands the search space, and may make the completion algorithm less scalable and the ranking scheme less effective. Nonetheless, SmartScript is able to express 90% of automation scripts that users care about (see Section 2.4).

The second possibility is when SmartSynth captures the intent from the NL description incorrectly. This is a common problem for any system that handles NL. Although it cannot completely avoid incorrect mappings, SmartSynth alleviates this problem by empowering the feedback loop, which estimates script likelihood and repeatedly generates alternative interpretations of the provided description, if the current script is not likely.

As with any programming paradigms that use examples or natural language descriptions, the correctness of the synthesized script in SmartSynth needs to be checked by the user. Another limitation of SmartSynth is its tight integration with the API set and the English language. To support any new APIs (or new languages), we have to add new rules and possibly new ranking metrics. Interesting future work is to learn these rules and ranking metrics automatically from training data. Currently, we only support small tasks that can be described in one-liners. We are considering expanding SmartScript to support task descriptions that use multiple sentences.

Chapter 3

FlashExtract: Synthesizing Data-Extraction Program from Examples

The IT revolution over the past few decades has resulted in two significant advances: the digitization of massive amounts of data and widespread access to computational devices. However, there is a wide gap between access to rich digital information and the ability to manipulate and analyze it.

Information is available in documents of various types such as text/log files, spreadsheets, and webpages. These documents offer their creators great flexibility in storing and organizing hierarchical data by combining presentation/formatting with the underlying data model. However, this makes it extremely hard to extract the underlying data for several tasks such as data processing, querying, altering the presentation view, or transforming data to another storage format. This has led to development of various domain-specific technologies for data extraction. Scripting languages like Perl, Awk, Python have been designed to support string processing in text files. Spreadsheet systems like Microsoft Excel allow users to write macros using a rich built-in library of string and numerical functions, or to write arbitrary scripts in Visual Basic/.NET languages. Web technologies like Xquery, HTQL, XSLT can be used to extract data from webpages, but this has the additional burden of knowing the underlying document structure.

Existing programmatic solutions to data extraction have three key limitations. First, the solutions are domain-specific and require knowledge/expertise in different technologies for different document types. Second, they require understanding of the entire underlying document structure including the data fields that the end user is not interested in extracting and their organization (some of which may not even be visible in the presentation layer as in case of webpages). Third, and most significantly, they require knowledge of programming. The first two aspects create challenges for even programmers, while the third aspect puts these solutions out of reach of the vast majority of business end users who lack programming skills. As a result, users are either unable to leverage access to rich data or have to resort to manual copy-paste, which is both time-consuming and error prone.

In this chapter, we address the problem of developing a uniform end-user friendly interface to support data extraction from semi-structured documents of various types. Our methodology includes two key novel aspects: a uniform user interaction model across different document types, and a generic inductive program synthesis framework.

Uniform and End-user Friendly Interaction Model Our extraction interface supports data extraction via examples. The user initiates the process by providing a nested hierarchical definition of the data that he/she wants to extract using standard structure and sequence constructs. The user then provides examples of the various data fields and their relationships with each other. An interesting aspect is that this model is independent of the underlying document type. This is based on our observation that different document types share one thing in common: a two-dimensional presentation layer. We allow users to provide examples by highlighting two-dimensional regions on these documents. These regions indicate either the fields that the user wants to extract or structure/record boundaries around related fields.

Inductive Program Synthesis Framework To enable data extraction from examples, we leverage inductive program synthesizers that can synthesize scripts from examples in an underlying domain-specific language (DSL). The key technical contribution of this chapter is an inductive program synthesis framework that allows easy development of

inductive synthesizers from mere definition of DSLs (for data extraction from various document types). We describe an expressive algebra containing four core operators: map, filter, merge, and pair. For each of these operators, we define its sound and complete generic inductive synthesis algorithms parameterized by the operator's arguments. As a result, the synthesis designer simply needs to define a DSL with two features: (a) It should be expressive enough to provide appropriate abstractions for data extraction for the underlying document type, (b) It should be built out of the operators provided by our core algebra. The synthesis algorithm is provided for free by our framework. This is a significant advance in the area of programming by examples, wherein current literature [Gulwani et al., 2012, Gulwani, 2012] is limited to domain-specific synthesizers.

3.1 Motivating Examples

In this section, we motivate some data extraction tasks across different document types and illustrate how FlashExtract can be used to automate the various tasks from examples.

Text Extraction

Example 6 *Consider the text file in Figure 3.1 (taken from a help forum thread*[1]*) that contains a sequence of sample readings, where each sample reading lists various "analytes" and their characteristics (analyte intensities). The user wants to extract the highlighted fields into an Excel spreadsheet in order to perform some analysis. Accomplishing this task by creating a one-off Perl script appears daunting, especially for a non-programmer.*

Suppose the user only wants to extract the *analyte names* (magenta regions starting with instance "Be") and their *mass* (violet regions starting with instance "9"). The user starts with the analyte names. She highlights the first two regions "Be" and "Sc" as examples in magenta color. FlashExtract synthesizes an extraction program and uses it to highlight all other analyte instances. The user inspects and approves the highlighted result because it matches the intended sequence. She then moves to the mass field and repeats the process with violet color. FlashExtract can now automatically relate

[1] *http://www.excelforum.com/excel-programming/608284-read-txt-file.html*

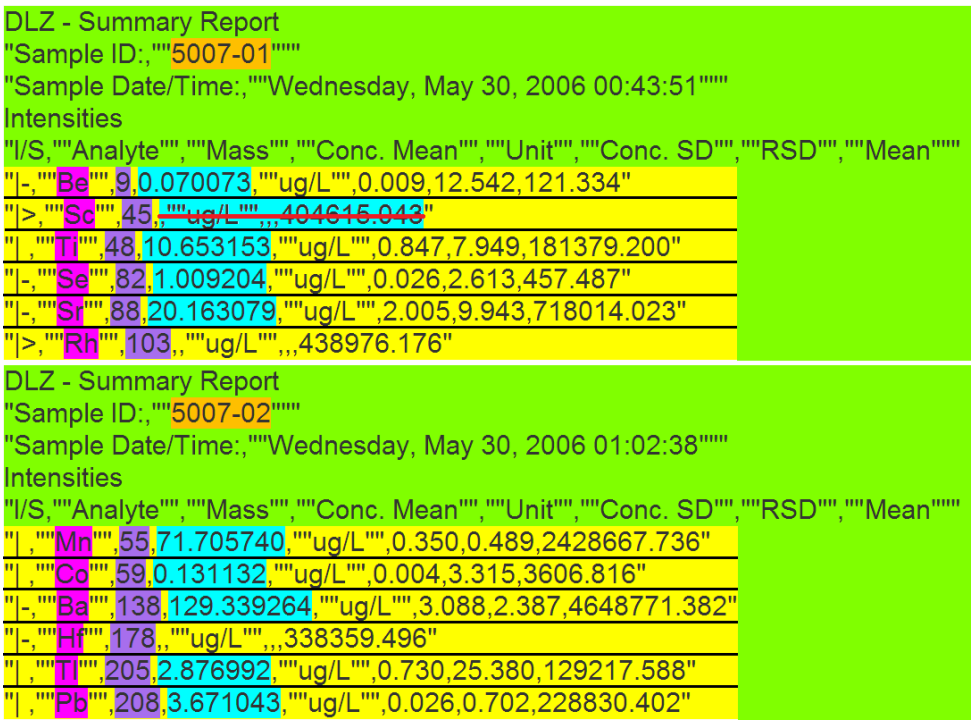


Figure 3.1: Extracting data from a text file using FlashExtract.

the respective instances from the magenta sequence and the violet sequence, and can generate a two-column Excel table if desired.

Now suppose the user also wants to extract the *conc. mean* (blue regions including instance "0.070073"). After one example, FlashExtract mistakenly includes the string ",""ug/L,,,404615.043" to the result (this should be null). To exclude this region, the user draws a red line through it to mark it as a *negative example*, and FlashExtract refines the learning with the new information. It then produces the correct result and the user stops providing any further examples. Although the third sequence contains fewer regions, FlashExtract is still able to relate it to the other two automatically because it is the only sequence containing null regions.

In case FlashExtract cannot relate different field regions, or does so incorrectly, the user can intervene by marking a structure boundary around related regions. For instance, the user may highlight the first yellow region as the structure boundary for the intensity of the first analyte. FlashExtract is able to infer similar yellow regions that group other

intensities. If the user wants to further organize the analyte intensities into different samples, she creates the outer green regions. The user can then add the sample ID (orange field, such as “5007-01”) to these green structures.

Once the highlighting process has been completed, the user can obtain the data (in different formats such as XML file or Excel table) and its associated data extraction program. The user may run the program on other similar files to extract data in the same output format without giving any additional examples.

Note that the user can extract data in any field order (we only demonstrated one such order). For example, the green regions can be highlighted before the yellow regions, which in turn can be highlighted before the violet regions. The top-down order is generally recommended and has higher chance of success (because an inner field knows who is its parent). Furthermore, the highlighting does not necessarily need to follow the actual file structure; it just needs to be consistent. For instance, the user may want the green structures to begin at “"Sample ID”, or the yellow structures to end in the middle of the lines, say before “ug/L”.

We can combine FlashExtract with existing end-user programming technologies to create new user experiences. For instance, integration of FlashExtract with FlashFill [Gulwani, 2011] allows users to both extract and transform the highlighted fields using examples, and possibly push the changes back to the original document. As an example, after highlighting using FlashExtract, the user can easily change the precision of *conc. mean* (blue field) or the casing of *analytes* (magenta field) using FlashFill.

Webpage Extraction

Example 7 *Google Scholar website (`http://scholar.google.com`) has author pages containing list of all publications. A publication consists of its title, list of authors, venue, number of citations, and year of publication. Figure 3.2 shows an excerpt from a page of the researcher Mandana Vaziri.*

Suppose the user wants to find all publication titles in which Dr. Vaziri is the first author. She can use FlashExtract to extract the publication title (blue) and just the first author (magenta) fields into an Excel spreadsheet, where she can utilize the native

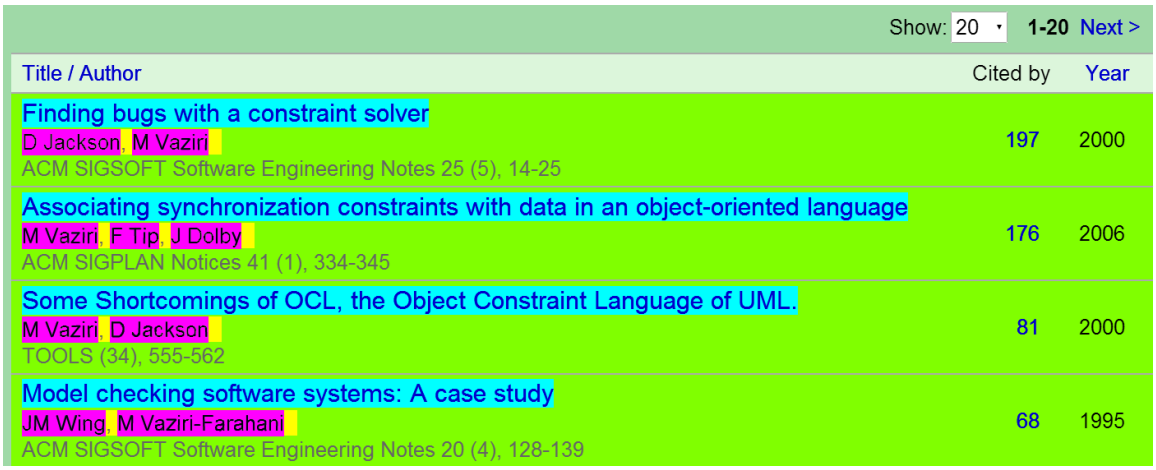


Figure 3.2: Extracting data from a Google Scholar webpage.

spreadsheet functionality to sort by first author field. A key design principle behind FlashExtract is to provide a uniform interaction model independent of the underlying document type. The user interacts with webpages exactly as with text files. That is, the user gives examples for the desired fields by using colored highlighting over the rendered webpage. She does not need to understand the underlying structure of the webpages (as is required in the case of querying languages such as XPath, XQuery).

Now suppose the user wants to extract publication titles along with the list of all authors. Extracting list of all authors is technically challenging because the comma separated list of all authors is represented as a string in a single `div` tag. However, we can use FlashExtract to highlight each author individually. Its underlying DSL is expressive enough to allow extracting list of regions in a text field of a single HTML node.

FlashExtract can be used to group a publication and all its authors together. The user may make this relation explicit by highlighting green regions. The same applies to the yellow regions that group author sequences. Once FlashExtract has produced the program, the user may run it on other scholar pages to perform the same task for other authors.

	Amount	Investigator
Advanced Materials Processing	Analysis Center (AMPAC)	
	$24,500.00	Coffey, Kevin R.
Subtotal	$24,500.00	
Biomolecular Science Center		
	$155,458.00	Khaled, Annette R.
	$71,777.00	Zervos, Antonis S.
Subtotal	$227,235.00	
Education (ED)		
	$272,349.00	Daire, Andrew
	$57,500.00	Little, Mary E.
	$272,349.00	Young, Mark E.
Subtotal	$602,198.00	

Figure 3.3: Extracting data from a semi-structured spreadsheet.

Spreadsheet Extraction

Example 8 *Consider the semi-structured spreadsheet* `"Funded - February#A835C.xlsx"` *from the EUSES benchmark [Ii and Rothermel, 2005] shown in Figure 3.3. Suppose the user wants to (a) add up the values in the Amount column (excluding the values in the subtotal rows), and (b) plot values in the Amount column grouped by department name.*

The user highlights few examples of the amount field (magenta), department field (yellow), and the record boundaries (green). FlashExtract is then able to highlight all other similar instances and creates a new relational table view. For task (a), the user can now simply add up all the values in the amount column by using the native spreadsheet SUM function over the new relational view (instead of having to write a complicated conditional arithmetic macro over the original semi-structured spreadsheet view). Task (b) is easily accomplished using the popular "Recommended Charts" feature in Excel 2013, wherein Excel automatically suggests relevant charts over user's data. Note that this feature only works when the data is properly formatted as a single relational table—it does not work on the original semi-structured spreadsheet.

$$
\begin{array}{rcl}
\textsf{Schema } M & ::= & S \mid T \\
\textsf{Structure } T & ::= & \texttt{Struct } (identifier: E_1, \ldots, identifier: E_n) \\
\textsf{Element } E & ::= & f \mid S \\
\textsf{Sequence } S & ::= & \texttt{Seq}(f) \\
\textsf{Field } f & ::= & [color]\,\tau \mid [color]\,T
\end{array}
$$

Figure 3.4: The language of schema for extracted data.

If the user later decides to also extract the investigator name (blue), she can simply provide an example. As before, once all interactions are recorded, the output view can be automatically updated if the user continues to edit and maintain the original ad-hoc format consistently.

3.2 User Interaction Model

Our user interaction model for data extraction requires the user to provide an *output data schema* and highlight examples of *regions* (in the document) that contain the desired information. The final result is a *schema extraction program* that extracts the desired data from the document as an instance of the output schema. We next define these aspects in more detail.

3.2.1 Output Schema

The final product of an extraction task is a nested organization of the extracted data using standard structure and sequence constructs. Figure 3.4 defines the language for the output schema. The output schema is either a sequence S over some field f, or a structure T with named elements $E_1, \ldots, E_n$. Each element E corresponds to either a field f or to a sequence S. Each field f is either an atomic type τ (also referred to as a *leaf field*) or a structure T. Each field f is associated with a unique color (denoted $f.\texttt{Color}$).

For example, the schemas for the two extraction tasks discussed in Example 6 are

presented below. The first one represents a sequence of yellow structures, each of which contains a magenta Analyte field and a violet Mass field. The second one represents organization of many more fields referenced in the task illustrated in Figure 3.1.

(1) `Seq`$\Big($[yellow] `Struct`$($Analyte : [magenta] Float, Mass : [violet] Int$)\Big)$

(2) `Seq`$\Big($[green] `Struct`$($SampleID : [orange] String,
Intensities : `Seq`$($[yellow] `Struct`$($
Analyte : [magenta] String,
Mass : [violet] Int,
CMean : [blue] Float$)))\Big)$

Note that the schema language does not allow a sequence to be directly nested inside another sequence. It requires a colored structure construct in between them. The structure serves as the boundary for the learning of the inner sequence.

Definition 3 (Ancestor) *We say that a field f_1 is an* ancestor *of field f_2 if f_2 is nested inside f_1. For notational convenience, we say that $\perp$, which stands for the top-level data schema definition, is an ancestor of every field. Additionally, f_1 is a* sequence-ancestor *of f_2 if there is at least one sequence construct in the nesting between f_1 and f_2. Otherwise, f_1 is a* structure-ancestor *of f_2. We say that $\perp$ is an ancestor of every other field.*

In the second schema above, the yellow structure is the structure-ancestor of leaf fields Analyte, Mass, and CMean. The top-level green structure is the sequence-ancestor of the yellow structure. We categorize ancestral relationship in order to invoke appropriate synthesis algorithms.

3.2.2 Regions

Definition 4 (Region) *A* region *R of a document is some two-dimensional portion over the visualization layer of that document that the user is allowed to highlight in some color. We use the notation f-*region *to denote any region that the user highlights in f.`Color`. Any region R that is associated with a leaf field (also referred to as a* leaf region*) has some value associated with it, which is denoted by R.`Val`. For a document D, we use the notation D.`Region` to denote the largest region possible in D.*

In case of text files, any region is represented by a pair of two character positions within the file and consists of all characters in between (these positions may or may not be within the same line). The value of such a region is the string of all characters in between those positions.

In case of webpages, a leaf region is represented by either an HTML node (the value of such a region is the text value associated with that node) or a pair of character positions within the text content of an HTML node (the value of such a region is the string of all characters between those positions). A non-leaf region is represented by an HTML node.

In case of spreadsheets, a leaf region is represented by a single cell (and its value is the cell's content), while a non-leaf region is represented by a pair of cells (and consists of the rectangular region determined by those cells).

Definition 5 (Highlighting) *A* highlighting *CR of a document D is a collection of colored regions in D. It can also be viewed as a function that maps a color to all regions of that color in D. We say that a highlighting CR is* consistent *with a data scheme M if the following conditions hold.*

- *For any two regions (in CR), either they don't overlap or one is nested inside the other.*
- *For any two fields f_1 and f_2 in M such that f_1 is an ancestor of f_2, each f_2-region R_2 is nested inside some f_1-region R_1.*
- *For any two fields f_1 and f_2 in M such that f_1 is a struct-ancestor of f_2, there is at most one f_2-region inside a f_1-region.*
- *For every leaf field f in M, the value of any f-region in CR is of type f.*

3.2.3 Schema Extraction Program

FlashExtract synthesizes extraction programs for individual fields and combines them into a schema extraction program following the structure imposed by the output schema. FlashExtract also leverages the schema's structure to simplify the learning of individual fields. In particular, it relates a field f to one of its ancestors, whose extraction program (in case of a non-$\perp$ ancestor) defines learning boundaries for f (*i.e.*, each f-region must reside inside one of these boundaries).

Algorithm 4: Execution semantics of extraction programs.

1 **function** Run *(schema extraction program Q, schema M, document D) : schema instance* **is**
2 $\quad$ CR $:= \emptyset$
3 $\quad$ **foreach** *field f in M in top-down topological order* **do**
4 $\quad\quad$ $\tilde{R} :=$ Run$(Q(f), D,$ CR$)$
5 $\quad\quad$ CR $:=$ CR $\cup \{(f.$Color$, R) \mid R \in \tilde{R}\}$
6 $\quad$ **if** *CR is inconsistent with M* **then return** $\perp$
7 $\quad$ **else return** Fill$(M, D.$Region$)$
8 **function** Run *(extraction program (f', P) of field f, document D, highlighting CR): f-regions* **is**
9 $\quad$ $\tilde{R}' := (f' = \perp)?\ \{D.$Region$\} :$ CR$[f'.$Color$]$
10 $\quad$ **return** $\bigcup_{R' \in \tilde{R}'} [\![P]\!] R'$ /* execute P on R' */

Definition 6 (Extraction Programs) *A* schema extraction program *Q for a given schema M is represented as a map from each field f in M to a field extraction program, denoted $Q(f)$. A* field extraction program *of field f is a pair (f', P), where f' (possibly $\perp$) is some ancestor field of f and P is either a* SeqRegion program *that extracts a sequence of f-regions from inside a given f'-region (in case f' is a sequence-ancestor of f), or is a* Region program *that extracts a single f-region from inside a given f'-region (in case f' is a struct-ancestor of f).*

The execution semantics of a schema extraction program is defined in Algorithm 4. FlashExtract executes the field extraction program corresponding to each field f in a top-down order and updates the document highlighting CR using the returned list of f-regions $\tilde{R}$ (lines 3–5). For each field f, it first finds the set of regions $\tilde{R}'$ determined by the ancestor f' (line 9), and then computes all f-regions by executing the field extraction program on each region R' in $\tilde{R}'$ (line 10). Once CR has been fully constructed, it generates a schema instance from the nesting relationship defined in the output schema M, using the Fill function (line 7).

$$\texttt{Fill}\Big(\texttt{Struct}\ (id_1\ E_1, \ldots, id_n\ E_n), R\Big) = \textbf{new}\ \texttt{Struct}\Big(\{id_1 = \texttt{Fill}(E_1, R), \ldots, id_n = \texttt{Fill}(E_n, R)\}\Big)$$

$$\texttt{Fill}\Big(\texttt{Seq}(f), R\Big) = \textbf{new}\ \texttt{Seq}\Big(\texttt{Map}(\lambda R' : \texttt{Fill}(f, R'), \texttt{Subregions}(R, \texttt{CR}[f.\texttt{Color}])\Big)$$

$$\texttt{Fill}([color]\ \texttt{Val}, R) = \texttt{Subregion}(R, \texttt{CR}[color]).\texttt{Val}$$

$$\texttt{Fill}([color]\ T, R) = \texttt{Fill}(T, \texttt{Subregion}(R, \texttt{CR}[color]))$$

$$\texttt{Fill}(\_, \perp) = \perp$$

Figure 3.5: Semantics of `Fill`.

Figure 3.5 defines the semantics of `Fill` recursively. Each definition takes a schema construct and a region corresponding to one of its ancestor fields, and returns a construct instance by recursively applying `Fill` functions on its descendants. $\texttt{CR}[c]$ returns all regions whose color is c. $\texttt{Subregions}(R, \tilde{R})$ returns the ordered set of regions from $\tilde{R}$ that are nested inside R. $\texttt{Subregion}(R, \tilde{R})$ returns the region from $\tilde{R}$ that is nested inside R; if no such region exists, $\perp$ is returned. Note that if `CR` is consistent with M, there is at most one such region. We assume the presence of an API for checking the nestedness of two regions.

3.2.4 Example-based User Interaction

Having defined all necessary concepts, we are now ready to discuss the way a user interacts with FlashExtract to extract their desired data. The user first supplies the output data schema. Then, for each field f in the schema (in an order determined by the user), the user simply provides sufficient number of examples of field instances of field f by highlighting appropriate regions in the document using $f.\texttt{Color}$. Our user interface supports standard mouse click, drag, and release gestures.

When the user provides examples for a field f, FlashExtract synthesizes a field extraction program for field f (using Algorithm 5) that is *consistent* with the provided examples, and executes it to identify and highlight other regions in $f.\texttt{Color}$. (See Definition 7 for a formal notion of consistency.) If the user is happy with the inferred

Algorithm 5: Synthesize a field extraction program.

```
1  function SynthesizeFieldExtractionProg (Document D, Schema M, Highlighting CR, Field
     f, Regions R̃1, Regions R̃2) is
       /* R̃1, R̃2 denote positive, negative instances */
2      foreach ancestor field f′ of f in schema M do
3          if f′ isn't materialized ∧ f′ ≠ ⊥ then continue
4          R̃ := (f′ = ⊥)? {D.Region} : CR[f′.Color]
5          if f′ is a sequence-ancestor of f then
6              ex := ∅
7              foreach R ∈ R̃ s.t. Subregions(R, R̃1 ∪ R̃2) ≠ ∅ do
8                  ex := ex ∪ {(R, Subregions(R, R̃1), Subregions(R, R̃2))}
9              P̃ := SynthesizeSeqRegionProg(ex)
10         else /* f′ is a structure-ancestor of f */
11             ex := ∅
12             foreach R ∈ R̃ s.t. Subregion(R, R̃1) ≠ ⊥ do
13                 ex := ex ∪ {(R, Subregion(R, R̃))}
14             P̃ := SynthesizeRegionProg(ex)
15         foreach P ∈ P̃ do
16             CR′ := CR ∪ {(f.Color, R) | R ∈ ⟦P⟧R′, R′ ∈ R̃}
17             if CR′ is consistent with M then return (f′, P)
18     return ⊥
```

highlighting, she can commit the results (the field f is said to have been *materialized* at that point of time), and then proceed to another (non-materialized) field. Otherwise, the user may provide any additional examples.

We say that a field f has been *simplified* if there exists a materialized field f' such that f' is a structure-ancestor of f. The examples for a non-simplified field consist of positive instances and optionally negative instances of regions that lie completely within the regions of the immediate ancestor field that has been materialized. If the user is not

happy with the inferred highlighting, the user provides additional positive instances (of regions that FlashExtract failed to highlight) or negative instances (of unintended regions that FlashExtract highlighted) and the synthesis process is repeated. The examples for a simplified field consist of at most a single positive instance (possibly null) inside each region of the immediate ancestor field that has been materialized. If the user is not happy with the inferred highlighting, the user provides additional examples and the synthesis process is repeated.

The procedure SynthesizeFieldExtractionProg, described in Algorithm 5, enables example-based interaction. It takes as input a document D, a schema M, a highlighting CR of the document that is consistent with M, a non-materialized field f, a set of positive instances $\tilde{R}_1$, and a set of negative instances $\tilde{R}_2$ (which is empty in case field f has been simplified). SynthesizeFieldExtractionProg returns a program P such that (a) P is consistent with the examples and (b) the updated highlighting that results from executing P is consistent with the schema M. Line 3 finds a suitable ancestor f' from CR that forms the learning boundary for f. The loops at lines 7 and 12 group the input examples into boundaries imposed by f'-regions. Depending on the relationship between f and f', FlashExtract invokes an appropriate API provided by our inductive synthesis framework. In particular, it invokes SynthesizeSeqRegionProg (line 9) to learn a program for extracting a sequence of f-regions in an f'-region (if f' is a sequence-ancestor of f), or SynthesizeRegionProg (line 14) to learn a program for extracting a single f-region in an f'-region (if f' is a structure-ancestor of f). Both SynthesizeSeqRegionProg and SynthesizeRegionProg actually return a sequence of programs of the right type. The loop at line 15 selects the first program P in this sequence (if it exists) that ensures that the updated highlighting that results from executing P is consistent with the schema M.

An interesting aspect of the above-mentioned interaction is the order in which the user iterates over various fields. FlashExtract is flexible enough to let users extract various fields in any iteration order. This is especially useful when the user dynamically decides to update the data extraction schema (e.g., extract more fields). Iterating over fields in a bottom-up ordering offers an interesting advantage. It allows FlashExtract to

guess the organization of leaf field instances by looking at their relative order (thereby obviating the need to provide examples for any non-leaf field.) While this is successful in most cases, it may not be able to deal with cases where field instances may be null. On the other hand, iterating over fields in a top-down topological order requires the user to provide examples for each field (including non-leaf fields), but it offers three advantages: (a) it provides an easy visualization for the user to inspect the results of the organization of various non-leaf field instances, (b) it provides greater chance of success since the synthesis task for a field can now be enabled relative to any ancestor field as opposed to the entire document, (c) it may also entail having to provide fewer examples for any field that is nested inside another field whose instances have all been identified. Hence, if the leaf field instances are never null and the user does not need help with identifying representative examples, the user may simply provide few examples for each leaf field and `FlashExtract` may be able to automatically infer the organization of the various leaf field instances. Otherwise, we recommend that the user iterates over fields in a top-down topological order.

3.3 Inductive Synthesis Framework

In this section, we describe a general framework for developing the inductive synthesis APIs (namely, `SynthesizeSeqRegionProg` and `SynthesizeRegionProg`) that enable the example-based user interaction model discussed in the previous section. We build this framework over the inductive synthesis methodology proposed in [Gulwani et al., 2012] of designing appropriate DSLs and developing algorithms to synthesize programs in those DSLs from examples. However, we go one step further. We identify an algebra of core operators that can be used to build various data extraction DSLs for various document types (Section 3.3.2). We also present modular synthesis algorithms for each of these operators in terms of the synthesis algorithms for its (non-atomic) arguments (Section 3.3.3)—this enables automatic generation of synthesis algorithms for any DSL that is constructed using our algebra of core operators. We start out by formalizing the notion of a data extraction DSL.

3.3.1 Data Extraction DSLs

A data extraction DSL is represented by a tuple (G, N_1, N_2). G is a grammar that defines data extraction strategies. It contains definitions for various non-terminals N. Each non-terminal N is defined as a ranked collection of rules (also referred to as N.`RHSs`) of the same type. The type of a non-terminal is the type of its rules. A rule consists of a fixed expression or an operator applied to other non-terminals of appropriate types or fixed expressions. The type of a rule is the return type of the fixed expression or the operator that constitutes the rule.

We say a non-terminal is *synthesizable* if each of its rules either (a) involves an operator from our core algebra applied to fixed expressions or synthesizable non-terminals, or (b) involves an operator that is equipped with an inductive synthesis algorithm of its own (*i.e.*, domain-specific operators), or (c) fixed expressions. N_1 is a distinguished (top-level) synthesizable non-terminal of type sequence of regions. N_2 is another distinguished (top-level) synthesizable non-terminal of type region.

An expression generated by a non-terminal of type T can be viewed as a program with return type T. Note that the expressions generated by N_1 represent SeqRegion programs and expressions generated by N_2 represent Region programs. The DSL expressions may involve one distinguished free variable R_0 (of type Region) that denotes the input to the top-level SeqRegion or Region programs. Any other free variable that occurs in a DSL expression must be bound to some lambda definition that occurs in a higher level expression.

A state σ of a program P is an assignment to all free variables in P. We use the notation $\{x \leftarrow v\}$ to create a state that maps variable x to value v. We use the notation $\sigma[v/x]$ to denote setting the value of variable x to value v in an existing state σ. We use the notation $[\![P]\!]\sigma$ to denote the result of executing the program P in state σ.

Next, we discuss the core operators in our algebra that can be used to build data extraction DSLs.

3.3.2 Core Algebra for Constructing Data Extraction DSLs

Our core algebra is based around certain forms of map, filter, merge, and pair operators. The pair operator (which returns a scalar) constitutes a *scalar expression*, while the other operators (which return a sequence) constitute a *sequence expression*.

Decomposable Map Operator A `Map` operator has two arguments $\lambda x : F$ and S, where S is a sequence expression of type $\texttt{List}\langle T\rangle$ and F is some expression of type T' and uses an additional free variable x. The return type of `Map` is $\texttt{List}\langle T'\rangle$. $\texttt{Map}(\lambda x : F, S)$ has the standard semantics, wherein it applies function F to each element of the sequence produced by S to construct the resultant sequence.

$$[\![\texttt{Map}(\lambda x : F, S)]\!]\sigma \;=\; [t_0, \ldots, t_n], \text{ where } n = |Y' - 1|, t_i = [\![F]\!](\sigma[Y'[i]/x]), Y' = [\![S]\!]\sigma$$

We say that a `Map` operator is *decomposable* (*w.r.t.* the underlying DSL, which defines the language for F and S) if it has the following property: For any input state σ and a sequence Y, there exists a sequence Z such that

$$\forall F, S : Y \sqsubseteq [\![\texttt{Map}(F, S)]\!]\sigma \implies Z \sqsubseteq [\![S]\!]\sigma \wedge [\![\texttt{Map}(F, Z)]\!]\sigma = Y$$

where $\sqsubseteq$ denotes the subsequence relationship. Let `Decompose` be a function that computes such a witness Z, given σ and Y. It has the following signature:

$$\texttt{Map.Decompose} : (Region \times \texttt{List}\langle T'\rangle) \to \texttt{List}\langle T\rangle$$

The `Decompose` function facilitates the reduction of examples for `Map` operator to examples for its arguments F and S, thereby reducing the task of learning the desired `Map` expression from examples to the sub-tasks of learning F and S expressions from respective examples.

Filter Operators Our algebra allows two kinds of filter operators over sequences, one that selects elements based on their properties (`FilterBool`), and the other one that selects elements based on their indexes (`FilterInt`).

A `FilterBool` operator has two arguments $\lambda x : B$ and S, where S is a sequence expression of type $\texttt{List}\langle T\rangle$ and B is a Boolean expression and uses an additional free

variable x. The return type of `FilterBool` is `List`$\langle T \rangle$. `FilterBool`$(\lambda x : F, S)$ has the standard filtering semantics: it selects those elements from S that satisfy B.

For example, let S be the set of all lines in the file in Example 6. The expression `FilterBool`$(\lambda x :$ `EndsWith`([Number, Quote]$, x), S)$ selects all yellow lines. The predicate `EndsWith`([Number, Quote]$, x)$ returns true iff the string x ends with a number followed by a double quote.

A `FilterInt` operator has three arguments: a non-negative integer `init`, a positive integer `iter`, and a sequence expression S. Its return value also has the same type as that of S. The `FilterInt` operator takes every `iter` elements from S starting from `init` as the first element. Its semantics is as follows:

$$[\![\texttt{FilterInt}(\texttt{init}, \texttt{iter}, S)]\!]\sigma \;=\; \textbf{let } L = [\![S]\!]\sigma \textbf{ in } \texttt{Filter}\Big(\lambda x : (\texttt{indexof}(L, x) - \texttt{init})\%\texttt{iter} = 0, L\Big)$$

For example, `FilterInt`$(1, 2, S)$ selects all elements at odd indices from a sequence.

The two kinds of filter operators can be composed to enable sophisticated filtering operations.

Merge Operator A `Merge` operator takes as input a set of n sequence expressions, each of which is generated by the same non-terminal A (of some type of the form `List`$\langle T \rangle$). The return value of `Merge` also has the same type as that of A. The `Merge` operator combines the results of these n expressions together—this is useful when a single expression cannot extract all intended regions. This operator is a disjunctive abstraction and allows extraction of multiple-format field instances by merging several single-format field instances. Its semantics is as follows:

$$[\![\texttt{Merge}(A_1, \ldots, A_n)]\!]\sigma \;=\; \texttt{MergeSeq}([\![A_1]\!]\sigma, \ldots, [\![A_n]\!]\sigma)$$

The `MergeSeq` operation merges its argument sequences with respect to the order of their elements' locations in the original file.

Pair Operator A `Pair` operator has two arguments A and B and has the following standard pair operator semantics.

$$[\![\texttt{Pair}(A, B)]\!]\sigma \;=\; ([\![A]\!]\sigma, [\![B]\!]\sigma)$$

The pair operator allows constructing region representations from smaller elements. For example, we can create a text region from a pair of its start and end positions.

3.3.3 Modular Synthesis Algorithms

The API `SynthesizeSeqRegionProg` takes as input a set of examples, each of which consists of a triple $(R, \tilde{R_1}, \tilde{R_2})$, where R denotes the input region, $\tilde{R_1}$ denotes positive instances of the regions that should be highlighted within R, and $\tilde{R_2}$ denotes negative instances of the regions that should not be highlighted within R. The API `SynthesizeRegionProg` takes as input a set of examples, each of which consists of a pair (R, R'), where R denotes the input region and R' denotes the output region. Both these APIs return an ordered set of programs in the DSL, each of which is consistent with the provided examples.

Algorithm 6 contains the pseudo-code for these APIs. In this algorithm, the method `SynthesizeSeqRegionProg` first learns from only positive instances by invoking the learn method of the top-level sequence non-terminal N_1 (line 3) and then selects those programs that additionally satisfy the negative instances constraint (loop at line 5). The operator :: appends an element to a list. The method `SynthesizeRegionProg` simply invokes the learn method of the top-level region non-terminal N_2 (line 11).

The learn method for any non-terminal N invokes the learn methods associated with its various rules (line 15) and returns the ordered union of the sequences of the programs synthesized from each. The operator $+\!\!+$ performs list concatenation.

Algorithm 7, 8, and 9 describe the learn methods for the operators that constitute the various rules. The higher level idea is to define them in terms of the learn methods of their arguments. This allows for a free synthesis algorithm for any data extraction DSL.

Learning Decomposible Map Operator The key idea here is to first find the witness Z_j for each example (σ_j, Y_j) using the operator's `Decompose` method (line 3). This allows us to split the problem of learning a map expression into two independent simpler sub-problems, namely learning of a scalar expression F(line 5), and learning of a sequence expression S (line 7) from appropriate examples. The returned result is an appropriate cross-product style composition of the results returned from the sub-problems (line 11).

Algorithm 6: Top level learning APIs in FlashExtract.

```
1  function SynthesizeSeqRegionProg(
   Set⟨(Region, List⟨Region⟩, List⟨Region⟩)⟩ Q) : List⟨Prog⟩ is
2      Q' := {({R0 ← R}, R̃1) | (R, R̃1, R̃2) ∈ Q}
       /* learn with positive examples */
3      P̃ := N1.Learn(Q')                          /* start symbol for sequence */
4      P̃' := []
5      foreach P ∈ P̃ do                  /* filter programs violating negatives */
6          if (∃(R, R̃1, R̃2) ∈ Q : ([[P]]{R0 ← R}) ∩ R̃2 ≠ ∅) then continue
7          P̃' := P̃' :: P
8      return P̃'
9  function SynthesizeRegionProg(Set⟨(Region, Region)⟩ Q) : List⟨Prog⟩ is
10     Q' := {({R0 ← R}, R') | (R, R') ∈ Q}
11     return N2.Learn(Q')                        /* start symbol for region */
12 function N.Learn(Set⟨(State, T)⟩ Q) : List⟨Prog⟩ is
13     P̃ := []
14     foreach C ∈ N.RHSs do
15         P̃ := P̃ ++ C.Learn(Q)
16     return P̃
```

Learning Merge Operator The key idea here is to consider all (minimal) partitioning of the examples such that the learn method of the argument for each partition returns a non-empty result. The set T (at line 16) holds all such minimal partitions. For each such partition (loop at line 18), we invoke the learn method of the argument for each class in the partition (line 20), and then appropriately combine the results. Although this algorithm is exponential in the size of the input set, it works efficiently in practice because the number of examples required for learning is very small in practice.

Learning Filter Operators The key idea in the learn method for `FilterBool` is to independently learn an ordered set $\tilde{P}_1$ of sequence expressions (line 24) and an ordered

Algorithm 7: Algorithm to learn `Map` and `Merge` operators.

function `Map.Learn` ($\texttt{Set}\langle(State, \texttt{List}\langle T\rangle)\rangle\ Q) : \texttt{List}\langle Prog\rangle$ **is**

  Let Q be $\{(\sigma_j, Y_j)\}_{1\le j\le m}$

  for $j := 1..m$ **do** $Z_j := \texttt{Map.Decompose}(\sigma_j, Y_j)$ `/* find witnesses` Z `*/`

  $Q_1 := \{(\sigma_j[Z_j[i]/x], Y_j[i]) \mid 0 \le i < len(Z_j), 1 \le j \le m\}$

  $\tilde{P}_1 := F.\texttt{Learn}(Q_1)$ `/* learn Map's function` F `*/`

  $Q_2 := \{(\sigma_j, Z_j) \mid 1 \le j \le m\}$

  $\tilde{P}_2 := S.\texttt{Learn}(Q_2)$ `/* learn Map's sequence` S `*/`

  $\tilde{P} := []$

  foreach $P_1 \in \tilde{P}_1$ **do**

    foreach $P_2 \in \tilde{P}_2$ **do** $\tilde{P} := \tilde{P}$ :: "$\texttt{Map}(P_1, P_2)$"

  return $\texttt{CleanUp}(\tilde{P}, Q)$

function `Merge.Learn` ($\texttt{Set}\langle(State, \texttt{List}\langle T\rangle)\rangle Q) : \texttt{List}\langle Prog\rangle$ **is**

  `/* Learn Merge by splitting the examples into disjoint partitions, and learn a program for each example partition.` A `is Merge's arguments. */`

  Let Q be $\{(\sigma_j, Y_j)\}_{1\le j\le m}$

  `/* Split examples` Y_j `into all possible subsets s.t. we can learn at least a program from each subset.` X `holds all possible subsets of examples. */`

  $X := \{Q' \mid Q' = \{(\sigma_j, Y'_j)\}_{1\le j\le m}, \forall 1 \le j \le m : Y'_j \sqsubseteq Y_j, A.\texttt{Learn}(Q') \neq []\}$

  $Y := \bigcup_{(\sigma_j, Y_j)\in Q} Y_j$ `/* all positive examples */`

  `/* Find some elements in X so that together they form the examples` Y_j `. */`

  $T := \{X' \mid X'$ is a minimal subset of X s.t. $\{Y'_j \mid (\sigma_j, Y'_j) \in Q', Q' \in X'\} = Y\}$

  $\tilde{P} := []$

  foreach $X' \in T$ *ordered by size* **do**

    Let $Q'_1, ..., Q'_n$ be the various elements of X'

    $\tilde{P}_1, \ldots, \tilde{P}_n := A.\texttt{Learn}(Q'_1), \ldots, A.\texttt{Learn}(Q'_n)$

    $\tilde{P} := \tilde{P} +\!\!+ \{$"$\texttt{Merge}(P_1, ..., P_n)$" $\mid \forall 1 \le i \le n : P_i \in \tilde{P}_i\}$

  return $\texttt{CleanUp}(\tilde{P}, Q)$

Algorithm 8: Algorithm to learn `FilterInt` and `FilterBool` operators.

23 **function** `FilterBool.Learn`($\texttt{Set}\langle(State, \texttt{List}\langle T\rangle)\rangle\ Q) : \texttt{List}\langle Prog\rangle$ **is**

24 $\quad \tilde{P}_1 := S.\texttt{Learn}(Q)$ `/* learn FilterBool's sequence` S `*/`

25 $\quad Q' := \{(\sigma[Y[i]/x], \texttt{True}) \mid (\sigma, Y) \in Q, 0 \leq i < len(Y)\}$

26 $\quad \tilde{P}_2 := B.\texttt{Learn}(Q')$ `/* learn FilterBool's predicate` B `*/`

27 $\quad \tilde{P} := []$

28 $\quad$ **foreach** $P_1 \in \tilde{P}_1$ **do**

29 $\quad\quad$ **foreach** $P_2 \in \tilde{P}_2$ **do** $\tilde{P} := \tilde{P} :: \text{“}\texttt{FilterBool}(P_1, P_2)\text{”}$

30 $\quad$ **return** $\texttt{CleanUp}(\tilde{P}, Q)$

31 **function** `FilterInt.Learn`($\texttt{Set}\langle(State, \texttt{List}\langle T\rangle)\rangle\ Q) : \texttt{List}\langle Prog\rangle$ **is**

32 $\quad$ Let Q be $\{(\sigma_j, Y_j)\}_{1 \leq j \leq m}$

33 $\quad \tilde{P}_1 := S.\texttt{Learn}(Q)$ `/* learn FilterInt's sequence` S `*/`

34 $\quad \tilde{P} := []$

35 $\quad$ **foreach** $P_1 \in \tilde{P}_1$ **do**

36 $\quad\quad \texttt{init} := \infty, \texttt{iter} := 0$

37 $\quad\quad$ **for** $j := 1 \ldots m$ **do**

38 $\quad\quad\quad Z_j := [\![P_1]\!]\sigma_j$

39 $\quad\quad\quad \texttt{init} := \texttt{Min}(\texttt{init}, \texttt{indexof}(Z_j, Y_j[0]))$

40 $\quad\quad\quad$ **for** $i := 0 \ldots |Y_j| - 2$ **do**

41 $\quad\quad\quad\quad t := \texttt{indexof}(Z_j, Y_j[i+1]) - \texttt{indexof}(Z_j, Y_j[i])$

42 $\quad\quad\quad\quad$ **if** $\texttt{iter} = 0$ **then** $\texttt{iter} := t$

43 $\quad\quad\quad\quad$ **else** $\texttt{iter} := \texttt{GCD}(\texttt{iter}, t)$

44 $\quad\quad$ **if** $\texttt{iter} = 0$ **then** $\texttt{iter} := 1$

45 $\quad\quad \tilde{P} := \tilde{P} :: \text{“}\texttt{FilterInt}(\texttt{init}, \texttt{iter}, P_1)\text{”}$

46 $\quad$ **return** $\texttt{CleanUp}(\tilde{P}, Q)$

set $\tilde{P}_2$ of Boolean expressions (line 26) that are consistent with the given examples. The returned result is a cross-product style composition of the sub-results $\tilde{P}_1$ and $\tilde{P}_2$.

The key idea in the learn method for `FilterInt` is to first learn an ordered set $\tilde{P}$ of

Algorithm 9: Algorithm to learn `Pair` operator.

47 **function** `Pair.Learn`($\mathit{Set}\langle(State, (T_1, T_2))\rangle\ Q$) : $\mathit{List}\langle Prog\rangle$ **is**
48 Let Q be $\{(\sigma_j, (u_j, u'_j))\}_{1 \le j \le m}$
49 $Q_1 := \{(\sigma_j, u_j)\}_{1 \le j \le m}$; $Q_2 := \{(\sigma_j, u'_j)\}_{1 \le j \le m}$
50 $\tilde{P_1} := A$.`Learn`(Q_1)
51 $\tilde{P_2} := B$.`Learn`(Q_2)
52 **if** $\tilde{P_1} = \emptyset$ **or** $\tilde{P_2} = \emptyset$ **then return** []
53 $\tilde{P} := []$
54 **foreach** $P_1 \in \tilde{P_1}$ **do**
55 **foreach** $P_2 \in \tilde{P_2}$ **do**
56 $\tilde{P} := \tilde{P} ::$ "`Pair`(P_1, P_2)"
57 **return** $\tilde{P}$

sequence expressions (line 33) that are consistent with the given examples. Then, for each such program, we learn the most strict filtering logic that filters as few elements as possible while staying consistent with the examples. In particular, we select `init` to be the minimum offset (across all examples) of the first element in Y_j in the sequence Z_j returned by executing the sequence program in the example state σ_j (line 39). We select `iter` to be the GCD of all distances between the indices of any two contiguous elements of Y_j in Z_j (line 43).

Learning Pair Operator The key idea here is to invoke the learn method of the first (second) argument at line 50 (line 51) to learn programs that can compute the first (second) element in the various output pairs in the examples from the respective inputs. The final result is produced by taking a cross-product of the two sets of programs that are learned independently (loop at line 54).

CleanUp Optimization An important performance and ranking optimization employed by the learn methods of various operators is use of the `CleanUp` method, which removes those programs that extract more regions than some retained program (Algorithm 10). More precisely, this method removes each of those (lower-ranked) programs from an

Algorithm 10: Algorithm to perform cleanup optimization.

```
1 function CleanUp(List⟨Prog⟩ P̃, Set⟨(State,List⟨T⟩)⟩ Q) : List⟨Prog⟩ is
2     P̃′ := []
3     foreach i = 1 to |P̃| do
4         P := P[i]
5         incl := true
6         foreach k = 1 to |P̃| do
7             if (P̃[k] subsumes P w.r.t. Q) and ((P does not subsume P̃[k] w.r.t. Q) or k < i)
              then incl := false
8         if (incl = true) then P̃′ := P̃′ :: P
9     return P̃′
```

ordered set of programs that is *subsumed* by some unremoved program (See Definition 8). Note that this does not affect the completeness property associated with the various learning methods (Theorem 3). Furthermore, it implements an important ranking criterion that assigns higher likelihood to the scenario wherein the user provides consecutive examples from the beginning of any immediately enclosing ancestral region (as opposed to providing arbitrary examples).

3.3.4 Correctness

We now describe the correctness properties associated with our two key synthesis APIs: `SynthesizeSeqRegionProg` and `SynthesizeRegionProg`. First, we state some useful definitions.

Definition 7 *(Consistency) A scalar program P (i.e., a program that returns a scalar value) is said to be* consistent *with a set $Q = \{(\sigma_j, u_j)\}_j$ of scalar examples if $\forall j : u_j = [\![P]\!]\sigma_j$. A sequence program P (i.e., a program that returns a sequence) is said to be* consistent *with a set $Q = \{(\sigma_j, Y_j)\}_j$ of sequence examples with positive instances if $\forall j : Y_j \subseteq [\![P]\!]\sigma_j$. A sequence program P is said to be* consistent *with a set $Q = \{(\sigma_j, Y_j, Y'_j)\}_j$ of sequence examples with positive and negative instances if $\forall j : (Y_j \subseteq [\![P]\!]\sigma_j \wedge Y'_j \cap [\![P]\!]\sigma_j = \emptyset)$.*

Definition 8 *(Subsumption) Given a set $Q = \{(\sigma_j, Y_j)\}_j$ of sequence examples with positive instances, and two sequence programs P_1, P_2 that are consistent with Q, we say that P_1* subsumes *P_2 w.r.t. Q if $\forall j : [\![P_1]\!]\sigma_j \subseteq [\![P_2]\!]\sigma_j$.*

The following two theorems hold.

Theorem 1 (Soundness) *The programs P returned by `SynthesizeSeqRegionProg` and `SynthesizeRegionProg` are consistent with the input set of examples.*

The proof of Theorem 1 follows easily by induction (on the structure of the DSL) from similar soundness property of the learn methods associated with the non-terminals and core algebra operators.

Theorem 2 (Completeness) *If there exists some program that is consistent with the input set of examples, `SynthesizeSeqRegionProg` (and `SynthesizeRegionProg`) produce one such program.*

The proof of Theorem 2 follows from two key observations: (a) The learn methods associated with the scalar non-terminals and the (scalar) Pair operator satisfy a similar completeness property. (b) The learn methods associated with the sequence non-terminals and the sequence operators of the core algebra satisfy a stronger completeness theorem stated below (Theorem 3).

Theorem 3 (Strong Completeness) *The learn methods associated with the sequence non-terminals and the sequence operators of the core algebra (namely Map, FilterBool, FilterInt, and Merge) satisfy the following property: "For every sequence program P that is consistent with the input set of examples, there exists a program P' in the learned set of programs that subsumes P."*

The proof of Theorem 3 follows from induction on the DSL's structure. Note that the `CleanUp` optimization only removes those (lower-ranked) programs that are subsumed by other programs.

3.4 Instantiations

We now present instantiations of our framework to three data extraction domains: text files, webpages, and spreadsheets. For each domain, we define the notion of a region, describe the domain's underlying DSL, and discuss the implementation of domain-specific learn methods.

3.4.1 Text Instantiation

A region in this domain is a pair of character positions in the input text file.

Language $\mathcal{L}_{\text{text}}$ Figure 3.6 shows the syntax of $\mathcal{L}_{\text{text}}$, our data extraction DSL for this domain. The core algebra operators are in **bold**. We name the various `Map` operators differently in order to associate different `Decompose` methods with them. The non-terminal N_1 is a `Merge` operator over constituent sequence expressions SS. The non-terminal N_2 is defined as a `Pair` operator over two position expressions.

The position expression `Pos`(x, p) evaluates to a position in string x that is determined by the attribute p (inspired by a similar concept introduced in [Gulwani, 2011]). The attribute p is either an absolute position k, or is the k^{th} element of the position sequence identified by the regex pair rr which consists of two regexes r_1 and r_2. The selection order is from left-to-right if k is positive, or right-to-left if k is negative. The position sequence identified by (r_1, r_2) in string x, also referred to as `PosSeq`(x, rr), is the set of all positions k in x such that (some suffix of the substring on) the left side of k matches with r_1 and (some prefix of the substring on) the right side of k matches with r_2. A regex r is simply a concatenation of (at most 3) tokens. A token T is a pre-defined standard character class such as alphabets, digits, colon character, or a special data type such as date time and IP address. Our instantiation contains 30 of such tokens. We also define some context-sensitive tokens dynamically based on frequently occurring string literals in the neighborhood of examples highlighted by the user. For instance, in Example 6, our dynamically learned tokens include the string "DLZ - Summary Report" (which is useful for learning the green outer structure boundary) and the string ""Sample ID:"" (which is useful to extract the orange sample ID).

$$
\begin{aligned}
\text{Disjunct Pair Seq } \mathbf{N_1} &::= \textbf{Merge}(SS_1, \ldots, SS_n) \\
\text{Pos Pair Region } \mathbf{N_2} &::= \textbf{Pair}(\texttt{Pos}(R_0, p_1), \texttt{Pos}(R_0, p_2)) \\
\text{Pair Seq } SS &::= \texttt{Lines}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(x, p_1), \texttt{Pos}(x, p_2)), LS) \\
&\quad \mid \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Pos}(R_0[x:], p)), PS) \\
&\quad \mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(R_0[:x], p), x), PS) \\
\text{Line Seq } LS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, BLS) \\
\text{Bool Line Seq } BLS &::= \textbf{FilterBool}(b, \texttt{split}(R_0, \text{`\textbackslash n'})) \\
\text{Position Seq } PS &::= \texttt{Lines}\textbf{Map}(\lambda x : \texttt{Pos}(x, p), LS) \\
&\quad \mid \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{PosSeq}(R_0, rr)) \\
\text{Predicate } b &::= \lambda x : \texttt{True} \\
&\quad \mid \lambda x : \texttt{\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{Contains}(r, k, x) \\
&\quad \mid \lambda x : \texttt{Pred\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{PredContains}(r, k, x) \\
&\quad \mid \lambda x : \texttt{Succ\{Starts,Ends\}With}(r, x) \mid \lambda x : \texttt{SuccContains}(r, k, x) \\
\text{Position Attribute } p &::= \texttt{AbsPos}(k) \mid \texttt{RegPos}(rr, k) \\
\text{Regex Pair } rr &::= (r_1, r_2) \\
\text{Regex } r &::= T\{0, 3\} \\
\text{Token } T &::= C+ \mid \texttt{DynamicToken}
\end{aligned}
$$

Figure 3.6: The syntax of $\mathcal{L}_{\text{text}}$, the DSL for extracting text files.

The first rule of SS consists of a `Map` operator `Lines`**Map** that maps each line of a line sequence LS to a pair of positions within that line. The `Decompose` method for `Lines`**Map** takes as input a region R and a sequence of position pairs and returns the sequence of lines from R that contain the corresponding position pairs.

The second (third) rule of SS pairs each position x in a position sequence PS with a position that occurs somewhere on its right (left) side. The notation $R_0[x:]$ ($R_0[:x]$) denotes the suffix (prefix) of the text value represented by R_0 starting (ending) at position x. The `Decompose` method associated with `StartSeq`**Map** (`EndSeq`**Map**) takes as

input a region R and a sequence of positions and maps each position k in the input sequence to the string $R[k:]$ ($R[:k]$).

The line sequence non-terminal LS uses a nested combination of `FilterInt` and `FilterBool`. The various choices for predicate b (used in `FilterBool`) have the expected semantics. For example, `StartsWith`(r, x) asserts if line x starts with regex r, while `Contains`(r, k, x) asserts if line x contains k occurrences of regex r. We also take hints from preceding and succeeding lines via `Pred`$*$ and `Succ`$*$ predicates. For example, `PredStartsWith`(r, x) asserts that the line that precedes line x ends with regex r.

The position sequence non-terminal PS includes expressions that select a position within each line of a line sequence (using the `Lines`**Map** operator) or that filter positions returned by the `PosSeq` operator (using the **FilterInt** operator).

Example 9 *Below is a program in $\mathcal{L}_{\text{text}}$ for extracting the yellow regions in Example 6 (from the top-level region of the entire file).*

Lines$\mathbf{Map}(\lambda x : \mathbf{Pair}($*Pos*$(x, p_1),$*Pos*$(x, p_2)), LS)$*, where*

$p_1 =$ *AbsPos*$(0), p_2 =$ *AbsPos*(-1),

$LS = \mathbf{FilterInt}(0, 1, \mathbf{FilterBool}(\lambda x :$ *EndsWith*$($*[Number, Quote]*$, x), $*split*$(R_0, \text{'\textbackslash n'})))$

The `FilterBool` operator takes all the lines in the document and selects only those that end with a number and a quote. The `FilterInt` operator does not do any filtering (`init` = 0, `iter` = 1); it simply passes the result of `FilterBool` to LS. The map function in `Lines`**Map** returns the entire input line (`AbsPos` (0) denotes the beginning of the line, while `AbsPos` (-1) denotes the end of the line). The `Lines`**Map** operator thus returns a sequence identical to LS, which is the yellow sequence.

Example 10 *Below is a program for extracting the magenta regions in Example 6 (from the top-level region of the entire file).*

EndSeq$\mathbf{Map}(\lambda x : \mathbf{Pair}($*Pos*$(R_0[:x], p), x), PS)$*, where*

$p =$ *RegPos*$\Big(([$*DynamicTok*$(,\text{""})], \epsilon), -1\Big), PS = \mathbf{FilterInt}(0, 1,$ *PosSeq*$(R_0, (r_1, r_2))$*, and*

$r_1 = [$*DynamicTok*$(,\text{""}),$ *Word*$], r_2 = [$*DynamicTok*$(\text{""},),$ *Number*$,$ *Comma*$]$

FlashExtract recognizes the prefixes (,"") and suffixes ("",) of the given examples as frequently occurring substrings and promotes them to dynamic tokens. The PosSeq operator returns the sequence of all end positions of the magenta sequence (since each of these have an r_1 match on the left and an r_2 match on the right). Note that there are other positions that either have an r_1 match on the left (such as the position before the number in "Sample ID:;""5007-01"""), or have an r_2 match on the right (such as the position after the character L in ""ug/L"",0.0009), but not both; hence, these positions are not selected by the PosSeq operator. Since FilterInt does not filter any elements, PS is the same sequence returned by the regex pair. The map function in EndSeq**Map** takes each end position in PS and finds its corresponding start position specified by p, which is the first position from the right (k = -1) that matches the dynamic token (,"") on the left side. The result is the magenta sequence.

Example 11 *If the magenta field is wrapped within the yellow structure, one of its extraction programs is as follows:*

Pair(Pos(R_0, p_1),Pos(R_0, p_2))*, where*

$p_1 = \langle [$DynamicTok(,"")$]$, $\epsilon, 1\rangle$, $\quad p_2 = \langle \epsilon, [$DynamicTok(""$,)]$, $1\rangle$

Since the yellow field is the structure-ancestor of the magenta field, FlashExtract learns a Pair operator to extract a magenta region within a yellow region. The start position of this pair is the first position from the left (k = 1) that matches (,"") on the left side (r_1), and the end position is the first position from the left that matches (""*,*) on the right side (r_2). This program is simpler than the one in Example 10, because it exploits the separation determined by the program for the yellow field.

Domain-Specific Learn Methods The learning of Boolean expression b is performed using brute-force search. The learning of position attribute expressions p is performed using the technique described in prior work [Gulwani, 2011].

3.4.2 Webpage Instantiation

A region in this domain is either an HTML node, or a pair of character positions within the text property of an HTML node.

$$
\begin{aligned}
\text{Disjuctive Seq } \mathbf{N_1} &::= \textbf{Merge}(NS_1, \ldots, NS_n) \\
&\mid \textbf{Merge}(SS_1, \ldots, SS_n) \\
\text{Region } \mathbf{N_2} &::= \texttt{XPath} \mid \textbf{Pair}(\texttt{Pos}(R_0, p_1), \texttt{Pos}(R_0, p_2)) \\
\text{Node Seq } NS &::= \texttt{XPaths} \\
\text{Pos Pair Seq } SS &::= \texttt{SeqPair}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(x.\texttt{Val}, p_1), \texttt{Pos}(x.\texttt{Val}, p_2)), ES) \\
&\mid \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Pos}(R_0[x:], p)), PS) \\
&\mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Pos}(R_0[:x], p), x), PS) \\
\text{Element Seq } ES &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{XPaths}) \\
\text{Position Seq } PS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, \texttt{PosSeq}(R_0, rr))
\end{aligned}
$$

Figure 3.7: The syntax of $\mathcal{L}_{\text{web}}$, the DSL for extracting webpages. Definitions of p and rr are similar to those in Figure 3.6.

Language $\mathcal{L}_{\text{web}}$ Figure 3.7 shows the syntax of the DSL $\mathcal{L}_{\text{web}}$ for extracting data from webpages. `XPath` (`XPaths`) denote an XPath expression that returns a single HTML node (a sequence of HTML nodes) within the input HTML node. Position attribute p and regex pair rr are similar to those in the text instantiation DSL $\mathcal{L}_{\text{text}}$. Our token set additionally includes dynamic tokens that we create for various HTML tags seen in the input webpage.

The non-terminal N_1 represents expressions that compute a sequence of HTML nodes or a sequence of position pairs within HTML nodes. The non-terminal N_2 represents expressions that compute a HTML node or a position pair within a HTML node. The design of $\mathcal{L}_{\text{web}}$ is inspired by the design of $\mathcal{L}_{\text{text}}$. HTML nodes in $\mathcal{L}_{\text{web}}$ play a similar role to that of lines in $\mathcal{L}_{\text{text}}$. We use XPath expressions to identify relevant HTML elements instead of using regular expressions to identify appropriate lines.

The non-terminal SS represents expressions that generate a sequence of position pairs by mapping each HTML node in a sequence ES to position pairs (`SeqPair`**Map** operator) or by pairing up each position in a sequence PS of positions with another position computed relative to it (`StartSeq`**Map** and `EndSeq`**Map** operators).

$$
\begin{aligned}
\text{Disjuctive Cell Pair Seq } \mathbf{N_1} &::= \textbf{Merge}(PS_1, \ldots, PS_n) \\
&\mid \textbf{Merge}(CS_1, \ldots, CS_n) \\
\text{Cell Pair Region } \mathbf{N_2} &::= \textbf{Pair}(\texttt{Cell}(R_0, c_1), \texttt{Cell}(R_0, c_2)) \\
&\mid \texttt{Cell}(R_0, c) \\
\text{Pair Seq } PS &::= \texttt{StartSeq}\textbf{Map}(\lambda x : \textbf{Pair}(x, \texttt{Cell}(R_0[x:], c)), CS) \\
&\mid \texttt{EndSeq}\textbf{Map}(\lambda x : \textbf{Pair}(\texttt{Cell}(R_0[:x], c), x), CS) \\
\text{Cell Sequence } CS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, CE) \\
&\mid \texttt{CellRow}\textbf{Map}(\lambda x : \texttt{Cell}(x, c), RS) \\
\text{Row Sequence } RS &::= \textbf{FilterInt}(\texttt{init}, \texttt{iter}, RE) \\
\text{Cell Attribute } c &::= \texttt{AbsCell}(k) \mid \texttt{RegCell}(cb, k) \\
\text{Cell Split Seq } CE &::= \textbf{FilterBool}(cb, \texttt{splitcells}(R_0)) \\
\text{Row Split Seq } RE &::= \textbf{FilterBool}(rb, \texttt{splitrows}(R_0)) \\
\text{Cell Boolean } cb &::= \lambda x : \texttt{True} \mid \lambda x : \texttt{Surround}(T\{9\}, x) \\
\text{Row Boolean } rb &::= \lambda x : \texttt{True} \mid \lambda x : \texttt{Sequence}(T+, x)
\end{aligned}
$$

Figure 3.8: The syntax of $\mathcal{L}_{\text{sps}}$, the DSL for extracting spreadsheets.

Domain-specific Learn Methods We need to define learn methods for `XPath` and `XPaths` from example HTML nodes. This is a well-defined problem in the data mining community, called wrapper induction (see Chapter 6). We implemented a learn method that generalizes example nodes to path expressions by replacing inconsistent tags at any depth with "*", and additionally incorporates common properties of example nodes. These properties include the number of children, their types, the number of attributes and their types. The result is a list of XPath expressions, ranging from the most specific to the most general.

3.4.3 Spreadsheet Instantiation

A region in this domain is a rectangular region formed by a pair of cells or a single cell.

Language $\mathcal{L}_{\text{sps}}$ Figure 3.8 shows the syntax of our DSL. The non-terminal N_1 represents expressions that extract a sequence of cell pairs or a sequence of cells from a given spreadsheet. The non-terminal N_2 represents expressions that extract a cell pair or a single cell from a given spreadsheet.

$\mathcal{L}_{\text{sps}}$ is inspired by $\mathcal{L}_{\text{text}}$ and $\mathcal{L}_{\text{web}}$. The notion of a row in $\mathcal{L}_{\text{sps}}$ is similar to that of a line in $\mathcal{L}_{\text{text}}$ and an HTML node in $\mathcal{L}_{\text{web}}$. Just as Boolean expressions in $\mathcal{L}_{\text{text}}$ help filter lines by matching their content against regular expressions, the row Boolean expression rb in $\mathcal{L}_{\text{sps}}$ selects spreadsheet rows by matching contents of consecutive cells inside a row against some token sequence. In addition, the cell Boolean expression cb selects cells by matching contents of the cell and its 8 neighboring cells against some 9 tokens.

As their name suggests, the two functions `splitcells` and `splitrows` split a spreadsheet region into a sequence of cells (obtained by scanning the region from top to down and left to right) and into a sequence of rows respectively, on which cell and row Boolean expressions can be applied.

Domain-specific Learn Methods The learning of Boolean expression cb and rb is performed using brute-force search. The learning of cell attribute expressions c is performed in a manner similar to that of position attribute expressions p in $\mathcal{L}_{\text{text}}$.

We need to define learn methods for domain-specific top-level non-terminals c, cb and rb. To learn c, we simply perform brute-force search to find all matching cell expression cb. Learning the row expression rb is similar.

3.5 Evaluation

We seek to answer the following questions related to effectiveness of FlashExtract.

- Can FlashExtract describe DSLs that are expressive enough for extraction tasks on real world files?
- How many examples are required to extract the desired data?
- How efficient is FlashExtract in learning extraction programs?

3.5.1 Setup

We implemented FlashExtract framework and its three instantiations (described in Chapter 3.4) in C#. We conducted all experiments on a machine running Windows 7 with Intel Core i7 2.67GHz, 6GB RAM.

Real-world Benchmarks We collected 75 test documents in total, 25 for each of the three domains. The text file domain is very broad. A text file might be relatively structured such as a log file, or loosely structured as in text copied and pasted from webpages or PDF documents. We selected a representative benchmark set that includes a few files for each kind. Additionally, we also included some benchmarks from the book "Pro Perl Parsing" [Frenz, 2005], which teaches how to use Perl to parse data.

For the webpage domain, we used the benchmark from [Oro et al., 2010], which describes SXPath, an extension of XPath to perform queries on Web documents. This benchmark includes 25 e-commerce popular websites with different underlying structures. For each of the website, they have two test cases corresponding to the HTML elements of the product name and the product price. In addition to these, we add a test case for the region covering all product information, and another test case for the actual price number (ignoring other texts in the price element such as "sale", "$" or "USD").

For the spreadsheet domain, we obtained 25 documents from two sources: benchmark used in previous work on spreadsheet transformation [Harris and Gulwani, 2011] (we selected those 7 documents from this benchmark that were associated with non-trivial extraction tasks), and EUSES corpus [Ii and Rothermel, 2005].

Experimental Setup For each document, we wrote down an appropriate schema describing the type of the hierarchical data inside the document, and we manually annotated all instances for the various fields in that schema to precisely define the extraction task. We used FlashExtract to learn the extraction programs for each field. Recall that we can learn the extraction logic for a field f by relating it to any of its ancestors. Among these, relating to $\perp$, is typically the hardest (in contrast, relating to one of the other ancestors can exploit the separation that has already been achieved by the extraction logic for that ancestor).

We wrote a script to simulate user interaction with FlashExtract to measure its effectiveness in the above-mentioned hardest scenario. Let $\tilde{R}$ denote all manually annotated instances for a field f. The simulator starts out with an empty set of negative instances and a singleton set of positive instances that includes the first region in $\tilde{R}$, and repeats the following process in a loop. In each iteration, the simulator invokes FlashExtract to synthesize a field extraction program from the various examples. If FlashExtract fails to synthesize a program, the simulator declares failure. Otherwise, the simulator executes the program to find any mismatch *w.r.t.* the golden result $\tilde{R}$. If no mismatch exists, the simulator declares success. Otherwise, the simulator adds the first mismatched region as a new example: it is added as a positive instance if the mismatched region occurs in $\tilde{R}$ but is not highlighted in the execution result of the previous interaction; otherwise the mismatch is added as a negative example. Furthermore, the simulator also adds all new highlighted regions that occur before the new example as positive instances.

3.5.2 Experimental Results

Expressiveness Each of the three instantiations of FlashExtract was successfully able to synthesize a desired field extraction program for the various tasks in the respective domains. Thus, FlashExtract supports data extraction DSLs that are expressive enough to describe a variety of real-world data extraction tasks.

Number of Examples FlashExtract required an average of 2.86 examples per field across all documents. Figure 3.9 shows the average number of examples (over all fields in the schema for a given document), split by positive/negative instances, for each of the 75 documents in our benchmark. We observe that users have to give more examples to extract data from text files because the structure of text files is more arbitrary. In contrast, webpages and spreadsheets are generally more structured (because of HTML tags and row/column based organization respectively), thus requiring fewer examples.

Synthesis Time Users interactively give more examples to FlashExtract in order to learn a field extraction program. We measure the synthesis time required by FlashExtract during the last iteration before FlashExtract succeeds or fails (since this last iteration

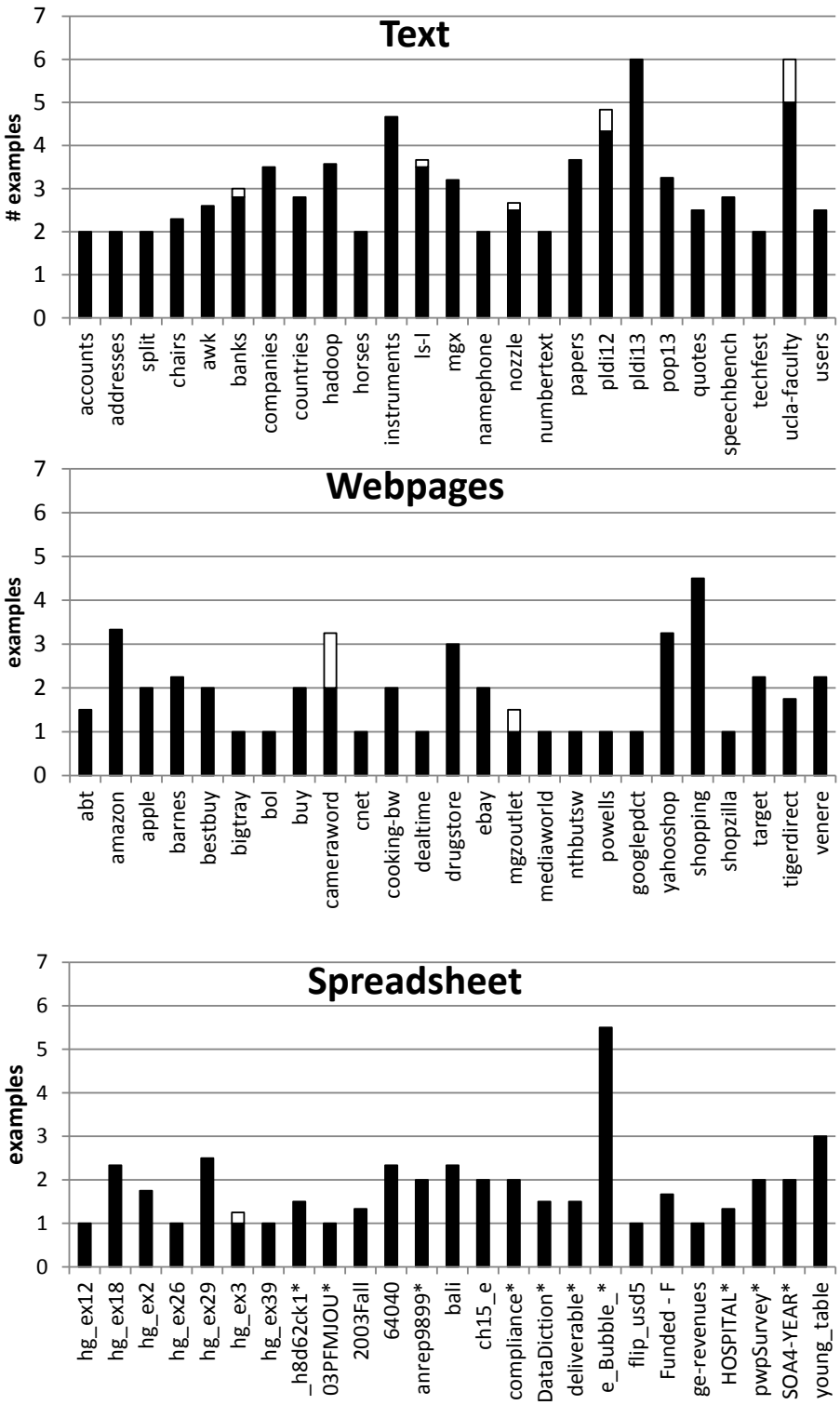


Figure 3.9: Average number of examples (solid/white bars represent positive/negative instances) across various fields for each document.

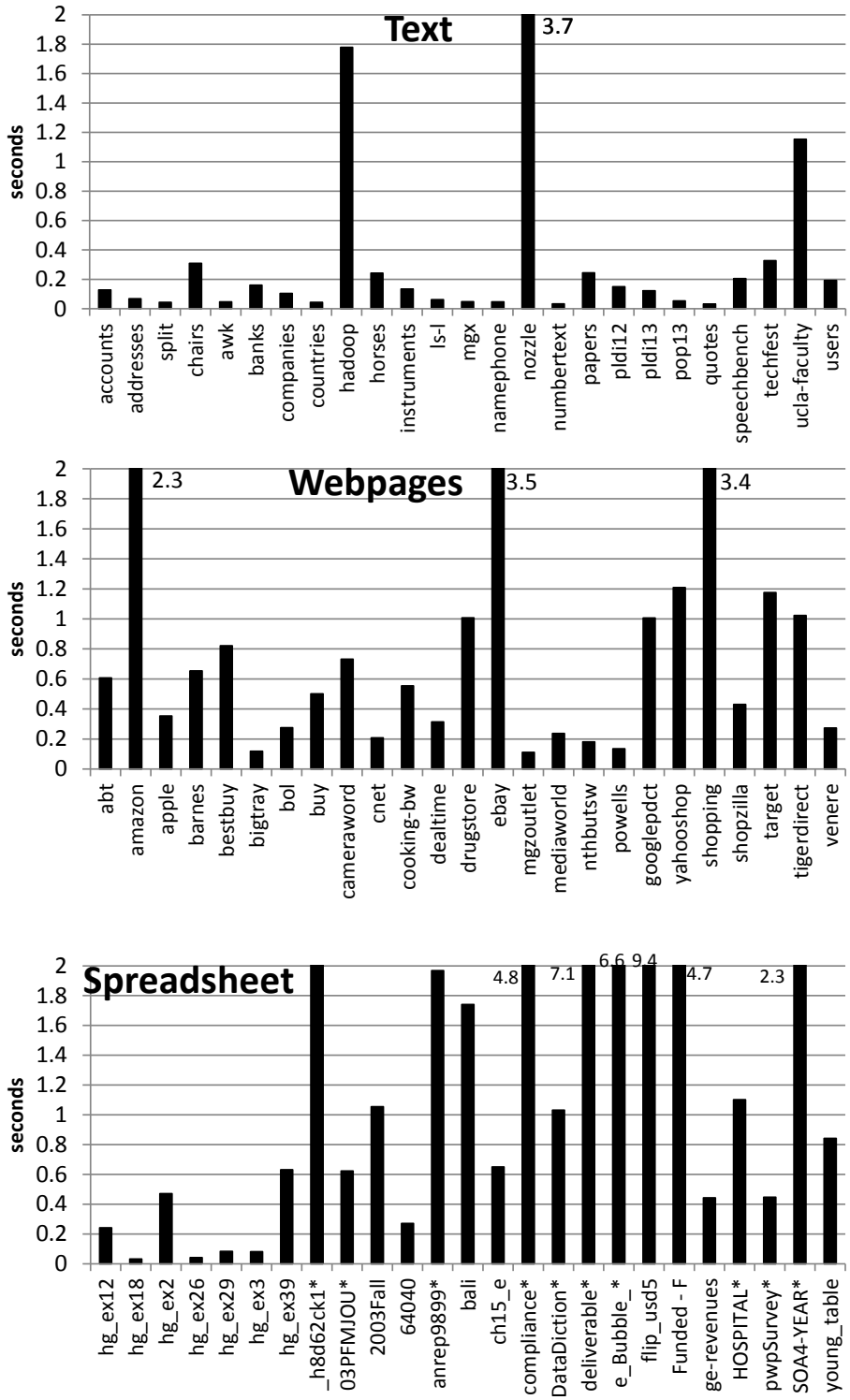


Figure 3.10: Average learning time of the last interaction across various fields for each document.

has the most number of examples, and thus typically consumes the longest synthesis time). FlashExtract required an average of 0.82 seconds per field across all documents. Figure 3.10 reports the average synthesis time (over all fields in the schema for a given document) from the last set of examples, for each of the 75 documents in our benchmark. While most text files and webpages require less than a second per field, spreadsheets sometimes take a few seconds to complete. This is because the spreadsheet DSL is richer with a larger search space.

Chapter 4

EMI: Synthesizing Compiler Test Programs from Existing Programs

Compilers are among the most important, widely-used and complex software ever written. Decades of extensive research and development have led to much increased compiler performance and reliability. Perhaps less known to application programmers is that production compilers do also contain bugs, and in fact quite a few. However, compiler bugs are hard to recognize from the much more frequent bugs in applications because often they manifest only indirectly as application failures. Thus, when compiler bugs occur, they frustrate programmers and may lead to unintended application behavior and disasters, especially in safety-critical domains. Compiler verification has been an important and fruitful area for the verification grand challenge in computing research [Hoare, 2003].

Besides traditional manual code review and testing, the main compiler validation techniques include testing against popular validation suites (such as [Plum Hall, Inc., 2015] and [ACE, 2015]), verification [Leroy, 2006], translation validation [Pnueli et al., 1998, Necula, 2000], and random testing [Yang et al., 2011]. These approaches have complementary benefits. For example, CompCert [Leroy, 2006, Leroy, 2009] is a formally verified optimizing compiler for a subset of C, targeting the embedded software domain. It is an ambitious project, but much work remains to have a fully verified production compiler that is correct end-to-end. Another good example is Csmith [Yang et al., 2011],

a recent work that generates random C programs to stress-test compilers. To date, it has found a few hundred bugs in GCC and LLVM, and helped improve the quality of the most widely-used C compilers. Despite this incredible success, the majority of the reported bugs were compiler crashes as it is difficult to steer its random program generation to specifically exercise a compiler's most critical components—its optimization phases. We defer to Section 6 for a detailed survey of related work.

Equivalence Modulo Inputs (EMI) This chapter introduces a simple, broadly applicable concept for validating compilers. Our vision is to take existing real-world code and transform it in a novel, systematic way to produce different, but equivalent variants of the original code. To this end, we introduce *equivalence modulo inputs* (EMI) for a practical, concrete realization of the vision.

The key insight behind EMI is to exploit the interplay between *dynamically executing* a program P on a subset of inputs and *statically compiling* P to work on all inputs. More concretely, given a program P and a set of input values I from its domain, the input set I induces a natural collection of programs $\mathcal{C}$ such that every program $Q \in \mathcal{C}$ is equivalent to P modulo I: $\forall i \in I, Q(i) = P(i)$. The collection $\mathcal{C}$ can then be used to perform differential testing [McKeeman, 1998] of any compiler $Comp$: If $Comp(P)(i) \neq Comp(Q)(i)$ for some $i \in I$ and $Q \in \mathcal{C}$, $Comp$ has a miscompilation.

Next we provide some high-level intuition behind EMI's effectiveness (Section 4.1 illustrates this insight with two concrete, real examples for Clang and GCC respectively). The EMI variants can specifically target a compiler's analysis and optimization phases, and stress-test them to reveal latent compiler bugs. Indeed, although an EMI variant Q is only equivalent to P modulo the input set I, the compiler has to perform all its (static) analysis and optimizations to produce correct code for Q over *all inputs*. In addition, P's EMI variants, while semantically equivalent *w.r.t.* I, can have quite different static data- and control-flow. Since data- and control-flow information critically affects which optimizations are enabled and how they are applied, the EMI variants not only help exercise the optimizer differently, but also demand the exact same output on I from the generated code by these different optimization strategies.

EMI has several unique advantages:

- It is general and easily applicable to finding bugs in compilers, analysis and transformation tools for any language.
- It can directly target miscompilations and stress-test critical, complex compiler components.
- It can generate and explore test cases based on real-world code that developers actually write. This means that any detected errors are more likely to manifest in real-world code, and thus more directly impact software vendors and users.

Indeed, we believe that the EMI concept is simple and can be adapted to validate compilers, program analysis, and program transformation systems in general. Potential applications include, for example, (1) testing and validating production compilers and software analysis tools; (2) generating realistic, comprehensive test suites for validation and certification; and (3) helping software vendors detect potential compiler-induced errors in their software, which can be desirable for safety- and mission-critical domains.

Compiler Bug Hunter: Orion Given a program P and an input set I, the space of P's EMI variants *w.r.t.* I is vast, and difficult or impossible to compute. Thus, for realistic use, we need a practical instantiation of EMI. We propose a "profile and mutate" strategy to systematically generate a subset of a program's EMI variants. In particular, given a program P and input set I, we profile executions of program P over the input set I, and derive (a subset of) P's EMI variants (*w.r.t.* I) by stochastically pruning, inserting, or modifying P's unexecuted code on I. These variants should clearly behave exactly the same on the same input set I as the original program P. We then feed these variants to any given compiler. Any detected deviant behavior on I indicates a bug in the compiler.

We have implemented our "profile and mutate" strategy for C compilers and focused on *pruning unexecuted code*. We have extensively evaluated our tool, Orion[1], in testing three widely-used C compilers—namely GCC, LLVM, and ICC—with extremely positive results (Section 4.4). We have used Orion to generate variants for real-world projects,

[1] Orion was a giant huntsman in Greek mythology.

existing compiler test suites, and much more extensively for test cases generated by Csmith [Yang et al., 2011]. In eleven months, we have reported, for GCC and LLVM alone, 147 confirmed, *unique* bugs. More than 100 have already been fixed, and more importantly, the majority of the bugs were miscompilations (rather than compiler crashes), clearly demonstrating the ability of EMI—offered by Orion—to stress-test a compiler's optimizer. We have also found and reported numerous bugs in ICC initially, but later we only focused on the two open-source compilers, GCC and LLVM, as both use open, transparent bug-tracking systems.

We have also done less, but still considerable testing of CompCert [Leroy, 2006, Leroy, 2009]. Besides a few confirmed front-end issues we found and reported, we have yet to encounter a bug in CompCert's verified components. This fact gives strong evidence of the promise and quality of verified compilers, although it is true that CompCert still supports only a subset of C and fewer optimizations than production compilers, such as GCC and LLVM.

4.1 Illustrating Examples

This section uses two concrete examples to motivate and illustrate our work: one for LLVM, and one for GCC.

In general, compiler bugs are of two main types, and they vary in severity. Some merely result in a *compiler crash*, causing minor nuisances and portability problems at times. Others, however, can cause compilers to silently miscompile a program and produce *wrong code*, subverting the programmer's intent. Miscompilations are daunting, and the following characteristics make them distinctive:

***Lead to bugs in other programs*:** Normally, a bug in a program only affects itself. Compilers generating wrong code can effectively inject bugs to programs they compile.

Hard to notice: If a miscompilation only affects a less traversed code path or certain optimization flags, it might go unnoticed during program development and only trigger in specific circumstances. Note that a rarely occurring bug can still be a

severe issue. This can be especially troublesome for compilers targeting embedded platforms and micro-controllers.

Hard to track down to the compiler: Popular mainstream compilers are generally considered very reliable (indeed they are), making them often the least to suspect when client code misbehaves.

Weaken source-level analysis and verification: Correctness guarantees at the source code level may be invalidated due to a compiler bug that leads to buggy binaries, thus hindering overall system reliability.

Impact the reliability of safety-critical systems: A seemingly unimportant miscompilation bug can potentially result in a critical flaw in a safety-critical system, thus making compiler reliability critically important.

These characteristics of compiler miscompilations make their effects more similar to bugs in hardware — and in the case of popular compilers, like bugs in widely-deployed hardware — than bugs in most other programs. EMI is a powerful technique for detecting various kinds of compiler bugs, but its power is most notable in discovering miscompilations.

Our tool, Orion, detects compiler bugs by applying EMI on source programs. For instance, we took the test program in 4.1a from the GCC test suite. It compiles and runs correctly on all compilers that we tested. We subsequently apply Orion on the test program. Clearly, none of the `abort` calls in the function `f` should execute when the program runs, and the coverage data confirms this. This allows Orion to freely alter the body of the `if` statements in the function `f` or remove them entirely without changing this program's behavior. By doing so, Orion transforms the program and produces many new test cases. One of these transformations, shown in 4.1b, is miscompiled by Clang on the 32-bit x86 architecture when optimizations are enabled. Figure 4.2 shows its reduced version that we used to report the bug.

A bug in the LLVM optimizer causes this miscompilation. The developers believe that the Global Value Numbering (GVN) optimization turns the struct initialization

```c
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
      struct tiny z, long l) {
  if (x.c != 10) abort();
  if (x.d != 20) abort();
  if (x.e != 30) abort();
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) abort();
  if (z.c != 12) abort();
  if (z.d != 22) abort();
  if (z.e != 32) abort();
  if (l != 123) abort();
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(a) Test `931004-11.c` from the GCC test suite; it compiles correctly by all compilers tested.

```c
struct tiny { char c; char d; char e; };
f(int n, struct tiny x, struct tiny y,
      struct tiny z, long l) {
  if (x.c != 10) /* deleted */;
  if (x.d != 20) abort();
  if (x.e != 30) /* deleted */;
  if (y.c != 11) abort();
  if (y.d != 21) abort();
  if (y.e != 31) /* deleted */;
  if (z.c != 12) abort();
  if (z.d != 22) /* deleted */;
  if (z.e != 32) abort();
  if (l != 123) /* deleted */;
}
main() {
  struct tiny x[3];
  x[0].c = 10;
  x[1].c = 11;
  x[2].c = 12;
  x[0].d = 20;
  x[1].d = 21;
  x[2].d = 22;
  x[0].e = 30;
  x[1].e = 31;
  x[2].e = 32;
  f(3, x[0], x[1], x[2], (long)123);
  exit(0);
}
```

(b) Test case produced by Orion by transforming the program in 4.1a, triggering a bug in Clang.

Figure 4.1: Orion transforms 4.1a to 4.1b and uncovers a miscompilation in Clang.

```
struct tiny { char c; char d; char e; };
void foo(struct tiny x) {
  if (x.c != 1) abort();
  if (x.e != 1) abort();
}
int main() {
  struct tiny s;
  s.c = 1; s.d = 1; s.e = 1;
  foo(s);
  return 0;
}
```

Figure 4.2: Reduced version of the code in Figure 4.1b for bug reporting. (`http://llvm.org/bugs/show_bug.cgi?id=14972`)

into a single 32-bit load. Subsequently, the Scalar Replacement of Aggregates (SRoA) optimization decides that the 32-bit load is undefined behavior, as it reads past the end of the struct, and thus does not emit the correct instructions to initialize the struct. The developer who fixed the issue characterized it as

> *"... very, very concerning when I got to the root cause, and very annoying to fix."*

The original program did not expose the bug because Clang decided not to inline the function `f` due to its size. In contrast, the pruned function `f` in the EMI variant became small enough that the Clang optimizer at -Os (and above) — when the inliner is enabled — decided to inline it. Once `f` was inlined, the incompatibility between GVN and SRoA led to the miscompilation. Indeed, using an explicit "inline" attribute on `f` in the original program also exposes this bug.

In another case, Orion derives the code in 4.3 from a program generated by Csmith [Yang et al., 2011], which was miscompiled by GCC 4.8 and the latest trunk revision at the time when optimizations were enabled in both 32-bit and 64-bit modes. The correct execution of this program will terminate immediately, as the continuation condition of the second `for` loop will always be `false`[2], never letting its loop body

[2]The variable `c` and other global variables are initialized to `0` in C.

```
int a, b, c, d, e;
int main() {
  for (b = 4; b > -30; b--)
    for (; c;)
      for (;;) {
        e = a > 2147483647 - b;
        if (d) break;
      }
  return 0;
}
```

Figure 4.3: GCC miscompiles this program to an infinite loop instead of immediately terminating with no output. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58731`)

execute. GCC with optimizations enabled miscompiles this program to an infinite loop. Interestingly, it does issue a bogus warning under `-O2`, but not `-O3`, which hints at the root cause of the miscompilation:

> "*cc1: warning: iteration 5u invokes undefined behavior [-Waggressive-loop-optimizations].*"

The warning implies that the sixth iteration of the outermost loop (when `b = -1`) triggers undefined behavior (*i.e.* signed integer overflow). In fact, there is no undefined behavior in this program, as the innermost loop is dead code and never executes, thus never triggering signed integer overflow at run time.

Partial Redundancy Elimination (PRE) detects the expression "`2147483647 - b`" as loop invariant. Loop Invariant Motion (LIM) tries to move it up from the innermost loop to the body of the outermost loop. Unfortunately, this optimization is problematic, as GCC then detects a signed overflow in the program's optimized version and this (incorrect) belief of the existence of undefined behavior causes the compiler to generate non-terminating code (and the bogus warning at `-O2`).

The original program did not trigger the bug because there were other statements in the innermost loop that mutated the variable `b` (for instance, increasing `b` before the

assignment to `e`). The expression "`2147483647 - b`" was thus determined not to be a loop invariant and was not hoisted outside the inner loops. The program ran as expected. On the other hand, since the innermost loop was not executed, Orion could freely modify its body. It generated some variants in which all statements mutating `b` were removed. As explained earlier, in these variants, the expression became loop invariant, and thus was hoisted out of the loop, which effectively triggered the bug. The original program is quite complex, having 2,985 LOC; the EMI variant that exposed the bug has 2,015 LOC.

The two examples demonstrate that bugs can appear in both small, and large, complex code bases, potentially resulting in hard-to-detect errors, crashes, or security exploits, even in entirely correct, even verified, programs. They also highlight the difficulty of correctly optimizing code. Not only each optimization pass can introduce bugs directly, the interactions among different optimizations can also lead to latent bugs. EMI, being an end-to-end testing methodology, detects bugs that occur across optimization passes, as well as those that occur within an individual pass.

4.2 Equivalence Modulo Inputs

The concept of equivalence modulo inputs (EMI) that we have outlined in Chapter 1 is simple and intuitive. The main goal of this section is to provide more detailed and precise definitions.

Rather than formalizing EMI for a concrete programming language, we operate on a generic programming language $\mathcal{L}$ with *deterministic*[3] semantics $[\![\cdot]\!]$, *i.e.*, repeated executions of a program $P \in \mathcal{L}$ on the same input i always yield the same result $[\![P]\!](i)$.

4.2.1 Definition

Two programs $P, Q \in \mathcal{L}$ are *equivalent modulo inputs (EMI) w.r.t.* an input set I common to P and Q (*i.e.*, $I \subseteq \mathsf{dom}(P) \cap \mathsf{dom}(Q)$) iff

$$\forall i \in I \;\; [\![P]\!](i) = [\![Q]\!](i).$$

We use $[\![P]\!] =_I [\![Q]\!]$ to denote that P and Q are EMI *w.r.t.* input set I.

[3]Note that we may also force a non-deterministic language to assume deterministic behavior.

For the degenerate case where P and Q do not take inputs (*i.e.*, they are closed programs), EMI reduces to semantic equivalence:

$$[\![P]\!] = [\![Q]\!].$$

Or more precisely, P and Q are EMI *w.r.t.* the input set $\{\texttt{void}\}$, where `void` denotes the usual "*no argument*":

$$[\![P]\!](\texttt{void}) = [\![Q]\!](\texttt{void}).$$

For example, the GCC test `931004-11.c` and the output code from Orion shown respectively in Figures 4.1a and 4.1b are EMI (*w.r.t.* $I = \{\texttt{void}\}$).

Given a program $P \in \mathcal{L}$, any input set $I \subseteq \mathsf{dom}(P)$ naturally induces a collection of programs $Q \in \mathcal{L}$ that are EMI (*w.r.t.* I) to P. We call this collection P's *EMI variants*.

Definition 4.2.1 (EMI Variants) *A program P's* EMI variants w.r.t. *an input set I is given by:*

$$\{Q \in \mathcal{L} \mid [\![P]\!] =_I [\![Q]\!]\}.$$

It is clear that EMI is a relaxed notion of semantic equivalence:

$$[\![P]\!] = [\![Q]\!] \quad \Longrightarrow \quad [\![P]\!] =_I [\![Q]\!].$$

4.2.2 Differential Testing with EMI Variants

At this point, it may not be clear yet what benefits our relaxed notion of equivalence can provide, which we explain next.

Differential Testing: An Alternative View Our goal is to differentially test compilers. The traditional view of differential testing [McKeeman, 1998] is simple: If two programs (in our setting, compilers or compiler versions) "act differently" on some input (*i.e.* source programs), we have found a bug in one of the compilers (maybe also in both). This is, for example, the view taken by Csmith [Yang et al., 2011] (assuming that the input programs are well-behaving, *e.g.*, they do not exhibit any undefined behavior).

We adopt an alternative view: If an oracle can generate a program P's semantic equivalent variants, these variants can stress-test any compiler $Comp$ by checking whether

$Comp$ produces equivalent code for these variants. This view is attractive because we can (1) operate on existing code (or randomly generated, but valid code), and (2) check a single compiler in isolation (*e.g.* where competing compilers do not exist). However, we face two difficult challenges: (1) How to generate semantic equivalent variants? and (2) How to check equivalence of the produced code? Both have been long-standing challenges in software analysis and verification.

The "Profile and Mutate" Strategy Our key insight is that EMI provides a practical mechanism to realize our alternative view for differential testing of compilers. Indeed, by relaxing semantic equivalence *w.r.t.* an input set I, we reduce the second challenge to the simple task of testing against I. As for the first challenge, note that P's executions on I yield a *static slice* of P and unexecuted "dead code". One may freely mutate the "dead code" without changing P's semantics on I, thus providing a potentially enormous number of EMI variants to help stress-test compilers.

Once the EMI variants are generated, testing is straightforward. Let $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$ be a set of P's EMI variants *w.r.t.* I. For each $Q_i \in \mathcal{Q}$, we verify the following:

$$\forall i \in I \;\; Comp(Q_i)(i) = Comp(P)(i).$$

Any deviant behavior indicates a miscompilation.

So far, we have not specified how to "mutate" the unexecuted "dead code" *w.r.t.* I. Obvious mutations include pruning, insertion, or modification. Our implementation, which we describe next, focuses on pruning, and we show in evaluation that even such a simple realization is extremely effective — it has detected 147 unique bugs for GCC and LLVM alone in under a year. We discuss other mutation strategies in Section 5.2.

4.3 Implementation of Orion

We now describe Orion, our practical realization of the EMI concept targeting C compilers via the "profile and prune" strategy. At a high level, Orion operates on a program's abstract syntax tree (AST) and contains two key steps: (1) extracting coverage information (Section 4.3.1), and (2) generating EMI variants (Section 4.3.2).

One challenge for testing C compilers is to avoid programs with undefined behavior because the C standard allows a compiler to do anything with such programs. For example, one major, painstaking contribution of Csmith is to generate valid test programs most of the time. In this regard, Orion has a strong advantage. Indeed, the EMI variants generated by Orion do not exhibit any undefined behavior if the original program has no undefined behavior (since only dead code is pruned from the original program). This allows Orion to easily generate many valid variants from a single valid seed program.

Algorithm 11 describes Orion's main process. As its first step, Orion profiles the test program P's execution on the input set I to collect (1) *coverage information* and (2) the *expected output* on each input value $i \in I$ (lines 2–3). It then generates P's EMI variants *w.r.t.* I (lines 5–6), and uses them to validate each compiler configuration against the collected reference output (lines 7–11). Next, we discuss each step in detail.

4.3.1 Extracting Coverage Information

Code coverage tools compute how frequently a program's statements execute during its profiled runs on some sample inputs. We can conveniently leverage such tools to track the executed (*i.e.* "covered") and unexecuted (*i.e.* "dead") statements of our test program P under input set I. Those statements marked "dead" are candidates for pruning in generating P's EMI variants.

In particular, Orion uses `gcov` [GNU Compiler Collection, 2015], a mature utility in the GNU Compiler Collection, to extract coverage information. We enable `gcov` by compiling the test program P with the flag "`-O0 -coverage`", which instruments P with additional code to collect coverage information at runtime. Orion then executes the instrumented executable on the provided input set I to obtain coverage files with information indicating how many times a line has been executed.

Because `gcov` profiles coverage at the line level, it may produce imprecise results when multiple statements are on a single line. For example, in the example below,

```
if (false) { /* this could be removed */ }
```

`gcov` marks the entire line as executed. As a result, Orion cannot mutate it, although the statements within the curly braces could be safely removed. Note that we manually

Algorithm 11: Orion's main process for compiler validation

```
1  procedure Validate (Compiler Comp, TestProgram P, InputSet I):
2      begin
           /* Step 1:  Extract coverage and output */
3          P_exe := Comp.Compile(P,"-O0")                    /* without optimization */
4          C := ⋃_{i∈I} C_i, where C_i := Coverage(P_exe.Execute(i))
5          IO := {⟨i, P_exe.Execute(i)⟩ | i ∈ I}
           /* Step 2:  Generate variants and verify */
6          for 1..MAX_ITER do
7              P' := GenVariant(P, C)
               /* Validate Comp's configurations */
8              foreach σ ∈ Comp.Configurations() do
9                  P'_exe := Comp.Compile(P', σ)
10                 foreach ⟨i, o⟩ ∈ IO do
11                     if P'_exe.Execute(i) ≠ o then
                           /* Found a miscompilation */
12                         ReportBug (Comp, σ, P, P', i)
```

formatted the two test cases in Figure 4.1 for presentation. The actual code has every "`abort();`" on a separate line.

Occasionally, coverage information computed by `gcov` can also be ambiguous. For instance, in the sample snippet below (extracted from the source code of the Mathomatic[4] computer algebra system), `gcov` marks line 2613 as unexecuted (indicated by prepending the leading "#####"):

```
#####: 2613:  for (;; cp = skip_param(cp)) {
                    .....
    7: 2622:        break;
#####: 2623:  }
```

[4] `http://www.mathomatic.org/`

Based on this information, Orion assumes that it can remove the entire `for` loop (lines 2613–2623). This is incorrect, as the `for` loop is actually executed (indicated by the execution of its child statement `break`). What `gcov` really means is that the expression "`cp = skip_param(cp)`" is unevaluated. We remedy this coverage ambiguity by verifying that none of the children of an unexecuted statement is executed before removing it in the next step.

To avoid the aforementioned problems caused by collecting coverage statistics at line granularity, we could modify `gcov` or implement a new code coverage tool that would operate at the statement level. This can make our analysis more precise and help generate more variants. However, the practical benefits seem negligible as often there are only few such impacted statements. Our extremely positive results (Section 4.4) demonstrate that the use of `gcov` has been a good, well-justified decision.

4.3.2 Generating EMI Variants

Orion uses LLVM's LibTooling library [The Clang Team, 2014] to parse a C program into an AST and mutate it based on the computed coverage information to generate the program's EMI variants.

The mutation process happens at the statement level in the AST. We mark a statement unexecuted if (1) the line number of its first token is marked unexecuted by `gcov`, and (2) none of its child statements in the AST is executed. When Orion decides to prune a statement, it removes all tokens in its AST subtree, including all its child statements. Thus, all variants generated by Orion are syntactically correct C programs.

Algorithm 12 describes Orion's process for generating EMI variants. The process is simple — We traverse all the statements in the original program P and randomly prune the unexecuted "dead" statements. On line 9, we use the function `FlipCoin` to decide stochastically whether an unexecuted statement s should be kept or removed. We control Orion's pruning via two parameters, P_{parent} and P_{leaf}, which specify the probabilities to prune parent or leaf statements in `FlipCoin`. One can use static probabilities for deleting statements and uniformly vary these values across different runs. An alternative is to allow dynamically adjusted probabilities for each statement. From our experience,

Algorithm 12: Generate an EMI variant

```
1  function GenVariant (TestProgram P, Coverage C): Variant P′:
2      begin
3          P′ := P
4          foreach s ∈ P′.Statements() do
5              PruneVisit (P′, s, C)
6          return P′
7  procedure PruneVisit (TestProgram P′, Statement s, Coverage C):
8      begin
           /* Delete this statement when applicable */
9          if s ∉ C and FlipCoin(s) then
10             P′.Delete (s)
11             return
           /* Otherwise, traverse s's children */
12         foreach s′ ∈ s.ChildStatements() do
13             PruneVisit (P′, s′, C)
```

this additional dynamic control seems quite effective. In fact, our standard setup is to randomly adjust these two parameters after each statement pruning by resetting each to an independent random probability value from 0.0 to 1.0.

In our actual implementation, Algorithm 11 is realized using shell scripts. In particular, we have a set of scripts to collect coverage and reference output information, control the outer loop, generate EMI variants and check for possible miscompilation or compiler crashes. We have implemented Algorithm 12 in C++ using LLVM's LibTooling library [The Clang Team, 2014]. Unlike random program generators such as Csmith, Orion requires significantly less engineering effort. It has approximately 500 lines of shell scripts and 1,000 lines of C++ code, while Csmith contains about 30-40K lines of C++ code.

4.4 Evaluation

This section presents our extensive evaluation of Orion to demonstrate the practical effectiveness of our EMI methodology besides its conceptual elegance and generality.

Since January 2013, we have been experimenting with and refining Orion to find new bugs in three widely-used C compilers, namely GCC, LLVM, and ICC. We have also occasionally tested CompCert [Leroy, 2006, Leroy, 2009], a formally verified C compiler. In April 2013, we started our extensive testing of GCC, LLVM, and ICC. After finding and reporting numerous bugs in ICC, we stopped testing it for the lack of direct communication with its developers (although we did learn afterward by checking its later releases that many bugs we reported had been fixed). Since then, we have only focused on GCC and LLVM because both have open bug repositories, and transparent bug triaging and resolution. This section describes the results from our extensive testing effort for about eleven months.

Result Summary Orion is extremely effective:

- *Many confirmed bugs*: In eleven months, we have found and reported 147 confirmed, unique bugs in GCC and LLVM alone.
- *Many long-latent bugs*: Quite a few of the detected bugs have been latent for many years, and resisted the attacks from both earlier and contemporary tools.
- *Many have been already fixed*: So far, 110 of the bugs have already been fixed and resolved; most of the remaining ones have been triaged, assigned, or are being actively discussed.
- *Most are miscompilations*: This is perhaps the most important, clearly demonstrating the strengths of EMI for targeting the hard-to-find and more serious miscompilations (rather than compiler crashes). For example, Orion has already found about the same number (around 40) of miscompilations in GCC as Csmith did, but over several years' prior and continuing testing.

4.4.1 Testing Setup

Hardware and Compiler Our testing has focused on the x86-linux platform. Since late April 2013, we have performed our testing on two machines (one 18 core and one 6 core) running Ubuntu 12.04 (x86_64). For each compiler (*i.e.* GCC and LLVM), we test its latest development version (usually built once daily) under the most common configurations (*i.e.* `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`), generating code for both 32-bit (`-m32`) and 64-bit (`-m64`) environments. We did not use any finer-grained combinations of the compilers' optimization flags.

Test Programs In our testing, we have drawn from three sources of test programs to generate their EMI variants (note that none of the original test programs triggered a compiler bug):

- *Compiler Test Suites*: Each of GCC and LLVM has an already sizable and expanding regression test suite, which we can use for generating EMI variants (which in turn can be used to test any compiler). For example, the original test case shown in Figure 4.1 was from the GCC test suite, and one of its EMI variants helped reveal a subtle miscompilation in LLVM. We used the approximately 2,000 collected tests from the KCC [Ellison and Rosu, 2012] project, an executable formal semantics for C. Among others, this collection includes tests primarily from regression test suites of GCC and LLVM. The programs in these test suites do not take inputs, and are generally quite small. Nonetheless, we were able to find bugs by pruning them. The problem with this source is that the number of bugs revealed by their EMI variants saturated quickly, which is expected as they have few unexecuted statements.

- *Existing Open-Source Projects*: Another interesting source is the large number of open-source projects available. One challenge to use such a project is that its source code usually scatters across many different directories. Fortunately, these projects normally use the GNU build utilities (*e.g.* "`configure`" followed by "`make`") and do often come with a few test inputs (*e.g.* invoked by "`make test`" or "`make check`"), which we can leverage to generate EMI variants.

In particular, we modify a project's build process to generate coverage information for "`make test`". To generate an EMI variant for the project, we bootstrap its build process. Before compiling a file, we invoke our `transformer` tool that transforms the file into an EMI file, which is then compiled as usual. The output from the build process is an EMI variant of the original project. Then, we can use it to test each compiler simply by running the accompanying "`make test`" (or "`make check`"). If checking fails on the variant under a particular compiler configuration, we have discovered a compiler bug.

Now, we face another challenge, that is how to reduce the bug-triggering EMI variant for bug reporting. This is a more serious challenge, particularly for miscompilations. Although we have applied Orion on a number of open-source projects—including all nine SPEC2006 integer C programs, Mathomatic and `tcc`[5]—and found many inconsistencies, we were only able to reduce one GCC crashing bug triggered by an EMI variant of `gzip`. We are yet to reduce the others, such as an interesting GCC miscompilation triggered by a variant of `tcc`.

- *Csmith-Generated Random Code*: Csmith turns out to be an excellent source for providing an enormous number of test programs for Orion. Programs generated by Csmith, which do not take inputs, are generally complex and offer quite rich opportunities for generating EMI variants. Our Csmith setup produces programs with an average size of 4,590 LOC, among which 1,327 lines on average are unexecuted. This corresponds to a vast space of EMI variants. More importantly, these Csmith-variants can often be effectively reduced using existing tools such as C-Reduce [Regehr et al., 2012] and Berkeley Delta [McPeak et al., 2015]. Thus, most of our testing has been on top of the random programs generated by Csmith, running in its "swarm testing" setup [Groce et al., 2012].

Test Case Reduction Test case reduction is still a significant and time-consuming challenge. Our experience suggests that neither C-Reduce nor Berkeley Delta is the most

[5]The "Tiny C Compiler" (`http://bellard.org/tcc/`)

effective on its own. We have devised an effective *meta-process* to utilize both. It is a nested fixpoint loop. First, we use Delta to repeatedly reduce the test case until a fixpoint has been reached (*i.e.* no additional reduction from Delta). Then, we run C-Reduce on the fixpoint output from Delta. We repeat this two-step process until reaching a fixpoint. This meta-process strikes a nice balance to take advantage of Delta's better efficiency and C-Reduce's stronger reduction capability.

There is another challenge: How to reject code with undefined behavior in test case reduction? We follow C-Reduce [Regehr et al., 2012] and leverage (1) GCC and LLVM warnings, (2) KCC [Ellison and Rosu, 2012], and (3) static analysis tools such as Frama-C.[6] We also utilize Clang's (although imperfect) support for undefined behavior sanitization, as well as cross-checking using a few different compilers and compiler configurations to detect inconsistent behavior caused by invalid code (*i.e.* code with undefined behavior). Whenever possible, we use CompCert (and its C interpreter) for detecting and rejecting invalid code.

Number of Variants It is also important to decide how many variants to generate for each program. There is a clear trade-off in performance and bug detection. Our experience suggests that eight variants appear to strike a good balance. In earlier testing, we used a static bound, such as 8, as the number of variants to generate for each program. Later, we added a random parameter that has roughly an expected value of 8 to control the number of generated variants for each test program independently at random. This has been quite effective. For future work, we may explore white-box approaches that support less stochastic, more controlled generation of EMI variants.

We have conducted our testing over an extended period of time. We usually had multiple runs of Csmith with different configurations, especially after we learned that certain Csmith configurations may often lead to test cases with undefined behavior. One such example is the use of unions because code with unions can easily violate the strict aliasing rules, thus leading to undefined behavior. Sometimes, we also needed to restart due to system failures, Csmith updates, bugs in Orion, and re-seeding Csmith's random

[6] `http://frama-c.com/`

number generation. As the number of EMI variants that we generated for each test case was also stochastic, we do not have exact counts of the Csmith tests and their derived EMI variants, but both numbers were in the millions to tens of millions range.

4.4.2 Quantitative Results

This subsection presents various summary statistics on results from our testing effort.

Bug Count We have filed a total of 195 bug reports for GCC and LLVM during our testing period: (1) three derived from compiler test suites, (2) one from existing open-source projects, and (3) the rest from Csmith tests. They can be found under "su@cs.ucdavis.edu" and "dhazeghi@yahoo.com" in GCC's and LLVM's bugzilla databases. Till March 2014, 147 have been confirmed, 110 of which have been resolved and fixed by the developers. Note that when a bug is confirmed and triaged, it corresponds to a new defect. Thus, all of our confirmed bugs were unique and independent.

Also note that although we always ensured that all of our reported bugs had different symptoms, some of them were actually linked to the same root cause. These bugs were later marked as duplicate by developers. The remaining 13 bugs — 4 for GCC and 9 for LLVM — have not yet been confirmed as the developers have not left any comments on these reports.

Table 4.1 classifies the reported bugs across the two tested compilers: GCC and LLVM. It is worth mentioning that we have focused more extensive testing on GCC because of the very quick responses from the GCC developers and relatively slow responses from the LLVM developers (although later we had seen much increased activities from LLVM because of its 3.4 release). This partially explains why we have reported more bugs for GCC over LLVM.

Bug Types We distinguish two kinds of errors: (1) ones that manifest when compiling code, and (2) ones that manifest when the compiled EMI variants are executed. We further classify compile-time bugs into *compiler crashes* (*e.g.* internal compiler errors and memory-safety errors) and *performance bugs* (*e.g.* compiler hang or abnormally slow compilation).

	GCC	LLVM	TOTAL
Reported	111	84	195
Marked duplicate	28	7	35
Confirmed	79	68	147
Fixed	56	54	110

Table 4.1: Bugs reported, marked duplicate, confirmed, and fixed.

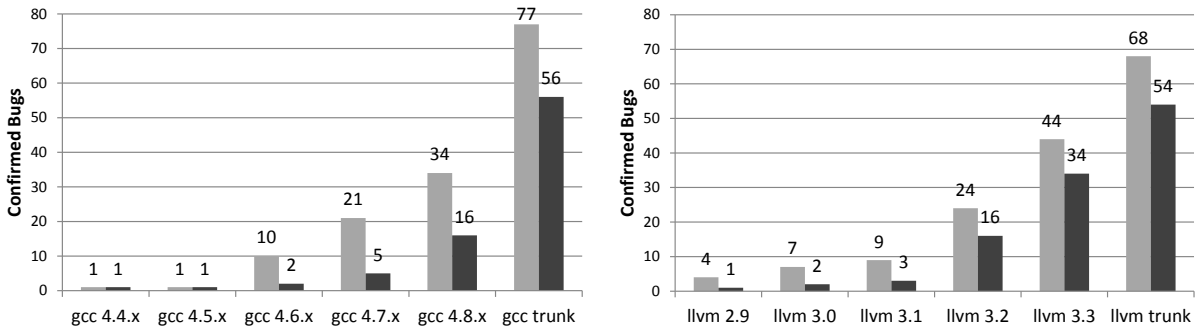
A compile-time crash occurs when the compiler exits with a non-zero status. A runtime bug occurs when an EMI variant behaves differently from its original program. For example, it crashes or terminates abnormally, or produces a different output. We refer to such compiler errors as *wrong code* bugs. Silent wrong code bugs are the most serious, since the program surreptitiously produces wrong result.

Table 4.2 classifies the bugs found by Orion according to the above taxonomy. Notice that Orion found many wrong code (more serious) bugs, confirming its strengths in stress-testing compiler optimizers. For example, Csmith found around 40 wrong code bugs in GCC over several years' prior and continuing testing, while Orion found about the same number of wrong code bugs in a much shorter time (and after GCC and LLVM had already fixed numerous bugs discovered by Csmith).

	GCC	LLVM	TOTAL
Wrong code	46	49	95
Crash	23	10	33
Performance	10	9	19

Table 4.2: Bug classification.

Importance of the Reported Bugs It is reasonable to ask whether the compiler defects triggered by randomly pruning unexecuted code matter in practice. This is difficult to answer and a question that Csmith has also faced. The discussion from the Csmith paper [Yang et al., 2011] is quite relevant here. First, most of our reported bugs have been confirmed and fixed by the developers, illustrating their relevance and importance.



(a) GCC (b) LLVM

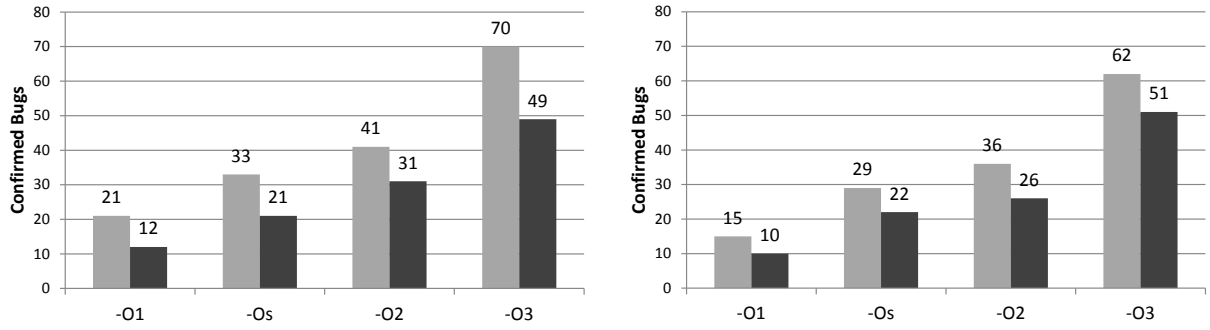
Figure 4.4: Affected versions of GCC and LLVM (*lighter bars*: confirmed bugs; *darker bars*: fixed bugs).

Second, some of our reported bugs were later reported by others when compiling real-world programs. As a recent example, from an EMI variant of a Csmith test, we found a miscompilation in GCC and reported it as bug 59747.[7] Later, others discovered that GCC also miscompiled the movie player `mplayer`, and filed a new bug report 59824.[8] The two bugs turned out to share the same root cause, and subsequently bug 59824 was marked as duplicate.

Affected Compiler Versions We only tested the latest development trunks of GCC and LLVM. When we find a test case that reveals a bug in a compiler, we also check the compiler's stable releases against the same test case. Figure 4.4 shows the numbers of bugs that affect various versions of GCC and LLVM. Obviously both development trunks are the most frequently affected. However, Orion has also found a considerable number of bugs in many stable releases that had been latent for many years.

Optimization Flags and Modes Figure 4.5 shows which optimization levels are affected by the bugs found in GCC and LLVM. In general, a bug occurs at lower optimization levels is likely to also happen at higher levels. However, we did encounter cases where a bug only affected one optimization flag. In most such cases, the flag is "`-Os`", which is quite intuitive because "`-Os`" is the only flag that optimizes for code size and is less used. Table 4.3 shows the number of bugs that affected code generated for 32-bit ("`-m32`") and

[7] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=59747`
[8] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=59824`



(a) GCC (b) LLVM

Figure 4.5: Affected optimization levels of GCC and LLVM (*lighter bars*: confirmed bugs; *darker bars*: fixed bugs).

64-bit ("`-m64`") environments, alone or both.

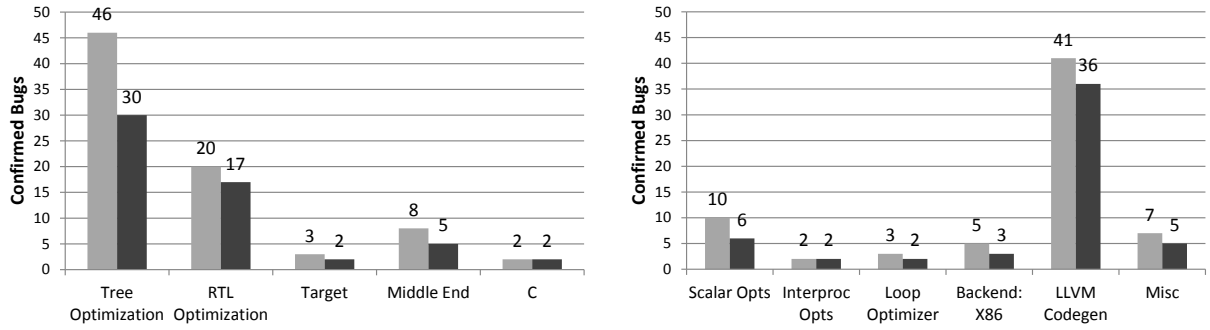
	GCC	LLVM	TOTAL
`-m32` alone	15	10	25
`-m64` alone	21	18	39
Both	43	40	83

Table 4.3: Bugs found categorized by modes.

Affected Compiler Components Figure 4.6 shows which compiler components in GCC and LLVM were affected by the reported bugs. Most of the bugs that Orion found in GCC are optimizer bugs. As for LLVM, the developers do not (or have not) appropriately classify the bugs, so the information we extracted from the LLVM's bugzilla database may be quite skewed, where most have been classified as "LLVM Codegen" thus far.

4.4.3 Assorted Bug Samples Found by Orion

Orion is capable of finding bugs of diverse kinds. We have found bugs that result in issues like compiler segfaults, internal compiler errors (ICEs), performance issues at compilation, and wrong code generation, and with various levels of severity, from rejecting valid code to release-blocking miscompilations. To provide a glimpse of the diversity of the uncovered bugs, we highlight here several of the more concise GCC and LLVM bugs. The wide variety of bugs demonstrates EMI's power and broad applicability.



(a) GCC (b) LLVM

Figure 4.6: Affected components of GCC and LLVM (*lighter bars*: confirmed bugs; *darker bars*: fixed bugs).

Miscompilations We first discuss a few selected wrong code bugs:

4.7a: All versions of GCC tested (4.6 to trunk) failed to correctly compile the program shown in 4.7a in 64-bit mode under `-O3`. The resulting code crashes with a segfault. The reason is believed to be a wrong offset computation in GCC's predictive commoning optimization. The generated code tries to access memory quite far from what it actually should access due to incorrectly generated offsets, causing a segmentation fault. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58697`)

4.7b: When compiling the code in 4.7b, Clang generated incorrect code, making the program return an incorrect value. The bug is caused by Clang's vectorizer. (`http://llvm.org/bugs/show_bug.cgi?id=17532`)

4.7c: GCC trunk failed to compile the program listed in 4.7c at `-O1` and above in both 32-bit and 64-bit modes because of a bug in its jump threading logic. The shape of the control-flow graph caused the code to handle jump threads through loop headers to fail. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58343`)

4.7d: Clang trunk failed to compile the test case in 4.7d and crashed with a segfault under `-O2` and above in both 32-bit and 64-bit modes. The problem was caused by GVN forgetting to add an entry to the leader table when it fabricated a "ValNum" for a dead instruction. Later on, when the compiler wants to access that table entry,

```
int b, f, d[5][2];
unsigned int c;
int main() {
  for (c = 0; c < 2; c++)
    if (d[b + 3][c]
        & d[b + 4][c])
      if (f) break;
  return 0;
}
```

(a) All tested GCC versions generated wrong code that crashed at run-time due to invalid memory access, when compiled at `-O3` in 64-bit mode (GCC bug 58697).

```
int main() {
  int a = 1; char b = 0;
  lbl:
    a &= 4;
    b--;
    if (b) goto lbl;
  return a;
}
```

(b) Clang bug affecting 3.2 and above: the vectorizer generates incorrect code affecting the program's return value. The bug disappears with `-fno-vectorize`. (LLVM bug 17532)

```
int a;
int main() {
  int b = a;
  for (a = 1; a > 0; a--);
lbl:
  if (b && a) goto lbl;
  return 0;
}
```

(c) GCC trunk crashed at `-O1` and above with an Internal Compiler Error (ICE) due to the unusual shape of the control-flow graph which causes problems in the jump threading logic and leads to a failure. (GCC bug 58343)

```
int *a, e, f;
long long d[2];
int foo() {
  int b[1]; a = &b[0];
  return 0;
}
int bar() {
  for (f = 0; f < 2; f++)
    d[f] = 1;
  e = d[0] && d[1] - foo();
  if (e) return 0;
  else return foo();
}
```

(d) Clang trunk segfaulted when compiled at `-O2` or `-O3` due to GVN's incorrectly updating the leader table. (LLVM bug 17307)

```
struct S0 {
  int f0, f1, f2, f3, f4 }
    b = {0,0,1,0,0};
int a;
void foo(struct S0 p) {
  b.f2 = 0;
  if (p.f2) a = 1;
}
int main() {
  foo(b);
  printf("%d\n", a);
  return 0;
}
```

(e) Clang trunk miscompiled this program when optimized for code size (`-Os`) as a result of an LLVM inliner bug, generating incorrect output. (LLVM bug 17781)

```
int a[8][8] = {{1}};
int b, c, d, e;
int main() {
  for (c = 0; c < 8; c++)
    for (b = 0; b < 2; b++)
      a[b + 4][c] = a[c][0];
  printf("%d\n", a[4][4]);
  return 0;
}
```

(f) GCC vectorizer regression from 4.6 triggers a miscompilation affecting program output under `-O3`. The bug goes away with the `-fno-tree-vectorize` flag. (GCC bug 58228)

Figure 4.7: Example test cases uncovering a diverse array of GCC and LLVM bugs.

it fails with a segfault as the entry is nonexistent. (`http://llvm.org/bugs/show_bug.cgi?id=17307`)

4.7e: The test program in 4.7e was miscompiled by Clang trunk when optimized for code size (*i.e.* at `-Os`), causing the binary to print "0" when executed where it should have printed "1". The root cause was traced to a bug in the LLVM inliner. (`http://llvm.org/bugs/show_bug.cgi?id=17781`)

4.7f: GCC's vectorizer was not immune to Orion either. It miscompiled the program in 4.7f, resulting in wrong output from executing the generated code. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58228`)

Compiler Performance Issues Orion also helps discover another category of bugs: compiler performance bugs resulting in terribly slow compilation. For instance, it took GCC minutes to compile the program in 4.8, orders of magnitude slower than both Clang and ICC. Across different versions, GCC 4.8 was considerably faster than trunk, whereas GCC 4.6 and 4.7 were much slower. The performance issue is believed to be caused by loop unrolling while retaining a large number of debug statements ($> 500,000$) within a single basic block that will have to be traversed later.

While Clang was much faster than GCC at compiling the program in 4.8, it had performance bugs elsewhere. Both Clang 3.3 and trunk failed to perform satisfactorily in compiling the code in Figure 4.9, taking minutes to compile the code under `-O3`, orders of magnitude longer than `-O2`, GCC, ICC, and even the previous Clang release (3.2), which took 9 seconds to compile at `-O3`. GCC had the fastest compilation — only 0.19 seconds at `-O3`.

Clang's performance issue was caused by its creation of thousands of stale lifetime marker objects within the compiler that are not properly cleaned up, drastically slowing down compilation.

4.4.4 Remarks

One of the reasons why Csmith has not been extended to C++ or other languages is because it requires significant engineering efforts to realize. One essentially has to

```
int a, b, c, d;
int *foo (int *r, short s, short t) {
  return &c;
}

short bar(int p) {
  int t = 0;
  for (a = 0; a < 8; a++)
    for (b = 0; b < 8; b++)
      for (p = 0; p < 8; p++)
        for (d = 0; d < 8; d++) {
          foo(&t, p, d);
        }
  bar (0);
  return 0;
}

int main() {
  return 0;
}
```

Figure 4.8: GCC retains many debug statements that will have to be traversed in a single basic block as a result of loop unrolling, causing orders of magnitude slowdown in compilation speed. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=58318`)

```
int a = 1, b, c, *d = &c, e, f, g, k, l, x;
static int * volatile *h = &d;
static int * volatile **j = &h;
void foo(int p) { d = &p; }
void bar() {
  int i;
  foo (0);
  for (i = 0; i < 27; ++i)
    for (f = 0; f < 3; f++)
      for (g = 0; g < 3; g++) {
        for (b = 0; b < 3; b++)
          if (e) break;
        foo (0); }
}
static void baz() {
  for (; a >= 0; a--)
    for (k = 3; k > 0; k--)
      for (l = 0; l < 6; l++) {
        bar (); **j = &x; }
}
int main() { baz(); return 0; }
```

Figure 4.9: It takes Clang 3.3+ minutes to compile at `-O3`, compared to only 0.19 seconds with GCC 4.8.1. Clang created a large number of lifetime marker objects and did not clean them up. (`http://llvm.org/bugs/show_bug.cgi?id=16474`)

rebuild a new program generator almost entirely from scratch. In contrast, Orion is much easier to be targeted for a new domain. To add C++ support to Orion, we simply need to handle a few more C++ statements and constructs in the generator `transformer`. Although we have not yet done any substantial testing of Orion's C++ support, our preliminary experience has been encouraging as Orion has already uncovered potential latent compiler bugs using small existing C++ code.

Even for a completely new domain, EMI also requires much less engineering effort because it can leverage existing tools and infrastructures. Our active ongoing work on adapting Orion to testing the JVM JIT confirms this.

Currently, Orion only focuses on integer C programs. We plan to extend the work to floating-point programs. This direction is new and exciting, and our EMI methodology offers a promising high-level approach. The key technical challenge is to define the "equivalence" of floating-point EMI variants considering the inherent inaccuracy of floating-point computation.

Chapter 5

Other Dimensions of Program Synthesis

Previous chapters have shown the practicality and effectiveness of program synthesis. This chapter further explores program synthesis in the following three dimensions. First, we investigate various user interaction models to improve the *usability* of program synthesis techniques (Section 5.1). Second, we discuss a guided, stochastic synthesis technique that helps improve synthesis *coverage*, thereby reveal many deep compiler bugs that otherwise could not be found (Section 5.2). Finally, we illustrate the general *applicability* of program synthesis by applying it to finding bugs in a new domain: link-time optimizers (Section 5.3).

5.1 New User Interaction Models: Program Navigation and Conversational

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 that, 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.

This section presents two novel user interaction models that communicate actionable

information to the user to help resolve ambiguity in the examples. One model allows 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. FlashProg, a web application that enables data extraction from textual documents, spreadsheets, and Web pages using examples, is our user interface that realizes the two interaction models.

5.1.1 FlashProg's User Interface

Figure 5.1 shows FlashProg's interface after providing some examples (top), and after finishing learning (bottom). The interface consists of 3 sections: (1) Top Toolbar, (2) Input Text View, and (3) PBE Interaction View.

Top Toolbar The Top Toolbar contains: (a) an Input button to open and upload files, (b) a Reset button to reset FlashProg to the initial state, (c) Undo/Redo buttons to undo/redo interaction steps, and (d) a Results button to export the output to CSV file.

Input Text View The Input Text View is the main area. It allows users to highlight desired document sections as examples (the examples can be nested). FlashProg then initiates the learning process and shows the highest ranked program in the Output pane. Each new row in the output corresponds to a region in the document (which is highlighted with dimmer colors). The scroll bars are colored with a *bird's-eye view* of highlighting, which makes it easier to identify discrepancies in output. The user can also provide *negative examples* by clicking on previously marked regions to communicate to FlashProg that the region should not be selected as part of the output.

PBE Interaction View The tabbed-pane PBE Interaction View lets users interact with FlashProg 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).

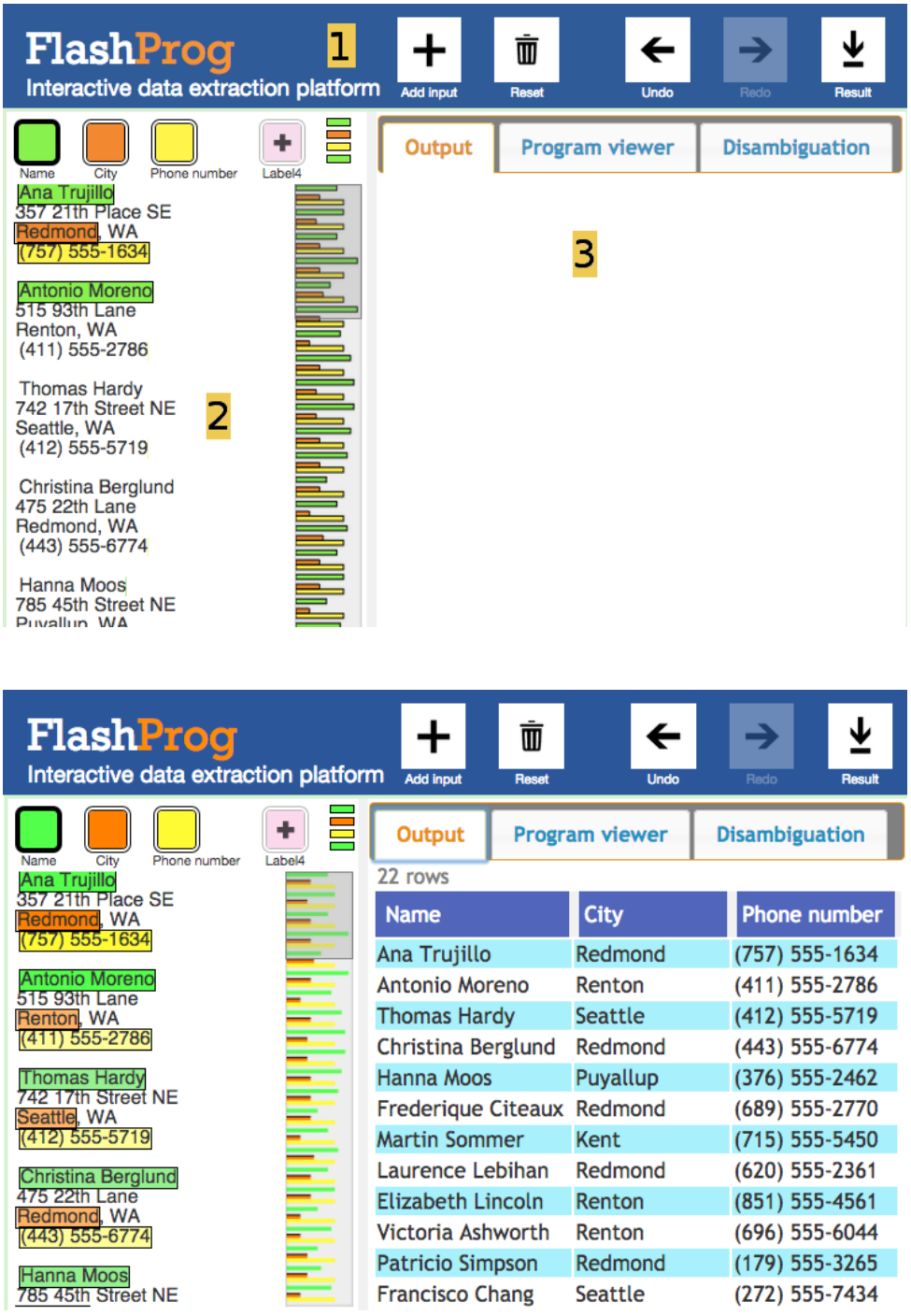


Figure 5.1: FlashProg UI with PBE Interaction View in the “Output” mode, before and after the learning process. 1 – Top Toolbar, 2 – Input Text View, 3 – PBE Interaction View.

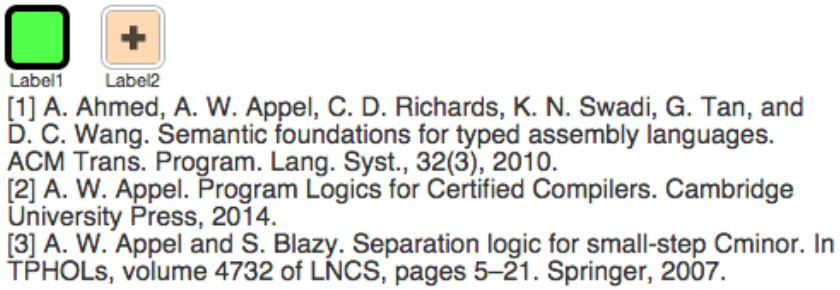


Figure 5.2: Initial input to FlashProg in our illustrative scenario: extraction of the author list from the PDF bibliography of "A Formally-Verified C Static Analyzer".

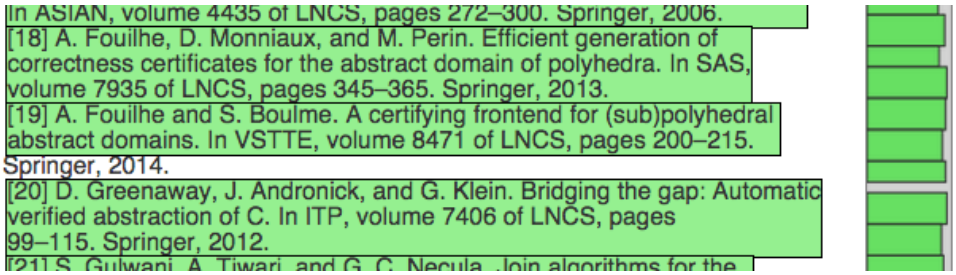


Figure 5.3: Bird's eye view showing discrepancy in extraction.

5.1.2 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 5.2). Our model user Alice wants to extract this data to figure out who is the most cited author in papers presented at POPL 2015.

Extracting Publication Records 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. However, the user notices that there is an unexpected gap between two extracted regions using the bird's-eye view (Figure 5.3). Giving another example to also include the missing text "`Springer, 2014.`" fixes the problem and a correct program is learned for the publication record regions.

Extracting Lists of Authors Next, Alice wants to extract the list of authors and provides an example inside the first record. She observes that the program learned is behaving

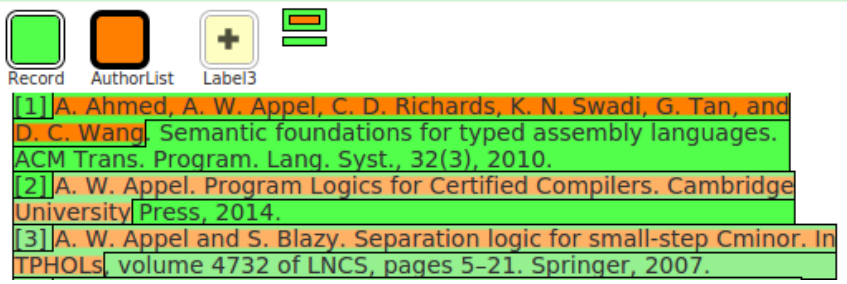


Figure 5.4: An error during the author list extraction.

incorrectly (Figure 5.4). Although Alice can provide more examples to learn the correct program, she finds it easier to switch to the Program Viewer tab, and select a correct alternative for the wrong subexpression (Figure 5.5).

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 is correct but the sub-program for the ending position seems too specific for the given example. Alice asks for alternative programs that the system has learned for the ending 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, correct program is

> "extract everything between first whitespace and first occurrence of Dot after CamelCase"

Note that "Wang" is considered to be in CamelCase by FlashProg, even though it is just one word. The logic is not obvious even for programmers.

Extracting Individual Authors 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. Although Alice can provide additional examples or look at the extraction program, she decides to engage in the Conversational Clarification mode, which helps FlashProg disambiguate between programs by answering clarifying

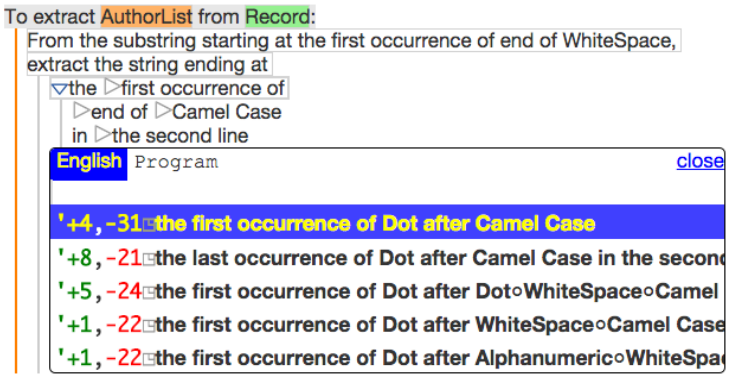


Figure 5.5: Program Viewer tab & alternative subexpressions.

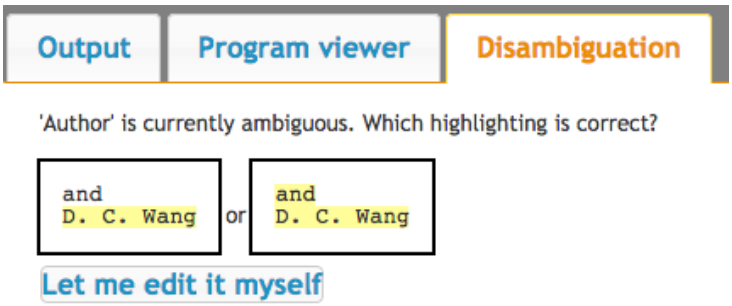


Figure 5.6: Conversational Clarification being used to disambiguate different programs that extract individual authors.

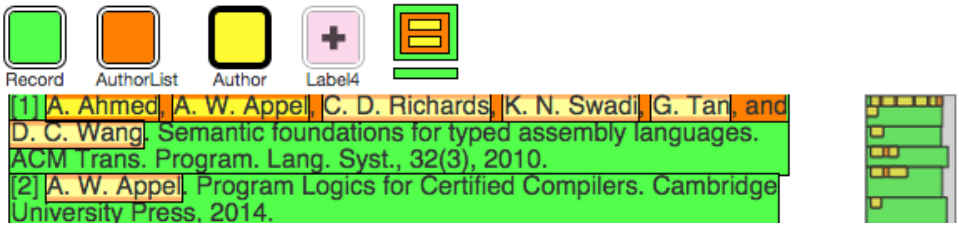


Figure 5.7: Final result of the bibliography extraction scenario.

questions (such as should the output include "`D. Richards`" or "`C. D. Richards`" and if "`and`" should be included, as shown in Figure 5.6). 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, which Alice is confident with (Figure 5.7).

5.1.3 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 is given by a sequence of positive and negative highlightings. The efficiency of the learning engine comes from the synthesis algorithm and the 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 given examples 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, our approach generates a *set* of satisfying programs, instead of just a single candidate. This enables our Program Navigation feature, where users may select alternate programs and observe their effects on the output.

Program Set Representation A typical learning session can return up to 10^{30} ambiguous programs, all consistent with the provided specification [Gulwani et al., 2012]. Our engine uses *version space algebra*, a polynomial-space representation of such a program set [Mitchell, 1982]. 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. Both p_1 and p_2 can expand to various position logics, such as absolute positions (*e.g.*, "5^{th} character") or positions based on regular expressions (*e.g.*, "after the second number"). 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 two program sets for p_1 and p_2 (also represented as VSAs). The join structure 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 (join) of two VSAs $\tilde{N}_1$ and $\tilde{N}_2$ represents a set that is a union (Cartesian product) of two sets represented by $\tilde{N}_1$ and $\tilde{N}_2$. VSA has two major benefits: (1) 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 key to our novel Program Navigation and Conversational Clarification interaction models, which we discuss next.

Program Navigation

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

Template Language To enable paraphrasing, we implemented a high-level template language, which maps *partial programs* into *partial English phrases*. Lau stated in one of her early PBE work [Lau, 2008]:

> "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 avoids arcane instructions by using "context-sensitive formatting rules", and avoids low-level instructions by using "idiomatic rules".

Paraphrasing is a bottom-up process. We use an idiom whenever possible. We remove context formatters from the template and apply them to their referenced child's template.

We illustrate this process with an S_1 program:

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

The program evaluates to the string between the start and end of the second line (note that line indices start at 0, whereas char indices 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, in which `{:0}` and `{:1}` refer to first and second arguments:

PosPair → "extract the string between {:0} and {:1}"

Pos → "the char number {:1} of {:0}"

Line → "line {:0}"

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 (lineNum) is a formatter mapping ints to a char ordinal (line ordinal). Formatters are lists of $\langle \text{regex}, \text{result} \rangle$ pairs that modify the template of the targeted child. We replace a formatter's template with the first matching regex result.

The limitation of this approach is that all rules are written and updated manually. We have to keep a DSL and its paraphrasing rules synchronized. Furthermore, because paraphrasing depends on order of formatters and idioms, some idiom templates may not allow users to explore the full program. We overcome this problem by letting the user switch between the paraphrase and the code (the latter is always complete).

Alternative Programs To enable exploring alternative programs, we record the original candidate program set for each subexpression. Because our program set 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. FlashProg then presents the top 5 subexpression programs to the user.

Conversational Clarification

Conversational Clarification selects candidates by differencing outputs produced by the synthesized 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 (k is set to 10 in FlashProg). We generate the clarifying question based on the *first discrepancy* between the outputs of the currently selected program P and those of the clarification candidateP'. Such a discrepancy can have three possible manifestations:

- The outputs of P and P' match until a region r in P does not intersect with any regions in P'. This leads to the question "Should r be highlighted or not?"
- The outputs of P and P' match until a region r' in P' does not intersect with any regions in P. This leads to the question "Should r' have been highlighted?"
- The outputs of P and P' match until a region r in P intersects with a region r' in P'. 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, which asks 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. We convert user choice into new examples, and invoke a new synthesis process.

Analysis Conversational Clarification always converges to the program representing user's intent in a finite number of rounds, if such program exists. The number of

rounds depends on the space of collisions in DSL outputs. However, the number of Conversational Clarification rounds is usually small in practice. In all of our benchmarks, this number never exceeds 5.

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), because we implement CC using our novel (recursively defined) *ranking* operation over the VSA [Polozov and Gulwani, 2015].

5.1.4 Evaluation

We now 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?

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.

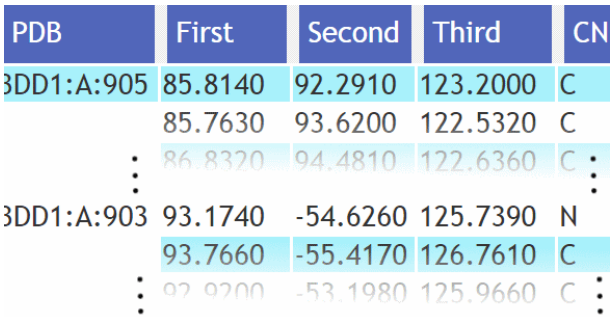
2. *Amazon research.* The text of the search results on Amazon for the query "chair". The data is visually structured as a list of records, but contains spacing and noise.

3. *Bioinformatic log.* A log of numerical values obtained from five experiments, from bioinformatics research (Figure 5.9).

We first provided users a brief video tutorial using the address file as example (Figure 5.1, `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 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 until 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 5.8). 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



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 5.8: Result sample of extracting bioinformatic log.

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 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.

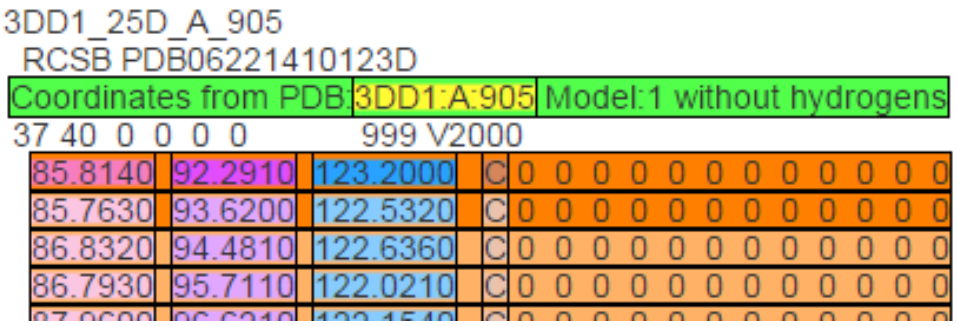


Figure 5.9: Highlighting for obtaining Figure 5.8.

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.

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.

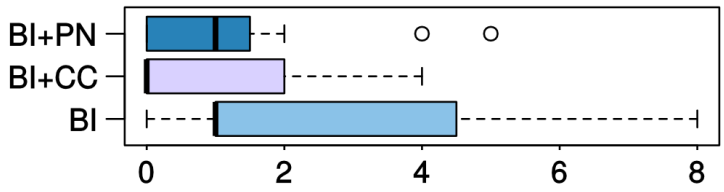


Figure 5.10: Distribution of error count across environments. Both Conversational Clarification (CC) and Program Navigation (PN) significantly decrease the number of errors.

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 5.10). 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 5.11a and Figure 5.11b). 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

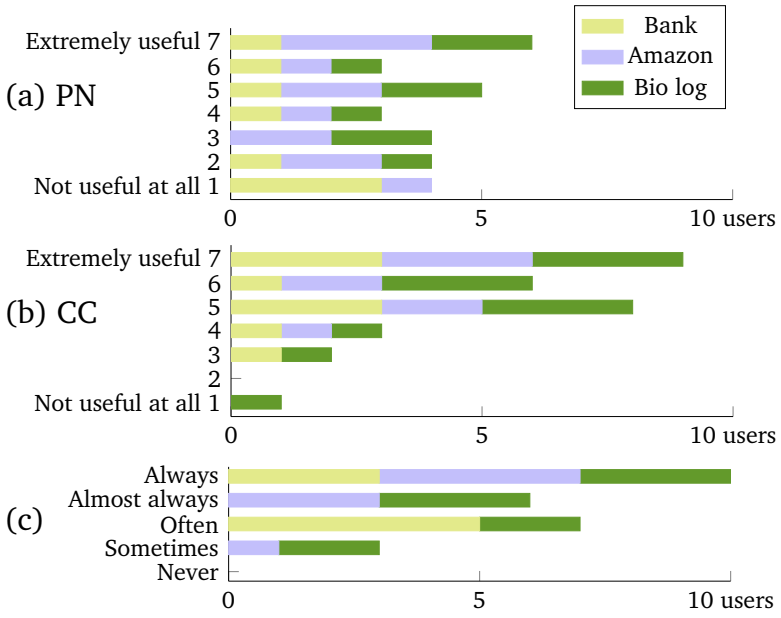


Figure 5.11: User-reported: (a) usefulness of PN, (b) usefulness of CC, (c) correctness of one of the choices of CC.

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.

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.*).

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 use Alternative subexpression viewer under 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.

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 several limitations 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).

5.2 New Techique: Guided Program Synthesis

Chapter 4 introduces Equivalence Modulo Inputs (EMI) as a promising approach for compiler validation. It is based on mutating the unexecuted statements of an existing program under some inputs to produce new equivalent test programs *w.r.t.* these inputs. Orion is a simple realization of EMI by only randomly deleting unexecuted statements. Despite its success in finding many bugs in production compilers, Orion's effectiveness is still limited by its simple, blind mutation strategy.

To more effectively realize EMI, this section introduces a guided, advanced mutation strategy based on Bayesian optimization. Our goal is to generate diverse programs to more thoroughly exercise compilers. We achieve this with two techniques: (1) the support of both code deletions and insertions in the unexecuted regions, leading to a much larger test program space; and (2) the use of an objective function that promotes control-flow-diverse programs for guiding Markov Chain Monte Carlo (MCMC) optimization to explore the search space.

Our technique helps discover *deep* bugs that require elaborate mutations. In 19 months, our realization, Athena, has found 72 new bugs — many of which are deep and important bugs — in GCC and LLVM. Developers have confirmed all 72 bugs and fixed 68 of them.

5.2.1 Illustrative Examples

This section uses two concrete bug examples to motivate and illustrate Athena: one LLVM bug and one GCC bug. Both bugs start from seed programs generated by Csmith [Yang et al., 2011], and trigger by a sequence of mutations, *i.e.*, inserting and deleting statements derived from the seeds.

LLVM Crashing Bug 18615 Figure 5.12a shows the *reduced* EMI variant that triggers the bug. Initially, `clang` compiles the seed program successfully at all optimization levels. However, after Athena replaces the original statement `f[0].f0 = b;` in the seed program with another statement `f[0] = f[b];`, it causes `clang` to crash at optimization `-O1` and above. The bug happens because an assertion is violated. LLVM assumes that array indices used in memory copy operations are non-negative. At line 9 of Figure 5.12a, the variable `b` is -1, making the array access `f[b]` illegal. However, this should not matter because the code is never executed. The compiler should not crash.

Athena extracts candidate statements for insertion from existing code and saves them into a database. Each database entry is a pair of statement and its required context, where the context specifies the necessary conditions to apply the statement. Figure 5.12b shows the database entry that was inserted into the seed program to reveal the bug. To use the statement of this entry, the insertion point must have an array of structs `g` and an integer `c` in scope.

While performing insertion, Athena only selects from the database those statements whose required contexts can be satisfied by the local context (at the insertion point) to avoid generating invalid programs. Athena also renames constructs in a selected database statement to match the local context when necessary. In this example, we can replace the original unexecuted statement with the statement in Figure 5.12b because their contexts are compatible under the renaming `g` $\rightarrow$ `f` and `c` $\rightarrow$ `b`.

```
1 int a;
2 struct S0 {
3   int f0; int f1; int f2;
4 };
5 void fn1 () {
6   int b = -1;
7   struct S0 f[1];
8   if (a) // true branch is unreachable
9     f[0] = f[b]; // was "f[0].f0 = b;"
10 }
11 int main () {
12     fn1 ();
13     return 0;
14 }
```

(a) The simplified EMI variant.

```
...
=====
// Required context
g: struct (int x int x int) [1]
c: int
-----
// Statement
g[0] = g[c];
=====
...
```

(b) The database entry used to insert into the variant. Athena renamed g, c to f, b to match the context at the insertion point.

Figure 5.12: LLVM 3.4 trunk crashes while compiling the variant at `-O1` and above (`https://llvm.org/bugs/show_bug.cgi?id=18615`).

Note that the program in Figure 5.12 is already reduced. The original program and its variant are quite large. Also, the bug was triggered not under one single step but under a sequence of mutations. Athena's MCMC algorithm plays a key role here. It guides the mutation process toward generating programs that are more likely to expose bugs. Orion cannot reveal the bug because it cannot insert new statements to trigger the assertion violation.

GCC Miscompilation Bug 61383 Figure 5.13a shows a simplified version of a GCC miscompilation bug. Because the loop on line 5 executes only once, the expected semantics of this program is to terminate normally. However, the compiled binary of the EMI variant aborts during its execution.

While compiling the variant, GCC identified the expression `1 % f` as a loop invariant for the loop on line 5. As an optimization, GCC hoisted the expression out of the loop to avoid redundant computation. However, this optimization is problematic because

```
1  int a, c, d, e = 1, f;
2  int fn1 () {
3    int h;
4    ...
5    for (; d < 1; d = e) {
6      h = f == 0 ? 0 : 1 % f;
7      if (f < 1)
8        c = 0;
9      else // else branch is unreachable
10       if(h) break; // was "c = 1;"
11   }
12   ...
13 }
14 int main () {
15   fn1 ();
16   return 0;
17 }
```

(a) The simplified EMI variant.

```
...
=====
// Required context
requires_loop
i: int
-----
// Statement
if (i)
  break;
=====
...
```

(b) The database entry used to insert into the variant. Athena renamed i to h. `requires_loop` requires the statement to be inserted inside a loop.

Figure 5.13: GCC trunk 4.10 and also 4.8 and 4.9 miscompile the variant at `-O2` and `-O3` (`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61383`).

now the expression is evaluated. Since `f` is 0, the expression `1 % f` traps and aborts the program. This expression should not be evaluated because the conditional expression always takes the "then" branch (`f == 0` is true).

By adding the extra statement `if (h) break;` from the database to line 10, Athena changes the control-flow of the variant. The extra complexity causes the optimization pass `ifcombine` to miss the check whether the expression `1 % f` can trap. Hence, it concludes that the statement does not have any side effect. The expression is incorrectly hoisted and the bug is triggered.

Because this bug-triggering statement contains a `break` statement, we can only insert it inside a loop. While traversing the source program, Athena keeps track of all variables in scope and an additional flag indicating whether the current location is inside the body

of a loop. On line 10, Athena renames `i` to one of the available variables, `h`, and replaces the unexecuted statement with `if (h) break;`.

5.2.2 Background on Markov Chain Monte Carlo

This section gives a gentle introduction to MCMC techniques. Most of the material comes from the following sources [Gilks, 1999, Andrieu et al., 2003]. Interested readers may consult them for further details.

Monte Carlo is a general method to draw samples X_i from a target density distribution $p(X)$ defined on a high-dimensional space $\mathcal{X}$ (such as the space of all possible configurations of a system, or the set of all possible solutions of a problem). From these samples, one can estimate the target density $p(X)$.

A stochastic process $\{X_0, X_1, X_2, \dots\}$ is a Markov chain if the next state X_{t+1} sampled from a distribution $q(X_{t+1} \mid X_t)$ only depends on the current state of the chain X_t. In other words, X_{t+1} does not depend on the history of the chain $\{X_0, X_1, \dots, X_{t-1}\}$.

Markov Chain Monte Carlo (MCMC) draws samples X_i in the space $\mathcal{X}$ using a carefully constructed Markov chain, which allows more samples to be drawn from important regions (*i.e.*, regions having higher densities). This nice property is obtained when the chain is *ergodic*, which holds if it is possible to transition from any state to any other state in the space $\mathcal{X}$. The samples X_i mimic samples drawn from the target distribution $p(X)$. Note that while we cannot sample directly on $p(X)$ (we are simulating this unknown distribution, which is why we use MCMC in the first place), we should be able to evaluate $p(X)$ up to a normalizing constant.

The Metropolis-Hasting algorithm is the most popular MCMC method. This algorithm samples the candidate state X^* from the current state X according to the proposal distribution $q(X^* \mid X)$. The chain accepts the candidate and moves to X^* with the acceptance probability as follows:

$$\mathcal{A}(X \to X^*) = \min\left(1, \frac{p(X^*)q(X \mid X^*)}{p(X)q(X^* \mid X)}\right) \tag{5.1}$$

Otherwise, the chain remains at X, and a new candidate state is proposed. The process continues until a specified computational budget is reached.

The Metropolis algorithm is a simple instance of the Metropolis-Hasting algorithm, which assumes that the proposal distribution is symmetric, *i.e.* $q(X^* \mid X) = q(X \mid X^*)$. Our acceptance probability simplifies to the following:

$$\mathcal{A}(X \to X^*) = \min\left(1, \frac{p(X^*)}{p(X)}\right) \tag{5.2}$$

While MCMC techniques can be used to solve many problems including integration, simulation and optimization [Gilks, 1999, Andrieu et al., 2003], our focus in this section is optimization.

5.2.3 MCMC Bug Finding

In our setting, the search space $\mathcal{X}$ is the space of all EMI variants of a seed program P. Because our computational budget is limited, we want to sample more "interesting" variants in this space $\mathcal{X}$. For this reason, we need to design an effective objective function that determines if a variant is interesting and worth exploring.

Objective Function

Our key insight for a suitable objective function is to *favor variants having different control- and data-flow* as the seed program. When compiling a program, compilers use various static analyses to determine — based on the program's control- and data-flow information — which optimizations are applicable. By generating variants with different control- and data-flow, we are likely to exercise the compilers more thoroughly by forcing them to use various optimization strategies on the variants. In particular, we use *program distance* to measure the difference between two programs.

Definition 5.2.1 *(Program Distance) The distance Δ between an EMI variant Q and its seed program P is a function of the distance between their control-flow graph (CFG) nodes (*i.e.*, basic blocks), the distance between their CFG edges, and their size difference. Specifically,*

$$\Delta(Q; P) = \alpha \cdot d(V_Q, V_P) + \beta \cdot d(E_Q, E_P) - \gamma \cdot |Q - P|$$

where

- $d(A, B) = 1 - \frac{A \cap B}{A \cup B}$ *is the Jaccard distance [Wikipedia, 2014];*

- *V_Q, V_P are Q and P's CFG node sets respectively;*
- *E_Q, E_P are Q and P's CFG edge sets respectively; and*
- *$|Q - P|$ is the program size difference of Q and P.*

Two nodes are different if their corresponding statements are different. If we modify a node in the variant, the node will be different from the original one. Two edges are different if their corresponding nodes are different.

Intuitively, our notion of program distance measures the *changes* in the variant. It is capable of capturing simple changes that do not alter the control flow, such as deleting and inserting straight-line statements, via the node distance. It also captures complex changes that modify the control and data flow considerably, such as deleting and inserting complicated statements, via both the node and edge distance.

Our program distance metric disfavors changes in program size. This helps avoid generating too small or too large variants. Small variants are less likely to reveal bugs. Large variants may take significant amount of time to compile, thus may prevent us from sampling many variants.

MCMC Sampling

When applied to optimization, an MCMC sampler draws samples more often from regions that have *higher* objective values. We leverage this crucial property to sample more often the program space that produces more different EMI variants (*i.e.*, ones with larger program distances Δ).

To calculate the transition acceptance probability, we need to evaluate the density distribution $p(.)$ at any step in the chain. According to [Gilks, 1999, Schkufza et al., 2013], we can transform any arbitrary objective function into a density distribution as follows

$$p(Q; P) = \frac{1}{Z} \exp\left(\sigma \cdot \Delta(Q; P)\right) \tag{5.3}$$

where σ is a constant and Z a normalizing partition function.

Algorithm 13: MCMC algorithm for testing compilers

```
1  procedure BugFinding(Compiler C, Seed test P, Input I):
2      O := C.Compile(P).Execute(I)                          /* ref. output */
3      Q := P                                                /* initialization */
4      for 1 .. MAX_ITER do
5          Q* := Mutate(Q, I)                                /* propose candidate */
6          O* := C.Compile(Q*).Execute(I)
7          if O* ≠ O then
8              ReportBug(C, Q*)
9          if Rand(0,1) < A(Q → Q*; P) then
10             Q := Q*                                       /* move to new state */
```

Deriving from (5.1), the probability to accept the proposal $Q \rightarrow Q^*$ is given below:

$$\begin{aligned}\mathcal{A}(Q \rightarrow Q^*; P) &= \min\left(1, \frac{p(Q^*;P)\cdot q(Q \mid Q^*)}{p(Q;P)\cdot q(Q^* \mid Q)}\right) \\ &= \min\left(1, \exp\left(\sigma\cdot(\Delta(Q^*;P) - \Delta(Q;P))\right)\cdot\frac{q(Q \mid Q^*)}{q(Q^* \mid Q)}\right) \end{aligned} \tag{5.4}$$

where $q(\cdot)$ is the proposal distribution, $q(Q \mid Q^*)$ is the probability of transforming Q^* to Q, and $q(Q^* \mid Q)$ is the probability of transforming Q to Q^*.

We develop our bug finding procedure based on Metropolis-Hasting algorithm. Algorithm 13 illustrates this procedure. We start with the seed program (line 3). The loop on lines 4-10 simulates our chain. At each iteration, based on the current variant Q, we propose a candidate Q^*, which we use to validate the compiler. The chain moves to the new state Q^* with the acceptance probability described in (5.4). Otherwise, it stays at the current state Q.

Variant Proposal

We generate a new candidate Q^* by removing existing statements from and inserting new statements into the current variant Q. All mutations must happen in the unexecuted regions under the profiling inputs I to maintain the EMI property.

Removing unexecuted statements is straightforward. We can safely remove any of these statements from the program without affecting its compilability. We only need to be careful not to remove any declaring statements whose declarations may be used later in the program.

However, inserting new statements into unexecuted regions is not as straightforward. In particular, we need to construct the set of all statements suitable for insertion. Also, while inserting new statements, we need to take extra care to make sure the new variants compile.

Extracting Statement Candidates We extract statement candidates from *existing code*. Given a corpus of existing programs, we construct the database of all available statements by traversing the ASTs of these programs and extract all statements at all levels. These statements have different complexities, ranging from a single-line statement to a whole function body.

For each statement, we determine the *context* required to insert the statement. When we perform insertion into a location, we only insert statements whose required contexts can be satisfied by the local context at the location. This is to guarantee that the new variant compiles after insertion.

Generating New Variant We use our statement database to facilitate variant mutation. At an unexecuted statement, we can perform the following actions according to the proposal distribution $q(\cdot)$:

Delete Similar to Orion, we keep track of two probabilities p^d_{parent} and p^d_{leaf}. We delete a parent statement, those that contain other statements, with probability p^d_{parent}. We delete a leaf statement with probability p^d_{leaf}. We distinguish these two kinds of statements because they have different effects on the new variant.

Insert We also have two probabilities p^i_{parent} and p^i_{leaf} for inserting at parent or leaf statements. We can insert a new statement either before or after the unexecuted statement (with the same probability). If the unexecuted statement does not directly belong to a compound statement, we promote it to a compound statement before inserting the new statement. This is to make these statements share the same parent.

We perform a breadth-first traversal on the AST of the current variant Q. During this traversal, we maintain a *context table* that contains necessary information (such as the variables in scope) to help select compatible statements from the database. At each statement marked as unexecuted, we delete the statement or insert a new statement according to the probabilities defined above. It is possible to do both, in which case the unexecuted statement is replaced by the new statement.

Maintaining Ergodicity Our mutation must satisfy the ergodicity property described in Section 5.2.2, which states that we should be able to walk from one state to any other state in our search space. If this property is violated, we cannot perform the walk effectively because the search space is disconnected.

Let us first see if we can revert our proposal Q^* to Q. We can easily revert an inserted statement by deleting it. However, it is impossible to revert a deleted statement. As the statement is removed, our coverage tool cannot mark the location as unexecuted, which is the necessary condition for insertion. Moreover, if the deleted statement does not exist in our database, our insertion process will not be able to recover this statement.

To address the first problem, we leave a syntactic marker in place of the deleted statement. While mutating the variant, we replace these markers with new statements as if they are unexecuted statements. In our implementation Athena, we use the statement "`while(0);`". Comments do not work because our tool only visits statements. Although these markers do not have any effects on the variant semantics, they may affect the compilers under test. Hence, we remove them from the variant before testing.

To solve the second problem, we allow code from the seed program. Before sampling, we extract statements from the seed and add them to the statement database. Because the deleted statement is either from the seed program or external code, we will be able to reconstruct it from the updated database.

Our process is now ergodic, and hence applicable for sampling. We can transform any EMI variant of a program to any other EMI variant of that program.

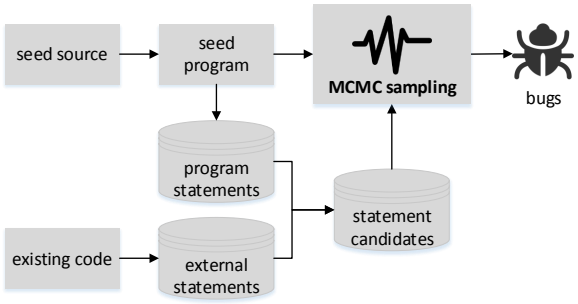


Figure 5.14: The high-level architecture of Athena. We extract statement candidates from both existing code and the seed program. We then perform MCMC sampling on the seed program to expose compiler bugs.

5.2.4 Implementation

Athena is a concrete realization of our MCMC bug finding procedure targeting C compilers. We highlight its architecture in Figure 5.14. This section discusses various design choices we made in implementing Athena.

Extracting Statement Candidates

Athena uses the tool `stmt-extractor` to extract statement candidates from existing code corpus. We implement `stmt-extractor` using LLVM's Libtooling library [The Clang Team, 2014]. For each program in the corpus, `stmt-extractor` visits all statements at all levels in the program's AST. It then determines the context required to apply such statements and inserts these ⟨context, statement⟩ pairs to the database.

A context contains the following information:

- The variables (including their types) used in the statement but defined outside. These variables must be in scope at the insertion point when we perform insertion (otherwise we would have inserted an invalid statement). We exclude variables that are defined and used locally.
- The functions (including their signatures) used in the statement.
- The user-defined types used in the statement such as structures or unions, and their dependent types.

- The (`goto`) labels *defined* in the statement. If a function has already defined these labels, we have to rename the labels in the inserted statement.
- The labels *used* in the statement. We have to rename these labels to match those defined in the function.
- A flag indicating whether the statement contains a `break` or `continue` statement. If this is the case, we can only insert this statement inside a loop.

To construct the context, we parse the statement and collect the information listed above. In particular, we find the required variables by finding all variables used in the statement, and subtracting them with those that are defined inside the statement. During this process, we also collect all used functions, user-defined types, labels, and break and continue statements. If the statement uses any user-defined type, we recurse into the type and construct its dependencies. Type dependencies are cached to avoid redundant computation.

In Figure 5.14, we only need to calculate the database of candidate statements *once* and the process happens *offline*. We also use `stmt-extractor` to extract statements from the seed program.

Proposing Variants

We implement a tool called `transformer` to transform variants, also using LLVM Libtooling. The tool takes as input a program (which is either the seed or one of its EMI variants in the chain), the program's coverage information (which is obtained using GNU gcov [GNU Compiler Collection, 2015]), and the four deletion/insertion probabilities mentioned in Section 10.

Transformation The `transformer` tool performs a breadth-first traversal on the program. It keeps a context table that stores variables and labels in scope, the available functions and user-defined types, and a flag indicating whether the current statement is in a loop. It deletes unexecuted statement or inserting new ones according to the given probabilities. If a statement is deleted, we will stop traversing into it. We do not traverse into newly inserted statements.

While it is possible to keep the deletion/insertion probabilities unchanged during the mutation, it is better to randomize them in our experience. Hence, we shuffle these probabilities after each deletion or insertion.

Statement Renaming Because external code has different semantics and naming convention, there are usually not many statement candidates compatible with the local program context at insertion points. For this reason, we relax the context matching condition to accept matches under *renaming*. In particular, we allow the renaming of variables in the statement candidates to those in the local context that have compatible types. For example, if the context of a statement candidate requires an integer variable b, but the local context only has an integer variable a, we can rename all occurrences of b in the candidate to a. Similarly, we can rename labels, functions and user-defined types to those that have compatible signatures.

Discussion

Although our algorithm satisfies the ergodicity property, it is not symmetric. That is, we cannot walk back to Q^* from Q in one step in general (but we may via several steps). This is because a deleted statement from Q may be constructed from several steps, and does not exist in our database. For example, we may insert a large statement s into Q_k, and since s is also unexecuted, some of its children are deleted in Q_{k+1} (let us call the updated statement s'). If we delete s' in Q_{k+2}, we cannot go back to Q_{k+1} in a single step because the database does not have s'.

One possible solution is to extract the statements from every variant, and update the database after each transition. We have decided not to do this because many statements are redundant. It is quite challenging to distinguish these statements since many of them have been renamed to match the local context.

As a result, our proposal distribution is not symmetric. To simplify our implementation, we assume that this distribution is symmetric (*i.e.*, it is equally likely to generate the new variant Q^* from the current variant Q and vice versa). The consequence of this assumption is that we may not sample the program space proportionally to the value of the objective function. For instance, we may not sample as often (or more often than

required) the program space that has high objective values.

This is an example of the trade-offs between precision and efficiency. Making our process more precise may incur major performance degradation. On the other hand, having a less precise process helps sample many more variants in the same unit of time. This justifies the lack of sampling precision.

Based on this assumption, our algorithm turns into Metropolis algorithm, which has the following simpler acceptance probability:

$$\mathcal{A}(Q \to Q^*; P) = \min\Big(1, \exp\Big(\sigma \cdot (\Delta(Q^*; P) - \Delta(Q; P))\Big)\Big) \tag{5.5}$$

5.2.5 Evaluation

We focus our testing efforts on two open-source compilers GCC and LLVM because of their openness in tracking and fixing bugs. We summarize below our results from the end of January 2014 to the end of August 2015:

- *Many detected bugs:* In 19 months, Athena has revealed 72 new bugs. Developers confirmed *all* of our bugs and fixed nearly all of them (68 out of 72).

- *Many serious bugs:* GCC developers marked 17 out of 40 GCC bugs as P1, the most severe kind of bugs that must be fixed before a new GCC release can be made. Three of our GCC bugs were linked to subsequent bugs exposed while compiling real-world code, namely `gcc`, `qtwebkit`, and `glibc`.

- *Many deep bugs:* Our experiments show that Athena is capable of detecting both shallow and deep bugs. The later requires sophisticated mutation sequences that could not be done using Orion.

- *Many long-latent bugs:* Although our focus is to test the development trunks of GCC and LLVM, we have found 15 latent bugs in old versions of the two compilers. These bugs had resisted traditional validation approaches, which further illustrates Athena's effectiveness.

Testing Setup

We first describe our testing setup before presenting our detailed results.

Sources of Seed Programs Athena is capable of using existing open-source projects as seed programs. However, it is challenging to reduce bugs triggered by these projects because the projects typically involve multiple directories and multiple files [Le et al., 2014]. Therefore, in our evaluation, we only use programs generated from the random program generator Csmith [Yang et al., 2011]. Existing reduction tools such as Berkeley's Delta [McPeak et al., 2015] and CReduce [Regehr et al., 2012] can reduce these programs effectively.

Sources of Statement Candidates Athena is capable of using existing code for insertion. As part of our evaluation, we built a database of all statements extracted from the SPEC CINT2006 benchmarks [Standard Performance Evaluation Corporation, 2015]. Because the database contains a huge number of statements, it is very expensive to perform a linear scan on it. Note that we cannot look up by required context because a local context is different from these required contexts, and it may satisfy not one but multiple of them. Our solution is to repeatedly draw a random statement from the database until we find one that satisfies the local context. If we cannot find any satisfying statement after some constant k attempts, we conclude that no satisfying candidate exists.

Unfortunately, our experiments show that inserting real-world code into Csmith-generated seeds is ineffective. This is because Csmith can only generate limited forms of constructs, making the contexts at insertion points incapable of accepting the more diverse real-world code. One way to mitigate this problem is to merge the required constructs from external projects to the current variant. This is quite challenging because these constructs may depend on other constructs, or locate in a different location. Therefore we leave this for future work.

The seed program turns out to be a great source for statement candidates. A seed program can yield hundreds of statements that have diverse complexities: statements range from one line to hundred lines of code. Moreover, these statements are well connected to the variants, which helps increasing the ratio of satisfying statements. Our evaluation uses only statements from the seed programs.

Selecting Statement Candidates For each unexecuted location, there are potentially multiple satisfying statements from the database. A good strategy to select the "best" statement from this satisfying set may yield a better result overall.

We hypothesize that the best statement is one that uses the most information from the local context. For instance, it uses the most number of variables defined in the variant. Using this statement puts more constraints on the compiler because it increases the dependencies between existing code and external code.

Under this strategy, we have to scan the database to find the best statement candidate. Our experiments show that this strategy is two orders of magnitude slower than the random sampling strategy. Because speed is key in testing, we adopt the random sampling strategy and use it in our evaluation.

Testing Infrastructure Our testing focuses on the x86-linux platform due to its popularity and ease of access. We conduct our experiments on two machines (one 18 cores and one 6 cores) running Ubuntu 12.04 (x86_64). While calculating the program distance Δ, we value equally changes in CFG nodes and edges ($\alpha = \beta = 0.5$). We fine-tuned γ to avoid generating programs larger than 500KB, a threshold at which we observed a significant degradation in compilation time for both GCC and LLVM.

As in the work on Csmith and Orion, we have focused on the five standard options, "-O0", "-O1", "-Os", "-O2" and "-O3", because they are the most commonly used. Athena tests only the daily-built development trunks of GCC and LLVM. Once it finds a bug in a compiler, Athena also validates the bug against other major releases of that compiler.

Quantitative Results

We next present some general statistics on our reported bugs.

Bug Count Table 5.1 summarizes our bug results. In 19 months, we reported 83 bugs, which are roughly equally divided between GCC (44 bugs) and LLVM(39 bugs). Developers confirmed 72 valid bugs and fixed 68 of them.

Not-Yet-Fixed Bugs Among two GCC bugs that have not been fixed, one was just reported recently. The other one (bug 62016) is a performance bug, which affects GCC

	GCC	LLVM	TOTAL
Fixed	38	30	68
Not-Yet-Fixed	2	2	4
WorksForMe	0	3	3
Duplicate	3	4	7
Invalid	1	0	1
TOTAL	44	39	83

Table 5.1: Reported bugs.

4.8.X, 4.9.X, and 4.10 trunk (at that time). GCC took a few minutes to compile a small program due to problems in inlining. Developers discussed about backporting some code across versions to fix the problem. Unfortunately, this was quite challenging because of a cross-version design gap. A few months later, GCC moved to version 5.0 and some design changes fixed the problem. The bug therefore remains unfixed, although it still affects earlier versions. One of the two not-yet-fixed LLVM bugs is a complicated one. Developers have not found a solution to fix it yet. The other bug triggers only in debug mode, which perhaps is the reason why developers have not fixed it.

WorksForMe Bugs It may take a while before developers consider our reported bugs. During this time, the trunk has changed and it is possible that these changes suppress the bug. Developers mark bugs that no longer trigger "WorksForMe". We have three LLVM bugs of this kind. We do not have this kind of bugs in GCC because GCC developers responded to our bugs very quickly. Moreover, even when this happens, GCC's policy recommends going back to the affected revision and check if the root cause has been properly fixed. If not, the bug may be latent and is likely to trigger later. Indeed, one LLVM WorksForMe bug (bug 21741) re-triggers in a later revision. We reopened the bug, but LLVM developers have not yet responded.

Duplicate Bugs Before reporting a bug, we ensure that it has different symptoms from the previously reported and not yet fixed bugs. However, reporting duplicated bugs may be unavoidable because compilers are complex. Bugs having different symptoms may

turn out to have the same root cause. During our evaluation, we reported 7 duplicated bugs (3 GCC and 4 LLVM).

As an example, GCC bugs 64990 and 645383 are duplicates of bug 61047. Bug 61047 only triggers at `-O1` on GCC 4.9 and 4.10 trunk. Bug 64990 triggers at all optimization levels on all versions of GCC from 4.6 to 5.0 trunk. Bug 65383 only triggers at `-O2` and `-O3` from GCC 4.7 to trunk, and the program looks very different from bug 64990. Although these bugs affected different versions and triggered at different optimization levels, they turned out to have the same root cause. Developers marked two bugs reported later as duplicates.

Invalid Bugs We reported one invalid bug (bug 63774) for GCC, in which a function returns the address of a local variable. We were unsure whether this behavior was implementation defined or undefined. It turned out that the behavior was undefined, and the bug was subsequently marked as invalid.

Bug Type We classify bugs into two main categories: (1) ones that manifest when we compile programs, and (2) ones that manifest when we execute the compiled binaries. A compile-time bug can be a *crashing bug* (*e.g.*, internal compiler errors), or *performance bug* (*e.g.*, the compiler hangs or takes a very long time to compile the program). A runtime bug happens when the compiled program behaves abnormally *w.r.t.* its expected behavior. For example, it may crash, hang, or produce wrong output. We call these bugs *miscompilation* bugs.

Table 5.2 classifies our 72 confirmed and valid bugs according to the above taxonomy. These bugs are quite diverse, illustrating the power of Athena in finding all kinds of bugs. A significant chunk of these bugs are miscompilation, the most serious kind among the three.

Importance of Reported Bugs Developers took our bugs seriously. They have confirmed all of our bugs and fixed nearly all of them. GCC developers are generally more responsive in fixing bugs — they fixed most of our reported bugs within several days.

Three of our GCC bugs were linked to bugs triggered while compiling real-world projects. Bug 63835 is related to a bug that crashes GCC while compiling GCC itself.

	GCC	LLVM	TOTAL
Miscompilation	11	17	28
Crash	26	13	39
Performance	3	2	5

Table 5.2: Bug classification.

Bugs 61042 and 61684 crash GCC while compiling our variants. Later, people reported similar bugs while compiling `qtwebkit` and `glibc`. Subsequently, these bugs were marked as duplicates. Such bugs are rare because developers usually fix our bugs very quickly, leaving only a small time window for people to rediscover these bugs using real-world projects.

Another way to measure the importance of our bugs is via the "Importance" field set by developers in the bug reports. Developers marked 17 of our 40 GCC bugs as "P1", the highest bug priority (the default is "P3"). Developers must fix all P1 bugs before they can release a new version of GCC. LLVM developers marked all our bugs using the default value "P normal". We do not know whether these bugs have normal severity or LLVM developers did not classify the bugs.

Effectiveness of MCMC Bug Finding

Athena is effective because it is able to find many bugs after developers have fixed numerous bugs reported by Csmith and Orion. However, it is unclear how many of these bugs are deep bugs and could not be found by Orion. We conduct another experiment to compare Athena directly with Orion to answer this question.

We run Athena and Orion in parallel for a certain amount of time using the same seed programs that trigger our bugs. If Athena finds a bug, we reset the chain to the seed program and continue. This does not apply to Orion because it always operates on the seed program.

We limit this experiment to bugs that affect previous stable releases. It is because if the bug only affects the trunk, we need to build the compiler at that revision. Since the number of bugs is large, it is quite expensive to build a revision for each of them.

Bug ID (1)	Type (2)	Optimizations (3)	Seed (4)	Variant (5)	Report (6)	DB (7)	#Bugs (8)	#Variants (9)
gcc-59903	Crash	-O3	4,694	6,238	38	1,723	14	23,479
gcc-60116	Mis.	-Os	11,596	11,843	25	3,092	367	20,082
gcc-60382	Crash	-O3	6,151	21,903	19	1,989	19	21,267
gcc-61383	Mis.	-O2, -O3	3,298	3,567	22	1,272	106	32,981
gcc-61452	Hang	-O1, -Os	3,308	3,474	17	885	0	49,158
gcc-61917	Crash	-O3	11,820	11,226	7	3,066	2	32,562
gcc-64495	Crash	-O3	2,767	1,951	20	517	4	45,896
gcc-64663	Crash	all	11,118	12,160	9	2,875	0	26,626
llvm-20494*	Mis.	-O2, -O3	8,080	11,009	23	1,683	2,660	24,588
llvm-20680	Mis.	-O3	6,250	7,584	15	1,753	22	23,438
llvm-21512*	Crash	all	8,455	5,087	11	3,081	988	21,882
llvm-22086	Crash	-Os, -O2, -O3	5,220	8,495	27	1,711	0	29,279
llvm-22338	Crash	-O2, -O3	2,923	7,197	23	1,302	13	19,469
llvm-22382	Crash	-Os, -O2, -O3	4,813	2,147	19	1,432	0	29,805
llvm-22704	Crash	all	3,684	23,250	11	981	12	28,740

Table 5.3: The result of running Athena and Orion on the bugs that affect stable releases for one week. Columns (4), (5) and (6) show the SLOC (in BSD/Allman style) of the original seed program, the bug-triggering variant, the reduced file used to report the bug. Column (7) shows the size of the database (*i.e.*, the number of <context, statement> pairs) constructed from the seed program. Column (8) shows the number of bugs Athena rediscovered. Column (9) shows the total number of variants Athena generated. Orion rediscovered two *shallow* bugs: LLVM 20494 and 21512.

Table 5.3 shows the results of running Athena and Orion in parallel on 15 of such bugs for one week. In the table, we assume that all bugs rediscovered under a same seed file are the same. From our experience with both Orion and Athena, it is unlikely for the same seed Csmith program to reveal multiple bugs in the same compiler revision.

Shallow Bugs During this period, Orion was able to discover only two LLVM bugs (20494 and 21512). Athena also rediscovered these bugs. Interestingly, these bugs were rediscovered most *often* (2,660 and 988 times), which indicates that they are shallow, and can be triggered using simple mutations. Orion failed to find bugs in the other seed programs. These bugs are deep bugs, and require sophisticated sequences of mutations.

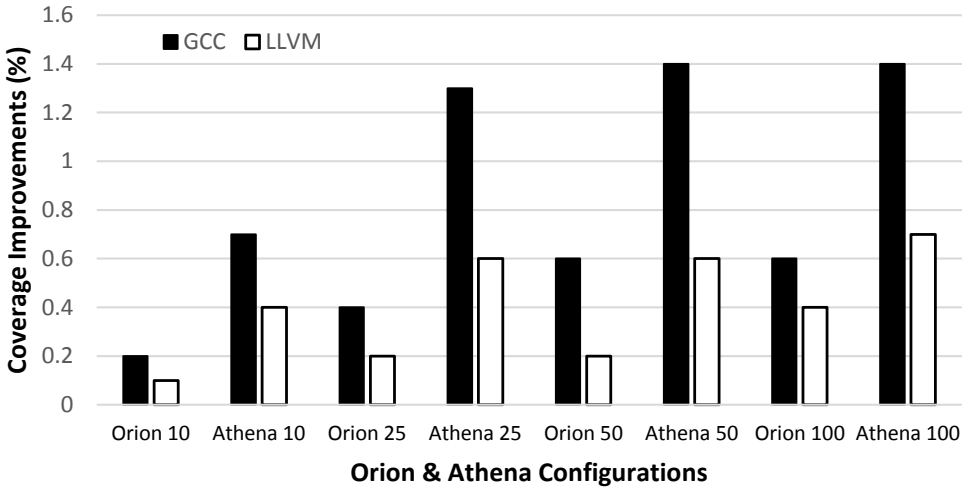


Figure 5.15: Improvement in line coverage of Orion and Athena while increasing the number of variants. The baseline is the coverage of executing 100 Csmith seeds, where GCC and LLVM have respectively 34.9% and 23.5% coverage ratios.

Deep Bugs Despite generating 28K variants on average for a seed program, Athena was unable to rediscover four of our bugs. Some other bugs were rediscovered only a few times. These bugs are indeed quite deep and require specific sequences of mutations. This is understandable because the search space is vast and our process is nondeterministic.

The sizes of the triggering variants vary. The variants are often larger than the original programs. Some are significantly larger because we may happen to insert some large chunks of code (such as bugs gcc-60382 and llvm-22704). On the other hand, some are smaller because we delete some large chunks of code (such as bugs gcc-64495 and llvm-22382).

This experiment confirms that Athena is more powerful than Orion in terms of bug detection. Indeed, Orion is Athena taking away insertion and limiting the length of random walks to one. However, Athena may take more time than Orion to find bugs. If the bug is shallow, Orion may find it faster because it only explores the smaller nearby neighborhood of a seed program.

Coverage Improvement

We now evaluate the line coverage improvement of Athena on GCC and LLVM in comparison with Orion. The baseline is the coverage of executing 100 Csmith seed programs,

on which GCC and LLVM achieve 34.9% and 23.5% coverage ratios respectively. We measure coverage using variants produced by Orion and Athena from these seeds. To evaluate the impact of the number of variants on code coverage, we also vary the number of variants for each seed.

Figure 5.15 shows the coverage improvement of Orion and Athena over the baseline. Although both Athena and Orion help increase line coverage, Athena is strictly better than Orion. In particular, 10 Athena variants yields slightly better coverage than 100 Orion variants. This is expected because Athena generate more diverse test programs.

5.3 New Domain: Stress-Testing Link-Time Optimizers

Link-time optimization (LTO) is an increasingly important and adopted modern optimization technology. It is currently supported by many production compilers, including GCC, LLVM, and Microsoft Visual C/C++. Despite its complexity, but because it is more recent, LTO is relatively less tested compared to the more mature, traditional optimizations. To evaluate and help improve the quality of LTO, we present the first extensive effort to stress-test the LTO components of GCC and LLVM, the two most widely-used production C compilers. In 11 months, we have discovered and reported 37 bugs (12 in GCC; 25 in LLVM). Developers have confirmed 21 of our bugs, and fixed 11 of them.

Our core technique is differential testing and realized in the tool Proteus. We leverage existing compiler testing tools (Csmith and Orion) to generate *single-file* test programs and address *two important challenges* specific for LTO testing. First, to thoroughly exercise LTO, Proteus automatically transforms a single-file program into multiple compilation units and stochastically assigns each an optimization level. Second, for effective bug reporting, we develop a practical mechanism to reduce LTO bugs involving multiple files.

5.3.1 Illustrative Examples

Proteus detects both crash/hang and miscompilation bugs in compilers' LTO components. Unlike traditional optimizations which are performed during compilation, LTO takes place at link time. It further complicates compilers, as they need to write intermediate representations to object files, read them back in, and perform whole-program analyses.

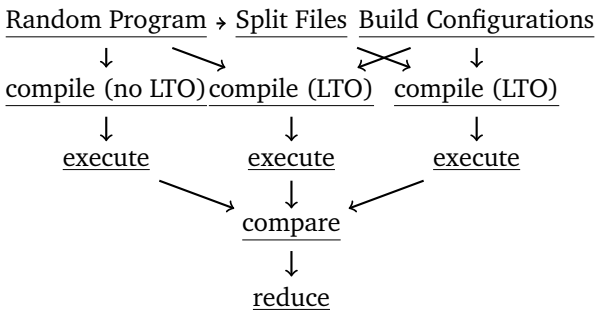


Figure 5.16: Overview of Proteus's approach.

Figure 5.16 shows the overview of Proteus's approach. It starts with a single-file program p (generated by Csmith or Orion), and compiles p in three different ways:

1. p is directly compiled without LTO.
2. p is compiled with LTO under various compilation and linker flags.
3. p is split into multiple compilation units (each corresponds to a function), which are separately compiled under different optimization flags and linked with LTO.

Proteus then executes these compiled programs and compares the execution results. Any inconsistency indicates a bug. We next demonstrate this process via two concrete bugs, one for GCC and one for LLVM.

GCC Bug #60404 Figure 5.17 shows Proteus's steps to find this bug. The original program, generated by Orion, prints 0 — the expected value of `a[b]` — and terminates (Figure 5.17a). Note that after the call `fn2(0)` in the function `main`, the value of variable `b` remains unchanged (*i.e.*, 0).

However, the GCC development trunk (revision 208268) miscompiles the files that are split from the original file by Proteus (Figure 5.17c), under the build configuration shown in Figure 5.17b. The compiled program in this case prints 1 instead of the expected 0. In this program, `b` was incorrectly assigned the value 1 after invoking `fn2(0)`. The printed value `a[b]` (or `a[1]`) is the memory location right after the boundary of the array `a`, which is coincidentally `b` (or 1). Figure 5.17d shows the files and the build configuration that we used for reporting after some cleaning up.

```
/*** small.c ***/
#include <stdio.h>
int a[1] = { 0 }, b = 0;
void fn1 (int p) { }
void fn2 (int p) {
  b = p++;
  fn1 (p);
}
int main () {
  fn2 (0);
  printf ("%d\n", a[b]);
  return 0;
}
```

(a) A simplified version of a program generated by Orion (the original version has 2367 lines of code). All compilers under test compile it correctly.

```
/*** configuration ***/
gcc -flto -O1 -c fn1.c
gcc -flto -O1 -c fn2.c
gcc -flto -O1 -c main.c
gcc -flto -O1 -c t.c
gcc -flto -O0 fn1.o fn2.o \
  main.o t.o
```

(b) A build configuration that triggers a GCC LTO bug on split files in Figure 5.17c.

```
/*** small.h ***/
#include <stdio.h>
int a[1], b;
void fn1 (int p);
void fn2 (int p);
/*** small.c ***/
#include "small.h"
int a[1] = {0}, b = 0;

/*** fn1.c ***/
#include "small.h"
void fn1 (int p) { }

/*** fn2.c ***/
#include "small.h"
void fn2 (int p) {
  b = p++;
  fn1 (p);
}

/*** main.c ***/
#include "small.h"
int main () {
  fn2 (0);
  printf ("%d\n", a[b]);
  return 0;
}
```

(c) Files split from the code in Figure 5.17a.

```
/*** fn1.c ***/
void fn1 (int p) { }

/*** fn2.c ***/
extern int b;
extern void fn1 (int);
void fn2 (int p) {
  b = p++;
  fn1 (p);
}

/*** main.c ***/
extern void fn2 (int);
int a[1], b;
int main () {
  fn2 (0);
  printf("%d\n", a[b]);
  return 0;
}

/*** configuration ***/
gcc -flto -O1 -c fn1.c
gcc -flto -O1 -c fn2.c
gcc -flto -O1 -c main.c
gcc -flto -O0 fn1.o fn2.o \
  main.o
```

(d) Cleaned up files with the bug-triggering configuration for bug reporting.

Figure 5.17: Proteus's workflow: from original file (in Figure (a)) to split files (in Figure (c)) to reported files (in Figure (d)). GCC revision 208268 miscompiles these files. The compiled program returns 1 instead of 0. (`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=60404`)

The bug happens because while coalescing SSA (single static assignment) parameter variables, the GCC developer mistakenly assumes that intermediate representation (IR) in object files are compiled without optimization, or without LTO. When the assumption is invalid, GCC misinterprets the IR stored in object files and generates incorrect code.

In this example, GCC compiles `fn2.c` at `-O1` to produce `fn2.o`, whose *optimized* IR is similar to the following:

```
void fn2 (int p) { p_1 = p + 1; b = p; fn1 (p_1); }
```

When GCC links the program with `-flto -O0`, it assumes that the IR of `fn2.c` is unoptimized and that all the SSA variables originated from the parameter `p` are already coalesced into a single partition. As the assumption is wrong, GCC fails to allocate a memory location for `p_1`, and consequently both `p` and `p_1` share the same address. The miscompiled code is similar to the following:

```
void fn2 (int p) { p = p + 1; b = p; fn1 (p); }
```

GCC correctly compiles the original single-file program and its split files without LTO because in these cases, the complication involving writing/reading/linking multiple IRs do not arise.

LLVM Bug #19201 Figure 5.18 shows an LLVM LTO bug, triggered by a test program generated by Orion. The test program is clearly well-defined according to the C standard. It should execute and terminate normally. However, both LLVM 3.4 and its development trunk miscompile the code with LTO enabled at `-O0`, resulting in a non-terminating program. With LTO disabled the program is correctly compiled.

It is evidently a bug in LLVM, because the program is valid and the semantics of the program compiled with LTO is inconsistent with that of its non-LTO counterpart. The LLVM developers have yet to comment on the root cause of the bug. To shed some light on this bug, we have disassembled the miscompiled program and inspected its assembly code. LLVM compiles the program into an infinite loop with an empty body:

```
int main() {  while(1) {}; return 0; }
```

```
/*** small.c ***/
int a, b = 1, c;
int fn1 (unsigned char p1, int p2) {
  return p2 || p1 > 1 ? 0 : p2;
}
int main () {
  int d = 0;
  for (; a < 1; a++) {
      c = 1;
      fn1 ((b &= c) | 10L, d);
  }
  return 0;
}
/*** configuration ***/
clang -flto -O0 -c small.c -o small.o
clang -flto -O0 small.o
```

Figure 5.18: LLVM 3.4 and trunk revision 204228 miscompile this program at -O0. The compiled executable hangs. (`http://llvm.org/bugs/show_bug.cgi?id=19201`)

We suspect that LLVM mistakenly concludes that the program contains undefined behavior, and consequently generates a non-terminating loop. This happens because there is no restriction on compilers for compiling programs having undefined behaviors. Compilers are only obligated to consider valid programs.

5.3.2 Design and Realization

This section describes our approach and realization of Proteus. At the high level, Proteus first leverages Csmith and Orion to generate single-file test programs to enable later phases of our differential testing of LTO. In particular, we use the generated programs to seed the following two-step process: (1) we modify Orion to insert arbitrary function calls to unexecuted code regions to increase function-level dependencies; and (2) we divide a single test program into separate compilation units. Both steps are semantics-preserving and designed to specifically target LTO testing. Our goal is to find build configurations that lead to deviant behavior.

Definition 5.3.1 (Split Function) *The function Split takes as input a single program and divides it into the followings: (1) a header file that contains all type definitions and global variable/function declarations; (2) an initialization source file that contains all initializations of global variables (this file includes the header file); (3) a set of source files, each of which contains one function definition from the original source file (these files also include the header file).*

For example, Split divides a single file in Figure 5.17a to a header file, an initialization file, and a set of function files in Figure 5.17c. The transformation imposed by Split does not change the semantics of the original program.

Definition 5.3.2 (Build Configuration) *A build configuration specifies how to compile and link a set of source files into a single executable. It compiles each source file into an object file and links all the object files into a final executable. Each compilation and linking step is parametrized over a set of optimization flags.*

For instance, Figure 5.17b is a build configuration that triggers a bug in GCC while compiling the files in Figure 5.17c.

Differential Testing of LTO

Proteus uses differential testing to find LTO bugs. The traditional view of differential testing [McKeeman, 1998] is quite simple: If two systems under test behave differently on some input, it indicates a bug in one of the systems, or both. Csmith has implemented this view [Yang et al., 2011]. It generates random C programs and seeks for deviant behavior in different C compilers while compiling/running the same source program.

Orion introduces an alternate view on differential testing [Le et al., 2014]. It profiles the execution of a program P under some input I. It then generates many variants of P by randomly pruning unexecuted statements in P. Since these variants are equivalent *w.r.t.* I (*i.e.*, they produce the same output under the input I), Orion seeks for deviant behavior in a compiler while compiling/running these variants on I.

We take yet another view of differential testing, based on the observation that all compiled programs built from different build configurations are semantically equivalent.

Proteus seeks for deviant behavior in a compiler while compiling/running the program (and its split version) under different LTO build configurations. Similar to the view in Orion, this view has some advantages over the original (and Csmith's) approach. Proteus can operate on existing code base (either real or randomly generated), and it can validate a single compiler in isolation (where competing compilers do not exist).

Implementation

Algorithm 14 describes the main procedure for finding LTO bugs from a test program in Proteus. It takes as input a compiler under test $Comp$, a program P and a set of its input I, and searches for build configurations that trigger LTO bugs on P (or its split files) under some input in I.

The algorithm consists of two main steps. First, Proteus calculates the expected output of the original program built without LTO (lines 3–4). It then searches for inconsistent behaviors in the compiler when LTO is enabled while compiling/running the program and its split files (lines 5–6). The loop on line 9 randomly generates build configurations (line 10) and checks for any deviance from the expected behavior (lines 12–17). Proteus reports a crashing or hang bug, if $Comp$ crashes or hangs during the build process (line 13), or a miscompilation bug, if the running output is different from the expected output on some input (line 17).

Algorithm 14 is realized as a shell script. We implement the function Split in C++ using LLVM's LibTooling [The Clang Team, 2014]. This implementation follows the function's description in Definition 5.3.1. Similarly, the implementation of `GenerateRandomBuildConfig` follows the description of build configuration in Definition 5.3.2. We assign a random optimization flag to each compilation or linking step to generate a random build configuration.

Proteus is simple to realize, yet very effective in finding LTO bugs (see Section 5.3.3). Our implementation contains only approximately 300 lines of bash scripts and 200 lines of C++ code. Its simplicity makes Proteus general and applicable to other language settings.

Algorithm 14: Proteus's main procedure

```
1  procedure Validate (Compiler Comp, TestProgram P, InputSet I):
2      begin
           /* Calculate expected output */
3          P_exe := Comp.Compile(P)
4          IO := {⟨i, P_exe.Execute(i)⟩ | i ∈ I}
           /* Perform differential testing */
5          DiffTest(Comp, P, IO)
6          DiffTest(Comp, Split(P), IO)
7  procedure DiffTest (Compiler Comp, TestPrograms 𝒫, InputOutputSet IO):
8      begin
           /* Generate configs and verify */
9          for 1..MAX_ITER do
10             σ := GenerateRandomBuildConfig(𝒫)
11             P'_exe := Comp.Compile(𝒫, σ)
12             if ∄P'_exe then
13                 ReportCrashHang(Comp, σ, 𝒫)
14             else
15                 foreach ⟨i, o⟩ ∈ IO do
16                     if P'_exe.Execute(i) ≠ o then
17                         ReportMiscomp(Comp, σ, 𝒫, i)
```

Bug Reduction

Once Proteus finds a bug, we need to reduce it before filing a report in the affected compiler's bug database. This step is important because developers usually ignore large test cases or specifically ask reporters to reduce their test cases further.

We can automate this step using delta-debugging. At a high-level, delta-debugging works by gradually reducing the input program and ensuring that the reduced program

still triggers the bug and is valid (*i.e.*, does not contain undefined behavior). State-of-the art delta-debugging reducers include Berkeley Delta [McPeak et al., 2015] and C-Reduce [Regehr et al., 2012]. Unfortunately, these reducers support reducing only a single file.

Traditional Approach to Reducing LTO Bugs Because LTO bugs normally involve multiple files, reducing them is quite challenging, especially for miscompilation bugs. In fact, the GCC official guide to reduce bugs does not even have any instruction for reducing LTO miscompilation bugs [GCC Wiki, 2015], forcing the developers to rely on their experience to craft their own reduction strategies.

The standard approach to reducing miscompilation LTO bugs is to reduce each file individually (*e.g.* with Delta or C-Reduce). This is very inefficient and error-prone as these files are normally interdependent. While reducing a file, reduction tools cannot remove constructs used in other files, as this would invalidate the integrity of the program. Therefore, the reduction results are usually unsatisfactory.

Moreover, we do not have a reliable way to detect undefined behavior in multiple-file programs. For example, the C interpreter in CompCert, which can detect undefined behavior, supports only single-file programs. This makes reducing LTO bugs even more challenging. Because we cannot check for program validity during reduction, the reduced program may contain undefined behavior and become invalid.

Reducing LTO Bugs in Proteus Fortunately, in our settings, by design the splitting function Split has a special property that allows us to perform reduction on a single file, which significantly improves reduction effectiveness.

Proposition 5.3.1 *The bug-triggering property of Split is preserved under reduction. That is,*

$$\forall P \forall Comp \forall \sigma \forall i : Bug(Comp, \mathsf{Split}(P), \sigma, i) \rightarrow$$
$$\exists P' : P' = \Delta(P) \wedge Bug(Comp, \mathsf{Split}(P'), \sigma, i)$$

where:

P *is the program whose split files trigger the bug,*

$Comp$ *is the compiler affected by the bug,*

σ is the bug-triggering build configuration,
i is the bug-triggering input, and
Δ is the reduction function.

The above claim states that, if the split files of a program P trigger a bug under some build configuration σ and some input i, the reduction tool Δ will produce a reduced (*i.e.* smaller) program P' such that its split files also trigger the same bug—to be precise the *same manifested bug characteristics*, *i.e.* crash or miscompilation—under the same configuration and input.

We leverage this property to reduce LTO bugs found by Proteus. Our reduction script first applies delta reduction on the original single file. It then uses Split to separate the reduced file, and checks for bug-triggering behavior on the split files. We carefully design the build configuration so that it is always valid for the reduced split files, although our reduction tool may eliminate several functions (and thus their corresponding split files).

5.3.3 Evaluation

We started our experiments with Proteus from the end of February 2014. We focus on testing two mainstream open-source C compilers—GCC and LLVM—because of their open bug tracking systems. This section describes the results of our testing effort in about 11 months.

Result Summary Proteus is very effective:

- *Many detected bugs*: Proteus has detected 37 bugs in GCC and LLVM. Developers have confirmed 21 of our bugs. Eight out of the 12 GCC bugs were discovered from split programs, while Csmith and Orion alone would fail to discover these. Thus, the results highlight the utility and effectiveness of Proteus.

- *Many long-latent bugs*: Many of the detected bugs have been latent in old versions of GCC and LLVM. These bugs had resisted all traditional validation approaches. This further emphasizes Proteus's effectiveness.

- *All but one reported GCC bugs are fixed*: So far, 11 out of our 12 reported GCC bugs have already been fixed.

- *Diversified bugs*: Proteus has found many kinds of bugs in GCC and LLVM's LTO components. The majority are miscompilations, the most serious kind.

Testing Setup

We first describe our set up for Proteus to find LTO bugs before discussing our empirical results.

Hardware and Compilers We focus our testing on the x86-linux platform due to its popularity and our ease of access. We perform our testing on two machines (one 18 core and one 6 core) running Ubuntu 12.04 (x86_64). We only test the daily-built development trunks of GCC and LLVM. Once Proteus has found a bug in a compiler, we also validate the bug against other major versions of that compiler.

Test Programs We build our LTO test corpus by leveraging Csmith-generated programs and their variants generated by Orion. Traditionally, Orion only generates variants by removing unexecuted statements. We modify Orion to also allow inserting arbitrary function calls to these unexecuted area. We then further split these programs into separate compilation units. We run Csmith in its "swarm testing" mode [Groce et al., 2012] to maximize its effectiveness.

Real-world projects are an interesting source to test compilers. We can certainly apply Proteus on these projects to detect LTO bugs. However, reducing these projects is very challenging as they usually involve many files, each of them can be very large. It is also more difficult to detect undefined behavior in these projects. This explains why we have only focused on randomly generated programs. We leave as future work how to test with real-world projects and how to reduce any detected bugs.

Build Configurations While generating build configurations, we only use the popular compiler optimization flags (*i.e.* `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`). Each compilation and linking step is assigned with the `-flto` flag to enable LTO. We consider generating build configurations for both 32-bit (`-m32`) and 64-bit (`-m64`) environments.

For a single program, we are able to enumerate all possible optimization flags to generate build configurations. However, this is infeasible in case we split the program into separate files, because the number of files is usually large (10+ files). We need to sample this large search space and select only a certain number of build configurations to perform testing.

The number of build configurations generated for each program is a trade-off between the depth and scope of Proteus's testing. If we generate many build configurations, we may test that program more thoroughly, but we may lose the opportunity to test other programs. On the other hand, generating a few build configurations helps Proteus cover more programs, but it may not be sufficient to trigger the buggy behavior of each of the generated programs. Our empirical experience suggests that 8 build configurations strikes a good balance. We control a random parameter whose expected value is 8 and use it to generate build configurations.

Quantitative Results

Having described our testing setup, we are now ready to discuss our results using Proteus to find LTO bugs. Table 5.4 shows the details of our reported bugs.

Bug Count We have reported 37 bugs: 12 in GCC and 25 in LLVM. Till end of January, 2015, GCC developers have fixed 11 bugs. The LLVM developers have confirmed 9 of our bugs, but they have not fixed any of them. A number of private communications suggested that they were busy fixing internal bugs and working on Swift.

Before reporting a bug, we ensure that it has a different symptom from the previously reported bugs. However, reporting duplicate bugs is unavoidable, as compilers are complex, and bugs having different symptoms may turn out to have the same root cause. So far, we only reported one duplicated GCC bug (we did not include it in our results). Our LLVM bugs may contain duplicates, but from our experience on previous work [Le et al., 2014], the duplication rate is low.

Importance of Reported Bugs Because we use randomly generated programs offered by Csmith and Orion to find LTO bugs, it is reasonable to ask if these bugs

Bug ID	Bug Type	Status	Size	Rep	Test Program	Affected Vers	Modes
gcc-60319	Miscomp.	Fixed	1818	11	Csmith+Split	4.6 → 4.9-trunk	m32, m64
gcc-60404	Miscomp.	Fixed	2367	28	Orion+Split	4.9-trunk	m32, m64
gcc-60405	Crash	Fixed	3242	5	Csmith	4.9-trunk	m32, m64
gcc-60461	Link Error	Fixed	3242	37	Csmith	4.9-trunk	m32, m64
gcc-61184	Miscomp.	Fixed	2821	13	Csmith+Split	5.0-trunk	m32, m64
gcc-61278	Crash	Fixed	1446	30	Csmith+Split	5.0-trunk	m64
gcc-61602	Crash	Fixed	6659	7	Orion	5.0-trunk	m32, m64
gcc-61786	Miscomp.	Fixed	1823	26	Csmith	5.0-trunk	m32, m64
gcc-61969	Miscomp.	Fixed	1860	261	Csmith+Split	4.8 → 5.0-trunk	m32
gcc-62209	Crash	Confirm	1495	23	Csmith+Split	4.8 → 5.0-trunk	m32, m64
gcc-62238	Crash	Fixed	4276	27	Csmith+Split	4.9, 5.0-trunk	m64
gcc-64684	Miscomp.	Fixed	1745	13	Orion+Split	5.0-trunk	m32, m64
llvm-18984	Miscomp.	Confirm	2266	27	Csmith	3.2 → 3.5-trunk	m32, m64
llvm-19026	Miscomp.	Confirm	2729	10	Csmith	3.2 → 3.5-trunk	m32, m64
llvm-19062	Miscomp.	New	3243	12	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19072	Miscomp.	New	1691	36	Csmith+Split	3.2 → 3.5-trunk	m64
llvm-19073	Miscomp.	New	1703	29	Csmith+Split	3.2 → 3.5-trunk	m64
llvm-19078	Miscomp.	New	3450	44	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19079	Link Error	New	3638	52	Csmith+Split	3.2 → 3.5-trunk	m64
llvm-19093	Miscomp.	New	4815	25	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19109	Miscomp.	New	8232	24	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19111	Miscomp.	New	20556	26	Csmith+Split	3.2 → 3.5-trunk	m32, m64
llvm-19132	Miscomp.	New	16626	23	Csmith+Split	3.2 → 3.5-trunk	m32, m64
llvm-19138	Miscomp.	New	5122	12	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19146	Miscomp.	New	5369	19	Orion	3.4, 3.5-trunk	m32, m64
llvm-19184	Link Error	New	4310	47	Orion	3.5-trunk	m32, m64
llvm-19201	Miscomp.	New	9821	17	Orion	3.4, 3.5-trunk	m64
llvm-19202	Miscomp.	New	3043	10	Orion	3.2 → 3.5-trunk	m32
llvm-19219	Miscomp.	New	2169	23	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19225	Miscomp.	New	2241	21	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19830	Link error	Confirm	1291	15	Orion	3.5-trunk	m32, m64
llvm-19885	Miscomp.	Confirm	9748	24	Orion	3.2, 3.3, 3.4	m32, m64
llvm-19889	Miscomp.	Confirm	8098	14	Orion	3.2 → 3.5-trunk	m32, m64
llvm-19891	Miscomp.	Confirm	11161	26	Orion	3.4, 3.5-trunk	m32, m64
llvm-19907	Miscomp.	Confirm	5356	25	Orion	3.5-trunk	m32, m64
llvm-20172	Miscomp.	Confirm	4683	59	Orion	3.5-trunk	m32, m64
llvm-20237	Miscomp.	Confirm	3502	44	Orion	3.4, 3.5-trunk	m64

Table 5.4: The valid reported bugs for GCC and LLVM. Rep is reported size.

	GCC	LLVM	TOTAL
Miscompilation	6	22	28
Crash	5	0	5
Link Error	1	3	4

Table 5.5: Bug classification.

really matter in practice. The related discussions in Csmith [Yang et al., 2011] and Orion [Le et al., 2014] are quite relevant here.

First, developers have acknowledged and fixed these bugs. GCC developers are impressively responsive; they generally confirmed our bugs within one day, and fixed them after three days on average. Second, some of these bugs were marked as critical. In fact, the GCC developers marked a third of our reported bugs as P1, the most severe, release-blocking type of bugs. Finally, both Csmith and Orion have encountered cases where compiler bugs triggered by real-world programs were actually linked to their reported bugs derived from random programs. We expect this to also hold for Proteus as we continue finding and reporting LTO bugs.

Bug Type We classify bugs into two main categories: (1) bugs that manifest when we compile programs, and (2) bugs that manifest when we execute the compiled programs. A compile-time bug can be a *compiler crashing bug* (*e.g.*, internal compiler errors), *complier hang bug* (*e.g.*, the compiler hangs while compiling the program), or *linking error bug* (*e.g.*, the compiler cannot link object files into an executable file). A runtime bug happens when the compiled program behaves abnormally *w.r.t.* its expected behavior. For example, it may crash, hang, or produce wrong output. We refer to these bugs as *miscompilation* bugs. These bugs are the most serious, because the compiled programs silently produce wrong results.

Table 5.5 classifies the bugs found by Proteus according to the above taxonomy (note that Proteus has not yet encountered any hang bugs). This result shows that the majority of LTO bugs found by Proteus are miscompilation bugs, the most important type. This is expected because we specifically target the LTO component.

Affected Compiler Versions Our strategy is to test only the latest development trunks of GCC and LLVM. Once a bug is found, we also use it to validate other versions. Another strategy is to test all compiler versions in parallel. We do not implement this strategy because it is much more expensive, and developers are more interested in bugs that occur in recent versions. Nonetheless, while all of our reported bugs affect the development trunk, most of them also affect earlier stable releases. These bugs had been latent for many years.

Discussion

Proteus found a few hundred inconsistencies during our testing period. We managed to reduce a good fraction of them, and reported 37 bugs.[1] However, we are yet to reduce many other interesting ones because the current reduction tools do not work very well for these programs. In general, these programs have very deeply nested constructs, and neither C-Reduce nor Berkeley Delta is able to simplify such constructs. For example, Proteus found a link error bug in GCC 4.7, in which the function calls recurse deeply on their arguments (*i.e.*, the function call argument is the call result of the same function, whose argument is also the call result of that function, and so on). As another example, Proteus found many LLVM bugs, in which array member accesses are deeply nested. We are developing new reduction strategies that exploit programs' syntactic and semantic structures to reduce these programs.

[1] Many inconsistencies are duplicate, thus we only report the representatives.

Chapter 6

Related Work

Our work on program synthesis is related to the large body of research, ranging from natural language processing, machine learning, to human-computer interaction, software engineering and programming languages. This section survey some representative work in each of the areas.

6.1 Natural Language Understanding

Natural Language Interfaces (NLIs) There have been numerous attempts to build NLIs for other application domains such as controlling robots [Finucane et al., 2010], performing navigation [Song et al., 2004], processing XML [Li et al., 2005], and most notably for querying databases (NLIDBs) [Androutsopoulos, 1995]. To solve the NL ambiguity problem, NLIDBs normally accept only a restricted subset of natural language [Epstein, 1985, Hendrix et al., 1978]. For example, in [Epstein, 1985], relative clauses must follow their noun phrases. In contrast, users of SmartSynth can give free-form descriptions.

Another challenge in implementing NLIDBs is the capability to perform conversations with users, where they can give anaphora (referring to previous objects) and elliptical (incomplete) sentences based on the query context and previous query results [Androutsopoulos, 1995]. Because of its nature, SmartSynth does not have this problem. However, it does provide feedback on the scope of queries and performs interactive conversations with the user to resolve query ambiguities.

The main difference between SmartSynth and other NLIs is its use of type-based synthesis algorithms. While other systems have difficulties in extracting *all* necessary constraints, SmartSynth overcomes this problem by exploiting the capabilities of working with under-specifications of synthesis algorithms to complete the missing dataflow relations. We believe that our approach is applicable for building natural language interfaces for other domains as well.

Specification Extraction From NL Extracting specifications automatically from NL has long been a research dream. Kate *et al.* propose an approach to transform NL to formal languages [Kate et al., 2005]. Xiao *et al.* develop a template-based method to extract security policies from NL software documentations [Xiao et al., 2012]. Pandita *et al.* analyze API descriptions to infer their method specifications [Pandita et al., 2012]. Our approach complements these existing approaches on specification extraction. SmartSynth can also leverage advances in this area to improve its NLP engine's capability of identifying components and detecting dataflow relations, which may reduce the burden on its synthesis engine to complete the relations.

General-purpose Programming Using NL and Keywords NaturalJava is an attempt to bridge the gap between NL and programming languages [Price et al., 2000]. It performs translation from a restricted form of NL, which is centralized around Java's programming concepts, to Java statements. It requires the user to think and give descriptions at the syntactical level of Java. Little and Miller propose a code completion tool that synthesizes the most likely Java expression in a code context from a set of keywords [Little and Miller, 2007]. SmartSynth is different, in that it synthesizes a complete script and does not require extra contextual information.

Metafor [Liu and Lieberman, 2005a, Liu and Lieberman, 2005b] considers the programming task as telling a story. It automatically generates the program structure from a given NL description. It can extract classes and method names from the description, but the paper is unclear about how to extract the constraints to form sequences of statements inside the methods.

Digital Assistants Siri [Apple, 2013] is a virtual personal assistant that allows users to ask questions and give verbal commands. It originated from the CALO project, the "largest known artificial intelligence project in U.S. history [1]" [SRI International, 2015]. Siri can handle a wide range of queries, but only those that are simple and relate to a single phone functionality [Aron, 2011]. In other words, users cannot give complicated commands that combine different phone features. Voice Action [Google, 2013] is another personal assistant application targeting Android phones. However, it requires its users to give queries that fit pre-defined patterns. It also does not allow compositions of supported functionalities.

In contrast, users of SmartSynth can give commands containing events, conditions, and actions without any restrictions. Thus, SmartSynth provides users with greater flexibility and control over their smartphones.

6.2 Program Synthesis

Program synthesis is the task of automatically synthesizing a program in some underlying domain-specific language from a given specification using some search techniques [Gulwani, 2010]. It has been applied to various domains [Gulwani, 2010], including bit-vector algorithms [Gulwani et al., 2011], graph algorithms, mutual exclusion algorithms, intellisense for auto completion [Mandelin et al., 2005, Gvero et al., 2011], string transformations [Gulwani, 2011, Singh and Gulwani, 2012a], number transformations [Singh and Gulwani, 2012b], and table transformations [Harris and Gulwani, 2011]. In this dissertation, we apply program synthesis to two new domains: smartphone automation scripts (in SmartSynth) and data extraction (in FlashExtract and FlashProg).

Because we are targeting end users, the traditional intent specification mechanisms based on logical specifications are not applicable. Indeed, logical specifications are beyond the expertise of end-users and not worth the effort for small scripts. In SmartSynth, the specification is natural language as it is quite natural for the user to express their intent in their language. The by-example paradigm does not apply because smartphone

[1] http://sri.com/about/siri.html

scripts are not functional programs: they are event-based and have side-effects. On the other hand, natural language does not work in the data extraction domain because the tasks are inherently complicated. Users often do not know how to perform the tasks, let alone explaining it in natural language. Expressing intent in input/output examples is more natural to them.

The synthesis algorithm in SmartSynth performs *type-based synthesis* that refers to a class of search techniques that use typing information/abstraction to perform search and produces a ranked set of results. Type-based synthesis has been used for various applications: synthesizing snippets that can convert a given source type into a given target type [Mandelin et al., 2005, Gvero et al., 2011], completing partial expressions [Perelman et al., 2012], assembling a given set of APIs into a program [Gulwani et al., 2011]. In each of these cases, the programmer starts out with (incomplete) program structures and the underlying synthesis engine generates ranked completions/assemblies of these structures. SmartSynth is different from these systems in two key ways: (i) SmartSynth does not require the user to start out with program structures — these are automatically inferred from NL descriptions; (ii) SmartSynth extracts relational information from NL descriptions, which helps disambiguate between multiple solutions and significantly improves the effectiveness of ranking.

While prior synthesis techniques [Gulwani et al., 2012] are specialized to a single underlying DSL, the technique in FlashExtract is more general and can be applied to any DSL that is constructed using our core algebra. FlashExtract's extraction capability also complements the transformation capability of prior work. In fact, we have combined them together to provide a better end-to-end user experience. For example, after using FlashExtract to extract data from a text file, the user can perform string transformations [Gulwani, 2011] or number transformations [Singh and Gulwani, 2012b] to modify the extracted fields. Our prototype even allows in-place editing by examples: FlashExtract is used to highlight regions that need to be edited repetitively, and string transformation techniques [Gulwani, 2011] are used to perform transformation on leaf regions (and these changes are pushed back to the underlying document).

6.3 Data Extraction

Data Extraction from Log Files The PADS project [Fisher and Walker, 2011] has enabled simplification of ad hoc data processing tasks for programmers by contributing along several dimensions: development of domain specific languages for describing text structure or data format, learning algorithms for automatically inferring such formats [Fisher et al., 2008], and a markup language to allow users to add simple annotations to enable more effective learning of text structure [Xi and Walker, 2010]. While PADS supports parsing of entire files, FlashExtract allows users to extract only parts of the file thereby avoiding unnecessary complications. PADS's learner only supports a fixed line-by-line chunking strategy to split the records; in contrast, FlashExtract can learn chunking (aka, structure boundaries) from examples, making it suitable for extracting data fields and records that have arbitrary length (and might cross multiple lines). Finally, PADS primarily targets ad hoc text files. Although one can view webpages and spreadsheet as text files, it is unclear if the PADS learning algorithm can be adapted to work effectively for webpages and spreadsheets.

Data Extraction from Webpages Wrappers are procedures to extract data from Internet resources. Wrapper induction is the method to automatically construct wrappers [Kushmerick et al., 1997]. There has been a wide variety of work in this area, ranging from supervised systems [Hsu and Dung, 1998, Muslea et al., 1999], semi-supervised systems [Chang and Lui, 2001], to unsupervised systems [Crescenzi et al., 2001]. The difference between FlashExtract and the above systems is that its users induce wrappers by interactively giving multiple positive/negative examples. In that sense, FlashExtract is similar to [Anton, 2005]. However, the system in [Anton, 2005] only learns XPath expressions to extract HTML elements. By defining other sequence operators to handle non-HTML text (*i.e.*, text that is within a tag), FlashExtract supports finer grain extraction. For instance, FlashExtract can extract a substring or a sequence of substrings from a text tag, as in Figure 3.2. Furthermore, we can leverage advances in wrapper induction research as part of the FlashExtract general framework to support much more sophisticated extraction tasks.

In general, our work is complimentary. Since most of wrapper inductors have limited support for handling non-HTML text, we can combine advances in wrapper induction research with our general framework to support much more sophisticated extraction tasks with minimal effort.

Data Extraction from Spreadsheets Cunha *et al.* [Cunha et al., 2009] detect functional dependencies in spreadsheet data in order to automatically derive even the data schema. However, their technique is not effective over spreadsheets with *hierarchical* data. Erwig *et al.* formulate the concept of spreadsheet units – which defines headers associated with a particular cell – and use them to detect potential errors in spreadsheets [Erwig and Burnett, 2002]. Abraham *et al.* identify spreadsheet headers automatically [Abraham and Erwig, 2004] and use that to extract relational data. In contrast, FlashExtract extracts (data from) cells based on the properties of the surrounding cells. This allows FlashExtract to deal with spreadsheets with no headers. Furthermore, instead of inferring the whole schema at once, FlashExtract allows users to work in an interactive manner. Users may focus only on cells of interest–this enables robustness on complex spreadsheets.

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 in FlashProg can augment and complement Wrangler's interaction model.

OpenRefine [Metaweb Technologies, Inc., 2015] help users clean and transform their spreadsheet data into relational form. Unlike FlashExtract and FlashProg, OpenRefine requires users to program. Both Wrangler and OpenRefine are limited in their extraction capabilities over spreadsheets with hierarchical data.

6.4 Interactive User Interfaces

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.

LAPIS [Miller, 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, 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 complimentary 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. SmartEdit [Lau et al., 2000] 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]. 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.

6.5 Compiler Testing and Verification

Compiler Testing The most directly related to our Equivalence Modulo Inputs (EMI) work is compiler testing, which remains the dominant technique for validating production compilers in practice. One common approach is to maintain a compiler test suite. For example, each major compiler (such as GCC and LLVM) has its own regression test suite, which grows over the time of its development. In addition, there are a number of popular, commercial compiler test suites (*e.g.* Plum Hall [Plum Hall, Inc., 2015] and SuperTest [ACE, 2015]) designed for compiler conformance checking and validation. Most of these test suites are written manually.

Random testing complements manually written test suites. Zhao *et al.* develop JTT, a tool that automatically generates test programs to validate the EC++ embedded compiler [Zhao et al., 2009]. It takes as input a test specification (*e.g.* optimizations to be tested, data types to use, and statements to include), and generates random programs to meet the given specification. Recent work by Nagai *et al.* focuses on testing C compilers' arithmetic optimizations by carefully generating random arithmetic expressions to avoid undefined behavior [Nagai et al., 2012, Nagai et al., 2013]. As of November 2013, they have found seven bugs each for GCC and LLVM. Another notable recent random C program generator is CCG [Balestrat, 2015], which targets only compiler crashes.

Csmith [Yang et al., 2011, Regehr et al., 2012, Chen et al., 2013] has been the most successful random testing system for C compilers. It has helped find a few hundred compiler bugs over the last several years and contributed significantly to improving the quality of GCC and LLVM. It is based on differential testing [McKeeman, 1998] by randomly generating C programs and checking for inconsistent behavior across compilers

or compiler versions. What make it stand out from other random C program generators are the many C language features it supports and its careful control to avoid generating programs with undefined behavior. Thus, in addition to compiler crashes, it is suitable for finding miscompilations. Csmith has also been applied to find bugs in static analyzers, for example, in Frama-C [Cuoq et al., 2012].

Orion and Athena are complementary. Different from Csmith-like tools, they do not generate random programs, but rather consume existing code (whether real or randomly generated) and systematically modify it. EMI variants generated from existing code, say via Orion, are likely programs that people may actually write. The EMI concept is general and can be adapted to any program analysis and transformation systems. Its simplicity makes it easy to implement for a new domain — there is no need to specifically craft a new program generator each time.

Holler *et al.*'s recent work on LangFuzz [Holler et al., 2012] is also related. It is a random generator that uses failing programs as stems for producing test programs. LangFuzz has found many bugs in the PHP and Mozilla JavaScript interpreters. The same idea may be adapted to Orion and Athena. As we have already experimented in this work, problematic programs (such as those from the GCC and LLVM test suites) can be used as seeds to generate their EMI variants. We can also incorporate them in generating other programs' EMI variants, which we plan to investigate in our future work.

Verified Compilers A decade ago, "the verifying compiler" was proposed as a grand challenge for computing research [Hoare, 2003]. Compiler verification in particular has been a fruitful area for this grand challenge. A verified compiler ensures that the semantics of a compiled program is preserved. Each verified compiler is accompanied by a correctness proof that guarantees semantic preservation. The most notable example is CompCert [Leroy, 2006, Leroy, 2009], a verified optimizing compiler for a sizable C subset. Both the compiler itself and the proof of its correctness have been developed using the Coq proof assistant. The same idea has been applied to the database domain. In particular, there is some early work toward building a verified relational database management system [Malecha et al., 2010]. There is also recent work by Zhao

et al. [Zhao et al., 2013] on a proof technique to verify SSA-based optimizations in LLVM using the Coq proof assistant.

The benefits of verified compilers are clear because of their strong guarantee of semantic preservation. Despite considerable testing, neither Csmith nor Orion nor Athena has uncovered a single CompCert back-end bug to date. This is a strong testimony to the promise and quality of verified compilers. However, techniques like Csmith, Orion, and Athena are complementary as much work remains to build a realistic production-quality verified compiler. CompCert, for example, currently supports fewer language constructs and optimization techniques than GCC and LLVM (thus is less performant). These make verified compilers mainly suitable for safety-critical domains that may be more willing to sacrifice performance for increased correctness guarantees.

Translation Validation It is difficult to automatically verify that a compiler correctly translates every input program. However, it is often much easier to prove that a particular compilation is correct, which motivated the technique of translation validation [Pnueli et al., 1998]. In particular, the goal of translation validation is to verify the compiled code against the input program to find compilation errors on-the-fly. Early work on translation validation focuses on transformations among different languages. For example, Pnueli *et al.*'s seminal work [Pnueli et al., 1998] that introduced translation validation considers the non-optimizing compilation from SIGNAL to C.

Subsequent work by Necula [Necula, 2000] extends this technique to handle optimizing transformations and validates four optimizations in GCC 2.7. Extending work on super-optimization [Joshi et al., 2002, Bansal and Aiken, 2006, Massalin, 1987], in particular Denali [Joshi et al., 2002], Tate *et al.* introduce the Peggy optimization and translation validation framework for JVM based on equality saturation [Tate et al., 2009]. Tristan *et al.* [Tristan et al., 2011] adapt the work and evaluate it on validating intra-procedural optimizations in LLVM.

Although promising, translation validation is still largely impractical in practice. Current techniques focus on intra-procedural optimizations, and it is difficult to handle optimizations at the inter-procedural level. In addition, each validator is attached to a

particular implementation of an optimizer, thus changes in the optimizer may require appropriate changes in the validator. Since the validator is not verified, it may also produce wrong validation results.

6.6 Markov Chain Monte Carlo Sampling

MCMC sampling is a popular algorithm that has played a significant role in science and engineering. People have applied MCMC techniques to solve a large number of problems in statistics, economics, physics, biology, and computer science [Gilks, 1999, Andrieu et al., 2003].

In computer science, Schkufza *et al.* recently applied MCMC sampling to perform superoptimization tasks, which transform a loop-free sequence of binary statements into a more optimized sequence [Schkufza et al., 2013, Schkufza et al., 2014]. They propose a cost function that captures both correctness and performance, and some basic operations to transform the program such as replacing the opcode or operand, or swapping the statements. The cost function guides their program space exploration toward lower cost (*i.e.*, more optimized) programs.

Our technique in Athena also uses MCMC sampling to explore the space of programs but with several distinct differences: (1) we target a different domain (*i.e.*, compiler testing); (2) the cost function in our work is to maximize the diversity of generated test programs; and (3) the eligible transformations generate EMI variants instead of instruction sequences.

Chapter 7

Conclusion and Future Work

In recent years we have witnessed an explosion in innovation in many critical critical areas. The number of people with access to computing devices has exploded due to falling prices and increased functionality of devices. Most of these users are non-programmers, yet want to take full advantage of their devices. At the same time, the availability of data and tools to exploit it have led to the big data revolution. This dissertation addresses the challenges arising from these areas.

Via SmartSynth, we help users use their phone more effectively by automating their repetitive tasks using natural language. FlashExtract is our first attempt to tackle the data cleansing problem in the big data revolution. The system allows users to extract data in semi-structured formats such as text files, webpages, and spreadsheets by simply highlighting some sample regions. FlashProg takes one step further and lets user explore the space of satisfying programs (which are already paraphrased in English), and even perform conversation with the system to resolve ambiguities. These systems enable users to perform tasks that otherwise are impossible for them. We are having more and more software nowadays, but they still all depend on a critical piece, the compiler. A bug in the compiler may potentially invalidate the software it compiles. Equivalence Modulo Inputs, which generates many equivalent test programs from existing test programs to find compiler bugs, is our solution to make this critical piece of software safer.

This dissertation has demonstrated a strong focus on end-to-end approaches that are motivated by real-world applications and provide practical impact. SmartSynth has

been integrated into TouchDevelop. We ship FlashExtract with PowerShell in Microsoft Windows 10, and with Microsoft Azure Operational Insights. Our work on EMI has resulted in more than 400 bugs in GCC and LLVM. Many compiler companies have adopted EMI as part of their testing framework. We believe that these successes are just the beginning in this research direction. There are many opportunities for future work.

Multi-Modal Synthesis Traditional program synthesis techniques take as input only one form of specifications, such as logical specifications, speech/natural language descriptions, input/output examples, or demonstrations. Each form typically has different expressive power of user intent, and requires a different synthesis algorithm. We plan to build new synthesis systems that combine multiple forms of specifications. The new systems can capture user intent more precisely, and consequently reduce user interaction. The main challenge is to develop novel synthesis algorithms that seamlessly integrate various forms of specifications to help effectively prune the search space. Another challenge is to design novel interaction models to enable new interaction and debugging experience. For example, when the result is unintended, we need to identify the most suitable form of specifications to interact with users.

Data-Driven Synthesis We want to build synthesis systems that leverage the massive amount of available data. One key component of any synthesis algorithm that deals with under-specification (e.g., input/output examples) is the ranking system that is used to resolve ambiguity. The ranking is usually built from expert rules derived from extensive studies of real-world problems. With sufficient data, we can leverage modern statistical machine learning approaches to build data-oriented ranking to generate more natural programs. For example, we can build a new ranking system for SmartSynth from hundreds of thousands user scripts submitted to TouchDevelop. We can also use available data to develop new applications, such as a system to automatically correct syntax errors in TouchDevelop scripts. Such system is very useful because most TouchDevelop users are novice programmers.

Equivalence Modulo Inputs A significant strength of EMI is its generality. We can use EMI to test compilers, interpreters, database engines, and program analysis and

transformation systems in general. Our immediate plan is to extend the work to floating-point programs. The key challenge is to define the equivalence of floating-point EMI variants considering the inherent inaccuracy of floating-point computation. We also plan to apply EMI to new languages such as C++, Java, and JavaScript. Each of these languages comes with a different challenge. For instance, we do not have reliable methods to detect undefined behavior in C++ programs, which can lead to invalid bug reports. Another domain that I want to explore is the validation of database engines. We can adapt EMI to modify both the input queries (e.g. by changing the unexecuted clauses and performing semantically equivalent rewriting) and the data (e.g. by changing the unused table rows/columns or relevant statistics on data).

We are on the cusp of an exciting era where we will fundamentally change how people interact with and benefit from technologies, how software is designed and developed, how we teach students, etc. Such changes will demand cross-disciplinary mindsets and solutions. The techniques described in this dissertation, which combine advances in natural language processing, programming languages, machine learning, and human-computer interaction, make one step to fuel this change.

REFERENCES

[Abraham and Erwig, 2004] Abraham, R. and Erwig, M. (2004). Header and unit inference for spreadsheets through spatial analyses. In *VL/HCC*.

[ACE, 2015] ACE (2015). SuperTest compiler test and validation suite. `http://www.ace.nl/compiler/supertest.html`.

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[Alur et al., 2013] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–17.

[Amershi, 2011] Amershi, S. (2011). Designing for effective end-user interaction with machine learning. In *Proceedings of the 24th Annual ACM Symposium Adjunct on User Interface Software and Technology*, UIST '11 Adjunct, pages 47–50, New York, NY, USA. ACM.

[Amershi et al., 2009] Amershi, S., Fogarty, J., Kapoor, A., and Tan, D. (2009). Overview based example selection in end user interactive concept learning. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 247–256, New York, NY, USA. ACM.

[Andrieu et al., 2003] Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. (2003). An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1-2):5–43.

[Androutsopoulos, 1995] Androutsopoulos, L. (1995). Natural language interfaces to databases - an introduction. *Journal of Natural Language Engineering*, 1:29–81.

[Anton, 2005] Anton, T. (2005). Xpath-wrapper induction by generalizing tree traversal patterns. *LWA 2005 - Workshopwoche der GI-Fachgruppen/Arbeitskreise*, pages 126–133.

[Apple, 2013] Apple (2013). Siri for iPhone. `http://www.apple.com/iphone/features/siri.html`.

[Aron, 2011] Aron, J. (2011). How innovative is Apple's new voice assistant, Siri? *The New Scientist*.

[Balestrat, 2015] Balestrat, A. (2015). CCG: A random C code generator. `https://github.com/Merkil/ccg/`.

[Bansal and Aiken, 2006] Bansal, S. and Aiken, A. (2006). Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403.

[Barowy et al., 2015] Barowy, D. W., Gulwani, S., Hart, T., and Zorn, B. (2015). Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 218–228, New York, NY, USA. ACM.

[Chang and Lui, 2001] Chang, C.-H. and Lui, S.-C. (2001). Iepad: Information extraction based on pattern discovery. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 681–688, New York, NY, USA. ACM.

[Chen et al., 2013] Chen, Y., Groce, A., Zhang, C., Wong, W.-K., Fern, X., Eide, E., and Regehr, J. (2013). Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208.

[Crafty Apps, 2015] Crafty Apps (2015). Tasker for Android. `http://tasker.dinglisch.net/`.

[Crescenzi et al., 2001] Crescenzi, V., Mecca, G., and Merialdo, P. (2001). Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 109–118, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Cunha et al., 2009] Cunha, J., Saraiva, J. a., and Visser, J. (2009). From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 179–188, New York, NY, USA. ACM.

[Cuoq et al., 2012] Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., and Yang, X. (2012). Testing static analyzers with randomly generated programs. In Goodloe, A. and Person, S., editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg.

[de Marneffe et al., 2006] de Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *IN PROC. INT'L CONF. ON LANGUAGE RESOURCES AND EVALUATION (LREC*, pages 449–454.

[Ellison and Rosu, 2012] Ellison, C. and Rosu, G. (2012). An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 533–544.

[Epstein, 1985] Epstein, S. S. (1985). Transportable natural language processing through simplicity–the PRE system. *ACM Transactions on Information Systems (TOIS)*, 3(2).

[Erwig and Burnett, 2002] Erwig, M. and Burnett, M. M. (2002). Adding apples and oranges. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 173–191, London, UK, UK. Springer-Verlag.

[Finucane et al., 2010] Finucane, C., Jing, G., and Kress-Gazit, H. (2010). Ltlmop: Experimenting with language, temporal logic and robot control. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1988–1993.

[Fisher and Walker, 2011] Fisher, K. and Walker, D. (2011). The pads project: An overview. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 11–17, New York, NY, USA. ACM.

[Fisher et al., 2008] Fisher, K., Walker, D., Zhu, K. Q., and White, P. (2008). From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 421–434, New York, NY, USA. ACM.

[Frenz, 2005] Frenz, C., editor (2005). *Pro Perl Parsing*. APress.

[GCC Wiki, 2015] GCC Wiki (2015). Finding miscompilations on large testcases. `http://gcc.gnu.org/wiki/Analysing_Large_Testcases/`.

[Gilks, 1999] Gilks, W. R. (1999). *Markov Chain Monte Carlo In Practice*. Chapman and Hall/CRC.

[GNU Compiler Collection, 2015] GNU Compiler Collection (2015). Gcov - Using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[Google, 2013] Google (2013). Voice Actions for Android. `http://www.google.com/mobile/voice-actions/`.

[Groce et al., 2012] Groce, A., Zhang, C., Eide, E., Chen, Y., and Regehr, J. (2012). Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–88.

[Gulwani, 2010] Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 26–28.

[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, 2012] Gulwani, S. (2012). Synthesis from examples: Interaction models and algorithms. In *Proceedings of the 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '12, pages 8–14, Washington, DC, USA. IEEE Computer Society.

[Gulwani et al., 2012] Gulwani, S., Harris, W. R., and Singh, R. (2012). Spreadsheet data manipulation using examples. *Communication of the ACM*, 55(8):97–105.

[Gulwani et al., 2011] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. (2011). Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, 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.

[Harris and Gulwani, 2011] Harris, W. R. and Gulwani, S. (2011). Spreadsheet table transformations from examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 317–328, New York, NY, USA. ACM.

[Hendrix et al., 1978] Hendrix, G., Sacerdoti, E., Sagalowicz, D., and Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3(2).

[Hoare, 2003] Hoare, T. (2003). The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*, pages 25–35. Springer.

[Holler et al., 2012] Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*.

[Hsu and Dung, 1998] Hsu, C.-N. and Dung, M.-T. (1998). Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 23(9).

[Ii and Rothermel, 2005] Ii, M. F. and Rothermel, G. (2005). The euses spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Workshop on End-User Software Engineering*.

[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*, ICSE '10, pages 215–224, New York, NY, USA. ACM.

[Joshi et al., 2002] Joshi, R., Nelson, G., and Randall, K. H. (2002). Denali: A goal-directed superoptimizer. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314.

[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.

[Jurafsky and Martin, 2008] Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 2nd edition.

[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*, CHI '11, pages 3363–3372, New York, NY, USA. ACM.

[Kate et al., 2005] Kate, R. J., Wong, Y. W., and Mooney, R. J. (2005). Learning to transform natural to formal languages. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pages 1062–1068. AAAI Press.

[Klein and Manning, 2003] Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA. Association for Computational Linguistics.

[Kushmerick et al., 1997] Kushmerick, N., Weld, D. S., and Doorenbos, R. B. (1997). Wrapper induction for information extraction. In *IJCAI*.

[Landauer and Hirakawa, 1995] Landauer, J. and Hirakawa, M. (1995). Visual AWK: a model for text processing by demonstration. In *IEEE Symposium on Visual Languages*, pages 267–267.

[Lau, 2008] Lau, T. (2008). Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, Florence, Italy.

[Lau et al., 2000] Lau, T. A., 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.

[Le et al., 2014] Le, V., Afshari, M., and Su, Z. (2014). Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[Le and Gulwani, 2014] Le, V. and Gulwani, S. (2014). Flashextract: A framework for data extraction by examples. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[LeBlanc, 2015] LeBlanc, A. (2015). Stanford parser wrapper. `http://nlp.naturalparsing.com/`.

[Leroy, 2006] Leroy, X. (2006). Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54.

[Leroy, 2009] Leroy, X. (2009). A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446.

[Li et al., 2005] Li, Y., Yang, H., and Jagadish, H. V. (2005). Nalix: An interactive natural language interface for querying xml. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 900–902, New York, NY, USA. ACM.

[Little and Miller, 2007] Little, G. and Miller, R. C. (2007). Keyword programming in java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 84–93, New York, NY, USA. ACM.

[Liu and Lieberman, 2005a] Liu, H. and Lieberman, H. (2005a). Metafor: Visualizing stories as code. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pages 305–307, New York, NY, USA. ACM.

[Liu and Lieberman, 2005b] Liu, H. and Lieberman, H. (2005b). Programmatic semantics for natural language interfaces. In *In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI-2005)*.

[Malecha et al., 2010] Malecha, G., Morrisett, G., Shinnar, A., and Wisnesky, R. (2010). Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248.

[Mandelin et al., 2005] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA. ACM.

[Massalin, 1987] Massalin, H. (1987). Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126.

[McDaniel and Myers, 1999] McDaniel, R. G. and Myers, B. A. (1999). Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 442–449, New York, NY, USA. ACM.

[McKeeman, 1998] McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal*, 10(1):100–107.

[McPeak et al., 2015] McPeak, S., Wilkerson, D. S., and Goldsmith, S. (2015). Berkeley Delta. `http://delta.tigris.org/`.

[Metaweb Technologies, Inc., 2015] Metaweb Technologies, Inc. (2015). OpenRefine. `http://openrefine.org/`.

[Microsoft Research, 2013] Microsoft Research (2013). on{X}. `http://onx.ms/`.

[Miller, 2002] Miller, R. C. (2002). *Lightweight Structure in Text*. PhD Dissertation, Carnegie Mellon University.

[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.

[MIT Center for Mobile Learning, 2015] MIT Center for Mobile Learning (2015). MIT App Inventor for Android. `http://appinventor.mit.edu/`.

[Mitchell, 1982] Mitchell, T. M. (1982). Generalization as search. *Artificial intelligence*, 18(2):203–226.

[Muslea et al., 1999] Muslea, I., Minton, S., and Knoblock, C. (1999). A hierarchical approach to wrapper induction. In *Proceedings of the Third Annual Conference on Autonomous Agents*, AGENTS '99, pages 190–197, 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.

[Nagai et al., 2012] Nagai, E., Awazu, H., Ishiura, N., and Takeda, N. (2012). Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53.

[Nagai et al., 2013] Nagai, E., Hashimoto, A., and Ishiura, N. (2013). Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93.

[Necula, 2000] Necula, G. C. (2000). Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94.

[Oro et al., 2010] Oro, E., Ruffolo, M., and Staab, S. (2010). Sxpath: Extending xpath towards spatial querying on web documents. *Proc. VLDB Endow.*, 4(2).

[Pandita et al., 2012] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., and Paradkar, A. (2012). Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 815–825, Piscataway, NJ, USA. IEEE Press.

[Perelman et al., 2012] Perelman, D., Gulwani, S., Ball, T., and Grossman, D. (2012). Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 275–286, New York, NY, USA. ACM.

[Plum Hall, Inc., 2015] Plum Hall, Inc. (2015). The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`.

[Pnueli et al., 1998] Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166.

[Polozov and Gulwani, 2015] Polozov, O. and Gulwani, S. (2015). Flashmeta: A framework for inductive program synthesis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '15, New York, NY, USA. ACM.

[Price et al., 2000] Price, D., Rilofff, E., Zachary, J., and Harvey, B. (2000). Naturaljava: A natural language interface for programming in java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, IUI '00, pages 207–211, New York, NY, USA. ACM.

[Regehr et al., 2012] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. (2012). Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346.

[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*, ICSE '08, pages 1–10, New York, NY, USA. ACM.

[Schkufza et al., 2013] Schkufza, E., Sharma, R., and Aiken, A. (2013). Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316.

[Schkufza et al., 2014] Schkufza, E., Sharma, R., and Aiken, A. (2014). Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 53–64.

[Singh and Gulwani, 2012a] Singh, R. and Gulwani, S. (2012a). Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751.

[Singh and Gulwani, 2012b] Singh, R. and Gulwani, S. (2012b). Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg. Springer-Verlag.

[Song et al., 2004] Song, I., Guedea, F., Karray, F., Dai, Y., and El Khalil, I. (2004). Natural language interface for mobile robot navigation control. In *Intelligent Control, 2004. Proceedings of the 2004 IEEE International Symposium on*, pages 210–215.

[SRI International, 2015] SRI International (2015). Cognitive Assistant that Learns and Organizes. http://www.ai.sri.com/project/CALO.

[Standard Performance Evaluation Corporation, 2015] Standard Performance Evaluation Corporation (2015). SPEC CINT2006 Benchmarks. https://www.spec.org/cpu2006/CINT2006/.

[Tate et al., 2009] Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. (2009). Equality saturation: a new approach to optimization. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276.

[The Clang Team, 2014] The Clang Team (2014). Clang 3.4 documentation: LibTooling. http://clang.llvm.org/docs/LibTooling.html.

[Tillmann et al., 2011] Tillmann, N., Moskal, M., de Halleux, J., and Fahndrich, M. (2011). Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 49–60, New York, NY, USA. ACM.

[Times, 2014] Times, N. (2014). For big-data scientists, 'janitor work' is key hurdle to insights. http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html?_r=0.

[Tristan et al., 2011] Tristan, J.-B., Govereau, P., and Morrisett, G. (2011). Evaluating value-graph translation validation for LLVM. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–305.

[Two forty four a.m. LLC., 2013] Two forty four a.m. LLC. (2013). Locale. http://www.twofortyfouram.com/.

[Wikipedia, 2014] Wikipedia (2014). Jaccard index. http://en.wikipedia.org/wiki/Jaccard_index.

[Wirth, 1971] Wirth, N. (1971). Program development by stepwise refinement. *Communication of the ACM*, 14(4).

[Witten and Mo, 1993] Witten, I. H. and Mo, D. (1993). TELS: learning text editing tasks from examples. In *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA.

[Xi and Walker, 2010] Xi, Q. and Walker, D. (2010). A context-free markup language for semi-structured text. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 221–232, New York, NY, USA. ACM.

[Xiao et al., 2012] Xiao, X., Paradkar, A., Thummalapenta, S., and Xie, T. (2012). Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 12:1–12:11, New York, NY, USA. ACM.

[Yang et al., 2011] Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294.

[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*, UIST '13, pages 495–504, New York, NY, USA. ACM.

[Zhao et al., 2009] Zhao, C., Xue, Y., Tao, Q., Guo, L., and Wang, Z. (2009). Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test (AST)*, pages 36–43.

[Zhao et al., 2013] Zhao, J., Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. (2013). Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186.