UNIVERSITY OF CALIFORNIA SAN DIEGO

Language Design for Synthesizing Diagrams, Layouts, and Invariants

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

John Terry Sarracino

2020

DEDICATION

To my wonderful and supportive wife, parents, and friends. I love you all. I could not have finished this without your continuous companionship and support.

EPIGRAPH

In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences.

C. A. R. Hoare[52]

Most programs are nonsense and wrong.

Tristan Knoth

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professors Sorin Lerner and Nadia Polikarpova for their mentorship and professional advice. Research with them has been a joy :).

I would also like to thank the amazing ProgSys lab at UCSD. On our many lunches, weight-lifting sessions, and frantic cleaning frenzies, ya'll have made UCSD a second home.

In addition, our student-worker's union at UCSD, UAW 2865, provided me with lots of companionship and enriched my soul. Solidarity ya'll.

Finally, I would like to especially thank my wife, Tamires Brito Da Silva, who has had the patience and kindness to support me through the ups and downs of grad school and through the many deadline crunches.

Less personally:

Thanks very much, Steve Checkoway, for your awesome thesis latex template!

Chapter 2, in full, is a reprint of the material as it appears in CHI Conference on Human Factors in Computing Systems 2017. Sarracino, John; Barrios-Arciga, Odaris; Zhu, Jasmine; Marcus, Noah; Lerner, Sorin; Wiedermann, Ben., ACM Press, 2017. The dissertation author was a primary investigator and author of this paper.

Chapter 3, in part is currently being prepared for submission for publication of the material. Sarracino, John; Lukes, Dylan; Coleman, Cora; Weingarten, Ariel; Lerner, Sorin; Polikarpova, Nadia. The dissertation author was a primary investigator and author of this material.

Chapter 4, in part is currently being prepared for submission for publication of the material. Sarracino, John; Barke, Shraddha; Peleg, Hila; Lerner, Sorin; Polikarpova, Nadia. The dissertation author was a primary investigator and author of this material.

VITA

2014	Bachelor of Science, Harvey Mudd College
2015–2020	Teaching Assistant, University of California San Diego
2015-2020	Research Assistant, University of California San Diego
2020	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

Language Design for Synthesizing Diagrams, Layouts, and Invariants

by

John Terry Sarracino

Doctor of Philosophy in Computer Science

University of California San Diego, 2020

Professor Nadia Polikarpova, Chair
Professor Sorin Lerner, Co-Chair

Program synthesis is a promising area of research concerned with automatically producing program implementations from high-level specifications of their behavior. Using synthesizers, programmers can write declarative and natural specifications instead of low-level implementations, and the synthesizer ensures that the resulting program does not contain errors.

The promise of synthesis is both elegant and compelling: wouldn't it be great if we didn't need to code and the compiler could magically transform a clearly, obviously correct specification into an executable implementation?

While this promise is enticing, alas there is no free lunch. Synthesis is fundamentally a search problem over the unbounded space of possible implementations. As a consequence, applications of program synthesis must bound the search space in an intelligent way, typically through clever language design of the space of possible implementations. Broadly speaking research in this field involves a tradeoff between the generality of the implementation language (i.e. how domain-specific possible programs are), and the completeness of the synthesizer (i.e. what types of programs the synthesizer can find).

In this thesis, I develop synthesizers in three different problem domains and explore the tradeoff space, from a very domain-specific and complete synthesizer to a general, domain-agnostic synthesizer that significantly restricts the space of output solutions. In particular, I present synthesis algorithms and languages for: (1) enabling non-programmers to add interactive behavior to static diagrams; (2) inferring dynamic visual layouts from input-output examples; (3) simpler and robust imperative programs through automatically maintained data invariants.

Pleasingly in all cases, I demonstrate ways in which a synthesizer can deliver on the promise of easing and eliminating the burden of programming.

Introduction

Humans have programmed computers for decades and indeed, programming is considered an art and a discipline. [69]. However, *manual programming*, i.e. the process of writing handwritten, interpretable computer programs, is tedious, error-prone, and difficult. Despite decades of research into *how to program* it remains the case that manual programming is necessary.

This is unfortunate for three reasons.

Error-prone: Low-level algorithmic programming is prone to implementation errors. These errors can range in consequence from benign (e.g. graphical user interface bugs), to expensive (errors in rockets and financial markets), to deadly (an error in the Therac 25 x-ray machine infamously killed several people [78]).

Barrier to entry: Low-level programming creates an artificial barrier to entry for tasks that require programming. Not only must the developer understand their problem, but they must also understand how to implement their problem in a programming language. Indeed, the field of end-user development has emerged exactly to tackle this knowledge gap.

Maintenance burden: Low-level implementations add an additional maintenance burden for software developers. As the overall system changes and adapts, the developers must think about both the changes to the system as well as the changes to the software implementation.

Program Synthesis

Program Synthesis is a research field concerned with raising the level of abstraction of programming by generating executible code from high-level specifications. These specifications can take the form of logical formulae (i.e. synthesis from specifications) or input-output examples (i.e. inductive synthesis). Because many different people program, there are a broad variety of domains in which synthesis can enable easier, more reliable programs.

In this thesis we survey some recent related work on eliminating manual programming in chapter 1 and present three different applications of synthesis drawn accross the end-user spectrum: in chapter 2, EDDIE, which removes the barrier of programming for interactivity in diagrams; in chapter 3, MOCKDOWN, which generates linear layout constraints for dynamic web layouts from input-out examples of the layout; and in chapter 4, SPYDER, which reduces the burden of general-purpose programming by automatically maintaining data invariants.

Eddie

Interactive diagrams are difficult to build and require significant programming experience. The knowledge barrier of building such diagrams often prevents novice programmers or non-programmers from directly authoring them. In this chapter, we present user-guided synthesis techniques that transform a static diagram into an interactive one without requiring the user to write code. We also present a tool called EDDIE that prototypes these techniques. We evaluate EDDIE through: (1) a case study in which we use EDDIE to implement existing real-world diagrams from the literature and (2) a usability session with end-users (K-12 teachers and college professors) build several diagrams in EDDIE and provide feedback on EDDIE's user experience. Our experiments demonstrate that EDDIE is usable and expressive, and that EDDIE enables real-world diagrams to be implemented without requiring programming knowledge.

Mockdown

Declarative constraint systems, such as CSS[79], AutoLayout[110], and ConstraintLayout [45], are a powerful and common technique for building visual layouts, mainly because because they enable a single declarative layout to dynamically adapt and conform to different display configurations and sizes. Despite their power, constraint-based layouts are notoriously complicated to create and typically require difficult manual programming and configuration.

In this chapter, we present inductive synthesis techniques for synthesizing general layout constraints from very few examples of the layout.

The main challenge is the *complexity* of of the problem. Realistic layouts are *deeply nested and have many distinct visual elements*, and general layout constraints can potentially relate *any number of elements together*. In tandem these two factors cause traditional inductive synthesis to not scale. To address the complexity of the problem, we introduce HIERARCHICAL DECOMPOSITION, which leverages the nested nature of many common layouts to decompose the search into separate, tractable suproblems.

We realize these techniques in a tool termed MOCKDOWN. We evaluate the expressiveness of MOCKDOWN in two ways. First, we conduct a *linearity case study* of many common layouts from the Alexa top 10. We find that common layouts can be precisely modelled using linear constraints. Second, using MOCKDOWN we synthesize the layout of DuckDuckGo [126], a popular search engine. We find that our correctness predicates enable MOCKDOWN to find a good layout from just one example, and in fact two examples are sufficient to completely specify the layout. Moreover, HIERARCHICAL DECOMPOSITION enables MOCKDOWN to significantly outperform traditional symbolic inference algorithms, and indeed we find that traditional algorithms to not scale to DuckDuckGo's layout.

Spyder

Data structures in a program are frequently subject to *data invariants* relations that must be maintained throughout program execution. Traditionally, invariants are implicit and are enforced by manually-crafted code. Manual enforcement is error-prone, as the programmer must account for all locations that might break an invariant. Moreover, implicit invariants are brittle under code evolution: when the invariants and data structures change, the programmer must repeat the process of manually repairing all of the code locations where invariants are violated. In this chapter, we present *programming with data invariants*, a new programming

model where invariants are exposed to the programmer as a language feature and statically enforced by the compiler. Importantly, whenever programmer's code breaks an invariant, the compiler synthesizes a patch to restore it. The two main challenges for implementing such a compiler are to make patch synthesis efficient and to avoid reverting changes made by the programmer. To tackle these challenges, we introduce TARGETED SYNTHESIS, an efficient patch synthesis algorithm, which exploits structural similarity between invariants and code to localize and simplify the synthesis problem. We evaluate our programming model and synthesis algorithm on a prototype language, SPYDER, which is a core imperative language with collections, and supports a restricted but useful class of data invariants, which we term *iterator-based invariants*. We evaluate the succinctness and performance of SPYDER on a variety of programs inspired by web applications, and show that SPYDER allows for more concise programs, and efficiently compiles and maintains data invariants.

Chapter 1

Related work

Broadly speaking our work applies techniques from program synthesis to the domains of interactive diagrams, dynamic layouts, and imperative programming. We first give context for the area of program synthesis and then give specific context for closely related work in each of the projects.

1.1 Program Synthesis

In recent years, *program synthesis* has emerged as a promising technique for automating tedious and error-prone aspects of programming [48, 113, 120]. Program Synthesis infers a program from a partial program or a specification [47]. The two main directions in this area are synthesis from informal descriptions (such as examples, natural language, or hints)[11, 105, 98, 39, 112, 38, 140, 92, 25] and synthesis from formal specifications, where the goal is to synthesize a program that is provably correct relative to the specification [49, 116, 67, 34, 102, 73]. Synthesis is difficult because the search space is large, which is typically addressed by limiting the domain language [50, 51, 115, 59, 97], cleverly enumerating programs [70, 141], or asking the human for help [109].

In EDDIE, we narrow the search space by (1) using a constraint-based formalism whose structure we can exploit and (2) asking the human to make key decisions. In MOCKDOWN, we narrow the search space by (1) again using a constraint-based formalism, (2) leveraging the geometry of layouts, and (3) leveraging the layout hierarchy to generate subproblems. In SPYDER, we narrow the search space by leveraging both the structure of the existing code, as well as the syntactic structure of invariants; this is most similar to work in the context of information-flow security [41, 103].

1.2 Diagram and Layout Related Work

Interactive diagrams and visual layouts are closely related due to their common visual nature.

1.2.1 Visual Editors

There is a long and rich line of work on visual editors for diagrams, for example [62, 12, 56, 32, 143, 64, 139, 65, 138, 137]. To understand how EDDIE fits into this broad line of research, it's important to first note that the vast majority of prior editors (including all those cited above) support either *static* or *animated* diagrams, but not authoring of *interactivity* for the viewer, which is our main goal.

Much fewer editors support authoring of interactivity – Kitty [63] and Apparatus [1] are the main examples in this space. Our work is different from these editors in two ways. First, our approach of encoding diagrams using constraints and synthesizing the constraints automatically is novel. Second, our tool requires much less expertise to use than prior approaches. Consider for example Apparatus [1], which is a direct-manipulation editor for authoring interactive dataflow diagrams. While Apparatus is a powerful and sophisticated system, to use Apparatus the author must first become familiar with the tool's dataflow programming paradigm. Consider as another example Kitty [63], which is sketch-based project for professional authoring of interactive animations. While Kitty enables the diagram author to add end-user interactivity to an animation, the author in Kitty must correctly express a global interactivity modality as the composition of pairwise element functional relationships.

In contrast to prior work, in EDDIE, the author must only: (1) draw a static, non-animated diagram, in a style that is familiar to many computer users (2) select between self-animating previews. This is a substantially lower cognitive burden: in our tool, the author has only two concepts to cognitively process (static diagrams and previews of dynamic diagrams) whereas in prior tools, the author must also process the language/mechanism for expressing interactivity (note that we have such a mechanism—the constraint language—but we *generate* the constraints automatically). We validate in our experiments that users with no prior knowledge can quickly generate interactive diagrams after just seeing a 15 minute demo.

There is however a key tradeoff here: while our approach reduces the cognitive burden required to build interactive diagrams, we don't provide the same expressiveness as prior approaches (as we will discuss in the Limitations and Future Work section of chapter 2). As such, our work complements prior work: we provide good automation for a subset of all possible diagrams, and authors can go to more complex techniques/approaches for the rest.

1.2.2 Constraints in User Applications

Constraints have been used for visual layout for many years, dating back to Sketch-Pad [119]. More recent work uses constraints for GUI builders [95, 94, 122], for user interactions [121], and as programming paradigms [96, 36]. The DeltaBlue project [42] coined the notion of an *incremental* constraint solver, which can dynamically resolve a system when constraints are added or removed. Our work presents an application of constraint solving to the domain of interactive diagram synthesis: indeed, both EDDIE and MOCKDOWN use Cassowary [5, 15], an incremental constraint solver also used in Mac OS X.

Inference of Layout Constraints. Closely related to EDDIE and MOCKDOWN is research on layout editors that use constraints in the back-end, for example the work on Programming by Manipulation for Layout [55], The Auckland Layout Editor for GUIs [142], and RockIT[60].

EDDIE is different in that, instead of just inferring layout constraints, our work also infers (with the help of a human) the constraints for *interactivity*: what happens when control points in a diagram are dragged by a user.

MOCKDOWN is different in that all of these tools pick constraints using various clever heuristics. In contrast, we develop and use robust logical correctness predicates to ensure that the inferred constraints generalize beyond the input examples.

Programming by Demonstration. Programming by Demonstration (PbD), also know as Programming by Example, builds programs from example input-output pairs [29, 93]. PbD is used in a variety of settings, such as generating web scripts [77], building visual tutorials [43], creating multi-touch interactions [82], and even inferring dynamic layouts [19].

In contrast to PbD, EDDIE does not use example program input-output pairs. While MOCKDOWN does use input-output pairs, InferUI [19] is the only other tool that also targets dynamic layouts. In MOCKDOWN, we target deeply nested web layouts, while InferUI targets smaller, flat Android layouts, does not have a predicate for eliminating ambiguity, and is focused on scaling to many more possible root dimensions than our work. As a consequence HIERARCHICAL DECOMPOSITION and the statistical pruning techniques of InferUI are complementary, and moreover our evaluation demonstrates that the basic algorithm used by InferUI does not scale to web layouts.

Interactive Technical Diagrams. There has been a lot of recent interest in digitizing figures and diagrams for STEM education [54, 3]. A large and successful project integrating computers with education is the Physics Education Technology (PhET) project [100], which builds and evaluates interactive diagrams for K-12 [9, 90, 101]. PhET's diagrams require significant programming expertise to build. The project employs 4 full-time software developers and states that each diagram typically involves a software developer, a scientist, and an educator [7]. In EDDIE our contribution is to enable users with no programming expertise to directly build interactive diagrams.

Interactive Diagram Toolkits. Kitty [63] is a sketch-based project for professional authoring of interactive animations. Kitty's model of interaction is that of *functional relationships* with one parameter, e.g., variable *A* as a function of variable *B*. The functional relationship model fits well within the domain of sketch-based animation authoring; however, many useful diagrammatic relationships involve more than two variables and so can't be expressed with Kitty's functional relationships. In addition, users of Kitty must specify the functional relationship between each diagram element and the remaining diagram elements, which leads to a quadratic growth in the number of user-provided relationships. In contrast, our use of constraints in EDDIE enables the technique to not only encode many useful relationships between more than two variables, but to do so very succinctly, without having to explicitly state pair-wise relationships between all diagram elements.

Sketch-based Simulation Inference. Sketching is a body of work focused on recognizing a simulation or animation from a user's diagrammatic sketch. By relying on properties of particular domains, sketching has been applied effectively to mechanical systems [12, 13], vector spaces [24], and fluid simulations [143]. In addition, more general sketching systems have also been developed [32, 58]. With the exception of Kitty [63], sketching work builds non-interactive diagrams: in contrast, we use program synthesis techniques to build interactive diagrams.

1.3 Imperative Program Synthesis

SPYDER builds upon two lines of prior work, which until now have developed independently: *declarative constraint programming*, where the goal is to enforce global constraints at run time, and *program synthesis* and *repair*, which enforces traditionally local, end-to-end functional specifications at compile time. We first discuss the trade-offs between static and dynamic constraint solving, and then we detail each of these areas.

Static and Dynamic Constraint Solving. Two of the longstanding research problems for constraint solving are performance [42, 15, 37], as well as debugging over- and under-constrained systems [37, 55, 111, 86]. In essence, the choice of static vs dynamic constraint solving boils down to a tradeoff between issues at compile time vs issues at run time.

For performance, solving constraint statically results in (notoriously) long synthesis and compilation times, but produces fast code. Conversely, dynamic constraint solving does not require an expensive compilation pass but results in large runtime overheads, as high as 10x-100x (as reported in [37]). Consequently, the choice of static vs. dynamic for performance is a tradeoff between compilation time and runtime performance.

Debugging constraint systems is a similar story in that static systems can report a compile-time error when the system is over- or under-constrained. Conversely, dynamic systems generally attempt to resolve ill-posed systems anyway, using techniques such as constraint hierarchies [20], which results in unintuitive solutions – unintuitive because the solution does not satisfy the constraints. In either case, the ill-posed system must be debugged. In the static case, it is strictly the programmer who debugs the system, while in the dynamic case, the end user might be exposed to the ill-posed system. Consequently, the choice of static vs. dynamic for debugging is a tradeoff between programmer time and user time.

Dynamic Invariant Enforcement. There are two closely related research arcs on dynamically enforcing invariants: the field of constraint imperative programming, and the work of functional reactive programming. Both of these areas provide mechanisms for dynamically solving invariants, and both are orthogonal to our efforts because we solve constraints statically through program synthesis.

Constraint Imperative Programming.

The field of constraint solving is rich and storied [14, 31], as constraint solvers excel at calculating global solutions. Despite their power, constraint solvers are traditionally relegated to libraries. The field of Constraint Imperative Programming aims to provide first-class language support for constraint solving [44, 37, 96], but again, fundamentally our work is orthogonal because we solve constraints statically.

Functional Reactive Programming

The field of Functional Reactive Programming (FRP) provide a dataflow language for building graphical systems [123]. Although inspired by animations, FRP quickly became popular as a tool for taming web application logic [87, 28]. The most popular recent work in this field are Elm [30] and its imperative cousin React [117], which provide a language and runtime for building client-side web applications. Although popular and powerful, FRP is a general, dynamic technique for abstracting over dataflow – in contrast, our work focuses on the problem of first-class data invariants, and solves for invariant patches statically.

Program Repair. Our work is related to sound program repair [66], where the problem is, given a formal specification and a program that violates it, modify the program so that it provably satisfies the specification. Program repair, however, is a very general problem, and so lacks a-priori restrictions on modifications the algorithm is allowed to make. As a result, if the given specification is incomplete, the problem is ill-defined. In this work we show that in the setting of enforcing data invariants, the space of possible modifications can be sufficiently restricted to make repair both *predictable* and *efficient*. Where efficiency is concerned, the deductive program repair technique of [66] does not scale with the number of patches generated in one function, whereas SPYDER leverages the restrictions to solve each synthesis task independently, hence avoiding a combinatorial explosion with the number of patches.

Example-based Repair. There has also been some recent work on scaling sound program repair to larger and more realistic programs. Angelix[85] and DirectFix[84] use test cases (i.e. examples) as a partial specification for program behavior. In contrast, similar to sound program repair, SPYDER uses formal specifications (in our setting, this is also a partial specification as data invariants don't fully specify the behavior of the program).

Program Verification. The programming and invariant language of SPYDER is purposefully simple, allowing us to explore the idea of automatic invariant maintenance without getting distracted by challenges of program verification in the presence of aliasing, dynamic object structures, and arbitrary quantified invariants. There is a rich body of prior work in program verification that deals with these challenges, both in general [106, 61] and in the specific context of object invariants [18, 74, 17, 76, 91, 88, 118, 104]. Extending TARGETED SYNTHESIS to support one of these verification methodologies is an interesting direction for future work, but we consider it orthogonal to the initial exploration of programming with invariants.

Chapter 2

Eddie: User-Guided Synthesis of Interactive Diagrams

2.1 Introduction

Interactive diagrams are animated diagrams that users can interact with using a computational device such as a computer or tablet. For example, an interactive physics diagram with pulleys and weights might allow the user to move the pulleys and vary the weights, while observing the physical simulation that ensues.

Developing an interactive diagram requires programming expertise with technologies like JavaScript, HTML5 and server-side databases, and diagram authors might not have the level of programming ability required to make an interactive diagram. In this paper, we present USER-GUIDED INTERACTION SYNTHESIS, a technique that bridges this gap by enabling users to build good interactive diagrams without requiring any programming knowledge. Instead of writing code directly to implement an interactive diagram, users first make a static version of the diagram and then our technique *synthesizes* (i.e., adds) interactivity automatically. This technique also allows the author to visualize alternative interactivity models, so the author can quickly explore the space of possible interactive diagrams, and select and/or refine the ones that are the most appropriate for the specific goals.

The main challenges in making program synthesis-based techniques work are twofold: (1) usability/expressiveness and (2) computational tractability. For the former, the synthesis langauage must be expressive and general enough to implement a variety of interaction models, while succinct enough to be amenable to synthesis and usable by non-programmers. For the latter, even with a succinct intermediate representation the search space is extremely large and underconstrained as the number of possible programs that implement interactive diagrams is unbounded and metrics for suitable interactions vary among diagrams.

To address these problems, USER-GUIDED INTERACTION SYNTHESIS makes use of the following two key ideas:

- **Constraint-based formalism**: This technique expresses and runs interactive diagrams using a constraint-based formalism with dynamically-adjusted constraints. This additional structure more effectively explores the search space of programs that implement interactive diagrams: instead of looking at all possible *programs*, the search looks at all possible *constraints* over a set of clearly defined variables. Furthermore, dynamic constraints offer a succinct and natural way of capturing the variation in interactivity across diagrams.
- **User-guided preview-based synthesis**: This technique employs a user-guided and interactive synthesis technique for programs in the constraint-based formalism. It does not rely on programming knowledge. Instead it uses a *preview-based* approach where the diagram author is shown previews of various interaction modalities, from which the author can pick the best one. The preview-based technique drastically reduces the user's manual effort while also narrowing down the search space.

Using the above ideas, we built a diagram editor called EDDIE (screenshot in Figure 2.1a) that implements USER-GUIDED INTERACTION SYNTHESIS. To demonstrate feasibility on a real-world application of interactive diagrams, we picked the domain of physics education. As benchmarks we used the interactive diagrams from the Physics Education Technology project (PhET) [100]. We performed our evaluation along three dimensions, *expressiveness*, *utility*, and *usability*.

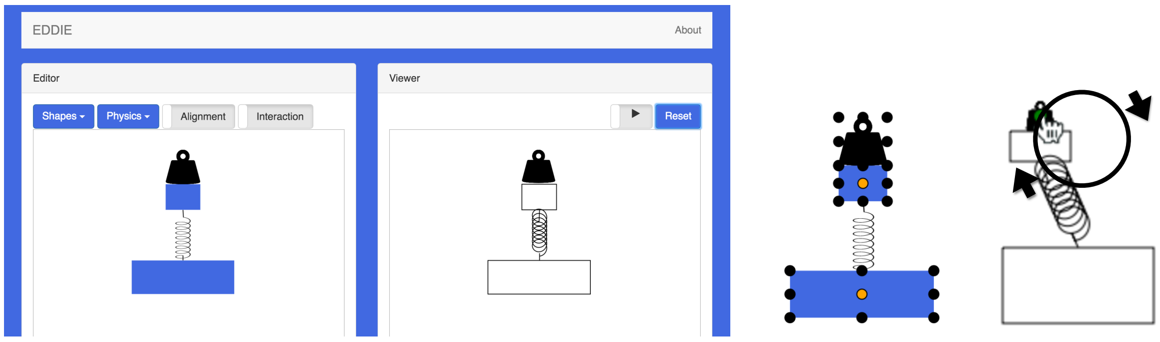
For expressiveness and utility, we performed a case study in which we used EDDIE to re-implement a variety of existing PhET diagrams. Of the 11 existing PhET diagrams on gravity, springs, and pendulum motion, EDDIE generates analogous versions for 9 of these 11 diagrams. Making these diagrams with EDDIE took a few minutes of human effort each. By comparison, these 11 benchmarks as implemented in PhET each required an average of roughly 5000 lines of expertly-written code.

For usability, we recruited a group of science teachers to author two diagrams using EDDIE and provide feedback on their experience. All of the participants were able to complete the task within an hour, and none of the participants required direct aid from the authors.

Our experiments demonstrate that EDDIE is usable and expressive, and that USER-GUIDED INTERACTION SYNTHESIS allows real-world diagrams to be authored with much less effort than the original diagrams.

Contributions. In this work we develop the following contributions:

- USER-GUIDED INTERACTION SYNTHESIS, a technique that transforms a static diagram into an interactive one without requiring the user to write code. The technique is summarized in the section 2.2.
- A constraint-based formalism for interactive diagrams, summarized in section 2.3
- A user-guided, preview-based synthesis engine for interactive diagrams in the constraint-based formalism language, discussed in section 2.4.
- An implementation of USER-GUIDED INTERACTION SYNTHESIS in a diagram editor named EDDIE, discussed and evaluated in section 2.5.
- An evaluation of EDDIE on a case study of PhET interactive physics diagrams, discussed in subsection 2.5.1.
- An evaluation of EDDIE on a user insight survey of science educators, discussed in subsection 2.5.2.



(a) (b) (c)

Figure 2.1. Screenshots of EDDIE: (a) the main view, with left editing pane and right preview pane (b) drag point selection view, with selected points in orange and unselected ones in black (c) animated preview pane—this particular interaction adjusts the spring's height and width to match the movement of the platform.

2.2 Overview

We begin with an overview of how EDDIE works through an example. There are two kinds of people who interact with EDDIE: users who *build* interactive diagram, which we will refer to as *authors* from now on; and users who *interact* with interactive diagrams, which we will refer to as *viewers* from now on. For a single diagram, EDDIE displays both perspectives simultaneously (screenshot in Figures 2.1a) using two panes, a left pane for the editing perspective of the author and a right pane for previewing the perspective of the viewer. The left author pane is akin to a traditional editor perspective, whereas the right pane provides a live interactive preview that is continuously updated.

For our running example, suppose that an author wants to build an interactive physics diagram depicting a platform with a weight on top of a spring, as shown in Figures 2.1a and 2.1b. The author places a weight (using an image link), a spring, a rectangle for the platform, and a rectangle for the base in the diagram by clicking the respective buttons in the "Shapes" and "Physics" dropdown menus. Once the shapes are in the diagram, the author uses the left pane of EDDIE to place the shapes in the desired configuration (as shown in Figures 2.1a and 2.1b). As the shapes are moved in the left author pane, the shapes in the right viewer pane follow.

The author next adds viewer interactivity to the diagram. EDDIE enables viewer interactivity through draggable points called *drag points*. Drag points are visible in the viewer's perspective (the right pane) and are draggable by the viewer. For this diagram, the author desires two viewer interactions through two drag points. First, when the viewer drags the *middle of the platform*, the following should happen: (1) the platform translates, but only in the Y dimension (2) the base remains in place (3) the weight translates and (4) the spring compresses/extends. Second, when the viewer drags the *middle of the base*, the entire diagram should translate in both dimensions.

To add these interactions, the author must first specify where drag points are located. This is done by clicking "Interaction", which displays all candidate drag points in the author pane. The author can then enable the desired drag points by clicking on them (clicking again disables the drag point). In the running example, the author selects the two desired drag points, in the middle of the weight and in the middle of the base, as shown in Figure 2.1b.

For each enabled drag point in the author pane, EDDIE automatically generates a tentative interaction modality. At this point, the author can interact with the diagram in the viewer pane and see what interaction modalities EDDIE has picked. Interacting with the diagram, the author notices that the drag point on the base translates the entire diagram, exactly as the author intended. The drag point on the weight moves the weight and platform and stretches the spring in both the X and Y dimensions, which—although a valid interaction—is *not* what the author intended. Instead, the author intended the weight and platform to ignore viewer input in the X dimension, i.e., to only translate and stretch in the Y dimension.

To change the interaction of the drag point on the weight, the author right-clicks the drag point in the author pane. EDDIE generates a list of valid interaction modalities for this drag point and displays this list in a preview window. The preview window shows an *animated preview* of the currently selected modality, along with "left" and "right" arrows to navigate through the list of animated previews. Each animated preview is *self-animating*, in that it shows what would happen if a viewer were to drag the drag point in a circular pattern. Figure 2.1c shows a static snapshot of such a self-animating preview. The circle and arrows are not part of the preview – they are used in Figure 2.1c to explain what path the drag point takes during the animation. These self-animating previews enable the author to quickly flip through many possible viewer interactions. In the running example, the author clicks “right” several times until the correct version is shown. The author then clicks “accept” to use this interaction modality.

The author now interacts again with the diagram in the viewer pane, and sees that the drag point on the base has the desired viewer interactivity. Furthermore, the author also tries the simulation component of the diagram in the viewer pane (the spring contains its own physical properties), and makes sure that the diagram works as intended. The diagram is now complete.

Overview of USER-GUIDED INTERACTION SYNTHESIS.

So far, this overview focused on a description of EDDIE from the perspective of the author and omitted technical details of the technique underlying EDDIE. EDDIE uses a technique we call USER-GUIDED INTERACTION SYNTHESIS, which broadly speaking works as follows. First, interactive diagrams are represented in an intermediate language in two parts: a symbolic description of the shapes and a constraint-based formulation of interaction modalities. Second, given a static diagram, which contains shape descriptions and locations of drag points (with no interactivity), a synthesis algorithm generates an ordered list of interaction modalities for each drag point, in which each interaction modality is represented by a set of constraints governing that point’s motion. Third, the author is able to preview all the interaction modalities by right-clicking on a drag point, and selecting from a list of animated previews.

The rest of this chapter explains these technical details in more depth. The next section (section 2.3) shows how to encode interactive diagrams using constraints, and the following section (section 2.4) describes our synthesis algorithm and previewing approach.

2.3 Interactive Diagrams Using Constraints

USER-GUIDED INTERACTION SYNTHESIS uses a constraint formalism to encode interactive diagrams. We first present background material on constraint systems and then we show how to use constraints to encode a given interactive diagram.

2.3.1 Background on Linear Constraint Systems

A *linear constraint* is a linear equality over variables, in this case floating-point variables. For example, $y = 3x$ and $t = 2$ are linear constraints, while $x^2 + y^2 = 3$ is not. A *linear constraint solver* takes a set of linear constraints and attempts to produce a valuation of constraint variables that satisfies the input constraints. If the input constraints are not solvable, e.g., $x = 2$ and $x = 3$, the constraint solver either throws an error or produces a variable valuation that minimizes an error metric, typically a weighted sum of violated constraints.[1] Linear constraint solving is an example of linear programming, which is typically solved using Dantzig's classic simplex algorithm [31] or a variant thereof.

This work uses a specific kind of constraint solver, namely an *incremental constraint solver* [42]. Such constraint solvers are designed specifically for interactive domains, including graphical applications, and are typically used in the following way. First, the layout of graphical objects on the screen is encoded using a set of constraints. Then, each time the viewer moves an object on the screen, the constraints are solved again to re-compute the layout of objects. A prominent example of such a constraint solver is the Cassowary [15] constraint solver, which we use in our implementation of EDDIE (and which is also used in the layout engine for Mac OS X).

Interactive constraint solvers have two features specifically designed for interactive graphical applications: *stay* constraints and *edit* constraints.

A *stay* constraint is a constraint of the form $Stay(v)$, where v is a variable. This constraint tells the constraint solver that the next time it is invoked to solve the constraints, the variable

[1]See the discussion below on constraint priorities for more details.

v should remain the same as its old value. This is useful because in most cases there are many solutions to the constraints, and the *stay* constraints enable an algorithm to state that for certain variables, all else being equal, they should remain unchanged.

An *edit* constraint is a constraint of the form $Edit(v = c)$ where v is a variable and c is a numerical constant. This constraint acts exactly like the regular constraint $v = c$, except that after the solver is done solving the constraints, it *deletes* the edit constraint. Edit constraints are used to encode changes due to viewer input. For example, if the viewer moves an object horizontally on the screen to a new position, one would create an edit constraint to temporarily set the X variable for that object to be the new position. The solver then solves the constraints (including the edit constraint) to find a new layout, after which the edit constraint is automatically removed. At this point the viewer's input has been incorporated into the current values of the variables, and so there is no need to keep the edit constraint around.

Finally, constraint solvers also have a way of *prioritizing constraints*. This is typically done by assigning constraints weights, and the constraint solver tries to minimize the sum of the weights of violated constraints. This prioritization feature is a side effect of the fact that these solvers typically use Simplex, which similarly solves constraints by minimizing the weighted sum of violated constraints. Despite being an implementation detail of the solver's algorithm, prioritization can be useful for interactive user applications.

2.3.2 Interactive Diagrams Using Constraints

Now that we have covered background material on constraint systems, we show how to encode a *particular interactive diagram* using constraints. At this point, we are not yet concerned with synthesis of interactive diagrams (i.e., synthesis of the constraints)—we are simply showing how to encode a single interactive diagram using constraints. In the **Synthesis** section, we will show how to *synthesize* an interactive diagram by exploring a set of interactive diagrams and selecting the best one using feedback from the author.

In our technique, interactive diagrams have two components: (1) a shape description

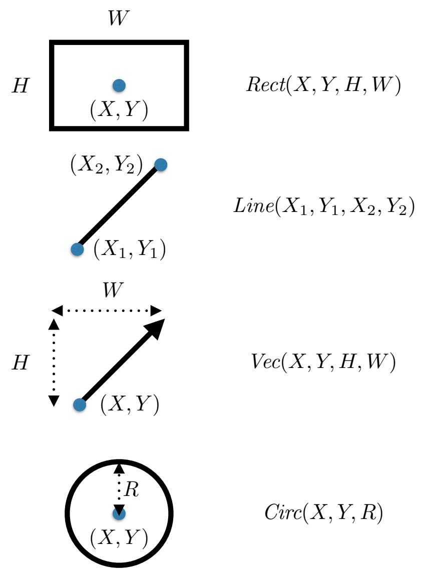


Figure 2.2. Example primitives for the shape description section. EDDIE also supports Images and Springs, which are analogous to Rectangles and Vectors respectively.

section, which states what shapes are in the diagram, and (2) a constraint section, which describes relationships between shapes as linear constraints. Figure 2.2 shows some examples of the shape primitives used in the shape description section, along with their graphical representation. The parameters to these shape primitives are constraint variables, which we call *control variables*; the relationships between these control variables are stated by equations in the constraint section.

We now describe the three different kinds of constraints and how they are used.

Layout Constraints. Layout constraints are linear equations over control variables which encode layout invariants that must remain true as the diagram is re-configured based on viewer actions. For example, one could state that two rectangles are always centered at the same location by equating their X and Y control variables; one could state that a spring is connected to the corner of a rectangle by equating the spring's end-point to an expression that corresponds to the rectangle's corner.

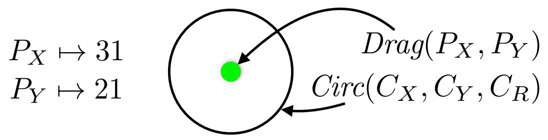
Taken in isolation, the layout constraints are usually *underconstrained*, meaning that there are many possible solutions. We address this in the standard way by adding *stay* constraints over all control variables. Although staying *all* control variables seems like it might pin the diagram and prevent it from moving, we will next describe how interactive changes happen to the diagram.

Interactivity Constraints. Whereas layout constraints capture layout invariants that must always be met, interactivity constraints are *edit* and *stay* constraints that capture interaction modalities. A viewer can interact with a diagram through *drag points*, which we capture using a primitive *Drag(X,Y)*, where *X* and *Y* are constraint variables. Intuitively a drag point *Drag(X,Y)* is simply a point that captures mouse events and sets *X* and *Y* to the mouse's position using edit constraints.

In prior work, when edit constraints were added to the system, all existing stay constraints would remain in effect. Although this might seem to lead to a system with no solution (since some points will be constrained to stay and to move at the same time), the edit constraints can be *prioritized* higher than the stay constraints, with the intention of having edit constraints over-ride stay constraints. However, to get this to work in practice requires a very careful selection of weights to get the right prioritization.

Instead, we take a different approach, in which we do not depend on any prioritization. In particular, we explicitly temporarily "unstay" the variables that we want to allow to move. Every drag point has a *free* set, which is a set of variables whose stay constraints will be temporarily removed when the drag point is moved.

Figure 2.3 shows an example of how this works. This example has a circle *C* and a drag point *P* at the middle of the circle. The free set for the drag point is $\{P_X, P_Y, C_X, C_Y\}$. The example shows what happens when the drag point is dragged from (30,20) to (31,21). The table at the bottom of Figure 2.3 shows the solver state (layout constraints, interactivity constraints, and current values of all variables) at three different points in time: (1) "Initial": before the drag happens (2) "Drag Update": after the drag happens and after the constraints are updated to reflect the drag, but before the solver has solved the new constraints (3) "Result": after the solver has solved the constraints to reflect the drag. Note how the stay constraints for the free set of *P*,



	Initial	**Drag update**	**Result**
Layout	$P_X = C_X$ $P_Y = C_Y$	$P_X = C_X$ $P_Y = C_Y$	$P_X = C_X$ $P_Y = C_Y$
Interactivity	$Stay(P_X)$ $Stay(P_Y)$ $Stay(C_X)$ $Stay(C_Y)$ $Stay(C_R)$	$Stay(C_R)$ $Edit(P_X = 31)$ $Edit(P_Y = 21)$	$Stay(P_X)$ $Stay(P_Y)$ $Stay(C_X)$ $Stay(C_Y)$ $Stay(C_R)$
Solver State	$P_X = 30$ $P_Y = 20$ $C_X = 30$ $C_Y = 20$ $C_R = 10$	$P_X = 30$ $P_Y = 20$ $C_X = 30$ $C_Y = 20$ $C_R = 10$	$P_X = 31$ $P_Y = 21$ $C_X = 31$ $C_Y = 21$ $C_R = 10$

Figure 2.3. Solver state before, as a result of, and after a drag update of (P_X, P_Y) to $(31, 21)$.

namely $\{P_X, P_Y, C_X, C_Y\}$ are temporarily removed, and then added back in.

Note that in this example if the drag point's free set had instead been $\{P_X, C_X\}$, then the stay constraints for P_Y and C_Y would have remained in effect throughout. Since this approach prioritizes stay constraints *over* edit constraints, this would make the Y-position of the circle remain the same, while still allowing the X-position to change to 31. In essence, the final result would be that the circle would only be allowed to move in the X-direction, even if the edit constraint (i.e., the viewer moving the mouse) would try to move the circle in the Y-direction. Note also that the prioritization of stay constraints over edit constraints is precisely the opposite of what prior work has done. This can be done without problems because this approach explicitly removes stay constraints for the variables that should be allowed to be free.

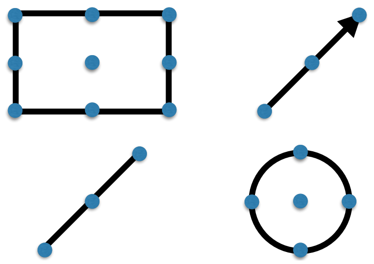


Figure 2.4. Snap points for our shape primitives. Images and Springs are analogous to Rectangles and Vectors.

2.4 User-Guided Diagram Synthesis

Now that we have described how to encode a *single* interactive diagram using constraints, we can now present how interactivity synthesis works. The key insight in this synthesis approach is that it is *user-guided*: during synthesis this approach will ask the author for help with certain decisions that the author is best equipped to do. This collaboration between human and computer enables USER-GUIDED INTERACTION SYNTHESIS to avoid expensive work and incorrect results, while also reducing the work that the author needs to do.

2.4.1 Generating Layout Constraints

After the author draws the static diagram, USER-GUIDED INTERACTION SYNTHESIS first generates layout constraints. Layout constraints capture adjacency relationships, invariants such as “a spring connects P and Q", or “X lies right next to Z". To do so, we first define *snap points* for each of our shape primitives and express the snap point coordinates in terms of shape control variables. Recall that shape control variables are the constraint variables passed as parameters to shape primitives like *Rect*. Figure 2.4 shows the snap points for several of our shape primitives. For example, given a vector rooted at (X,Y) and with height H and width W, an expression for the midpoint of the vector is $(X+W/2, Y+H/2)$. This expression calculates the midpoint regardless of updates to any of the underlying variables. We call such expressions *snap point expressions*.

Now that we have defined snap points, this approach starts with the *static* diagram that is provided by the author. For each snap point the algorithm now has two things: (1) as mentioned above, it has a snap point expression stating the snap point's coordinates in terms of control variables, for example $(X+W/2,Y+H/2)$ for the mid-point of a vector; (2) it also has the snap point's concrete coordinates in the static diagram, for example $(10,20)$.

This technique now searches the static diagram for *contact points*, which are two snap points that have the same concrete coordinates in the static diagram—in other words two snap points that are located at the same exact position. This algorithm assumes that a pair of overlapping snap points is not accidental, but rather indicates that the two shapes should be connected at that point. So, for each such pair of overlapping snap points in the static diagram, the point is considered to be a *contact point*, and this technique adds a constraint to keep the two snap points co-located by equating their (X,Y) snap point expressions.

2.4.2 Selecting Drag Points

After generating layout constraints, the next step is to select drag points, which are the points that viewers can drag around on the screen. A drag point is a snap point (as defined previously) which the author has decided to make interactive (recall that each shape has many snap points, as shown for example in Figure 2.4).

One approach for selecting drag points is to exhaustively search through all subsets of snap points, make them interactive drag points, prune non-sensical results, and then have the author go through these results. However, this approach is both expensive and likely to generate an overwhelming number of options, when in fact the author in most cases already knows precisely which points should be interactive.

Instead, we have decided to let the author make this decision upfront by stating directly which snap points to turn into drag points. The author can click on "Interactions", which displays all candidate points that can be made into drag points. The author can then click on the points that should be interactive. Figure 2.5 shows this interface after the author has selected three drag points, which have become green to indicate they are selected. At any point later in the editing,

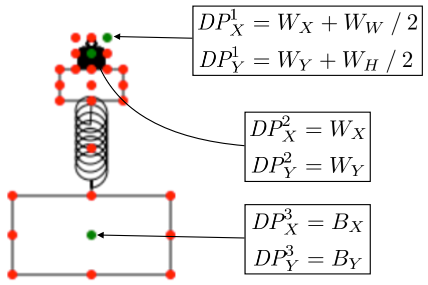


Figure 2.5. Drag points and their generated linear constraints.

the author can go back to this view and select additional drag points, or disable previously selected drag points.

USER-GUIDED INTERACTION SYNTHESIS now needs to generate layout constraints for the newly created drag points. For each snap point *P* that the author has selected to become a drag point, the technique creates control variables for the drag point's *X* and *Y* coordinates, and creates constraints equating those control variables to *P*'s snap point expressions. These constraints have two purposes. (1) When the underlying shape moves/resizes because of other changes in the diagram, the drag point also moves—consider for example a drag point in the corner of a box for resizing, which should move with the shape. (2) When the drag point is moved, it will have an effect on the shape itself.

2.4.3 Generating Interactivity Constraints

Now that the author has chosen drag points, what remains is to generate the constraints that govern the interactivity of these drag points. Recall that interactivity of drag points is governed by the free set of each drag point. This free set captures those variables which will be allowed to change when a given drag point is dragged. Thus, this algorithm must generate a free-set map *M* from drag points to sets of variables. Our general approach will again leverage human interaction: the engine generates free-set maps and orders them heuristically, and finally displays animated previews to the author showing what interactions look like under these different free-set maps. The author will then be able to quickly find their desired interactivity.

Pruning by Validity. First, we show how to significantly narrow down the search space of free-set maps before even displaying them to the author. We define a notion of validity for the free-set map, so that the search only needs to look at valid maps, instead of all maps. To understand where this notion of validity comes from, let's return to constraint solvers.

There are two situations where constraint solvers give results that are difficult to predict. The first is when the constraints are *underspecified*, in which case there are many solutions, and the constraint solver just picks one. The second is when the constraints are *overspecified*, in which case the constraint solver uses weights to figure out which constraints to violate. In both of these situations the results are hard to predict and are very sensitive to small changes in constraint weights. Instead, the synthesis algorithm will keep constraints *exactly specified*, where there is a single solution. The notion of validity for a free-set map *M* will *guarantee* that the constraints are always exactly specified, so that the solution depends only on the constraints, and not on internal details of the solver or brittle weights.

At first, one might think that there is a simple way to define validity, which is to require that the number of variables is equal to the number of constraints. Although in linear systems this guarantees no more than one solution (i.e., it avoids being underspecified), it doesn't guarantee a solution (i.e., it *does not* avoid being overspecified).

Instead, we define an algorithmic definition of validity. In particular, for each drag point and the free set in *M* for the drag point, the algorithm constructs the edit constraints that would be built at runtime if the drag point were dragged, and then uses a simple traversal through the constraints starting at the edited variables to make sure that all variables can be solved for exactly. To begin, the algorithm initially marks as *determined* all the variables that have changed (i.e., mentioned in edit constraints) and all other variables that are constrained to stay unchanged (i.e., mentioned in a stay constraint). Next, each time it sees a linear constraint where all but one of the variables is determined, the last variable is marked as determined (which works because linear constraints provide a unique solution for the last undetermined variable in an equation). We repeat this process until no more variables can be marked as determined. If during this process each variable is marked as determined *exactly once*, this drag point and its associate free set leads to a single unique solution when dragged. If this process succeeds for all drag points and their associated free sets in M, we say that M is valid.

Narrowing down to only valid maps is really important: for a drag point in our most complex benchmark, there are 28 shape control variables for roughly 268 million possible free-set maps but only 3 of these maps are valid. The above definition of validity can be applied directly using a naïve approach that enumerates all possible free sets for each drag point, and then applies the validity definition to only maintain valid maps. While this approach works, it is also very expensive and indeed, intractible in practice. We have developed an optimized dynamic-programming computation which builds the valid free sets bottom-up efficiently.

Preview Ranking and Visualization. Empirically the number of valid maps for a given drag point is relatively small (on average 14.5). As a result, for each drag point the algorithm generates all valid free sets and ranks the results with several heuristic aesthetics functions.

This ranking function encodes numerically the following four aesthetic observations: (1) changes in response to viewer interaction tend to involve only a handful of shapes, and so we favor interactions with fewer number of shapes; (2) shapes tend to respond to changes the same way in both dimensions, e.g., resizing in one dimension but translating (not resizing) in another is very unusual; (3) drag points on the corners of shapes tend to control stretching, while drag points in the middle of shapes tend to control translation; (4) drag points that move in one dimension are much less common than points that move in multiple dimensions.

Once the free sets are ranked, the top-ranking free set for each drag point becomes the default for that drag point. The author can now interact with the diagram in the viewer pane. If any drag point does not interact in the intended way, the author can right-click on the drag point and see an ordered list of all available interaction modalities. Each modality is previewed automatically through a self-animating diagram, showing the author what would happen if the drag point were moved in a circular motion. The author can flip through the different previews, and pick the one that captures the correct interactivity. Figure 2.1c in the Overview section depicts an automatically-animated preview. The preview adds an image of a hand to indicate exactly which part of the diagram is being dragged. There is also an "Accept" button (not shown) that allows the author to accept the current preview, and selection arrows (not shown) that allow the author to move between different previews.

By ordering the interactivity modalities heuristically, using the top-ranked modality as the default, and allowing the author to change modalities through previews, this technique leverages the computer's ability to quickly narrow down millions of modalities into a short ordered list, and also leverages the human's ability to quickly pick among a list of self-animating previews.

2.5 Prototyping and Evaluation

To evaluate USER-GUIDED INTERACTION SYNTHESIS, we built a diagram editor named EDDIE that implements our approach. We wanted EDDIE to be easily accessible and so implemented EDDIE as a client-side HTML5/JavaScript web application. For the static diagram editor, we slightly modified the FabricJS [2] library.

To demonstrate feasibility on a real-world application of interactive diagrams, we picked the domain of physics education. We evaluate EDDIE on diagrams in the Physics Education Technology (PhET) project, a library of interactive educational technical diagrams [100]. PhET has received many awards and accolades and has provided empirical evidence for its educational effectiveness [7, 90, 22, 10]. In addition to being interactive, PhET's diagrams include a simulation modelling the technical topic. As a proof of concept we extended EDDIE with a simple physics engine (and corresponding graphical primitives) to model a subset of PhET's physical diagrams.

We evaluate EDDIE along three dimensions: (1) *Expressiveness*, which looks at what kinds of interactive diagrams EDDIE can build (2) *Cost Savings*, which looks at how good EDDIE is at reducing a user's effort of authoring interactive diagrams and (3) *Usability*, which looks at how usable the tool is in practice. The first two dimensions (expressiveness and costs savings) are evaluated through a case study in which we ourselves implement several real-world diagrams. The third dimension (usability) is evaluated by asking thirteen educators to use EDDIE to build two PhET diagrams.

2.5.1 Case Study

In our case study, we looked at all interactive PhET physics diagrams for the following three topics: (1) simple spring mechanics, (2) gravity and planetary motion, and (3) pendulum motion. We discuss the physical and interactive properties of diagrams within each of these topics in turn.

- **Simple Spring Mechanics:** these diagrams study the dynamics of idealized, massless springs with one end fixed and the other subject to an applied force, for example a weight or another spring. The drag point is connected to the free end and controls the displacement of the spring. Springs can be connected end-to-end, in *series*, or side-by-side, in *parallel*. This category contains five diagrams: 1 spring, 2 springs, 2 series springs, 2 parallel springs, and a weight on a spring.
- **Gravity and Planetary Motion:** these diagrams study the force of gravity and its application to planetary motion. These diagrams have a number of spheres representing planets, as well as overlaid vectors representing velocity. PhET uses drag points in two ways: to determine the placement of a planet, and to determine the value of a velocity vector. This category contains four diagrams: a gravity lab, 2 planets, 3 planets, and 4 planets.
- **Pendulum Motion:** this diagram studies the motion of a simple pendulum subject to gravity and air resistence. The shapes consist of spheres representing the pendulum base and weight and a line between the spheres. A drag point controls the position (and by extension, angular displacement) of the pendulum weight. This category contains two diagrams, one with two pendulums and one with a single pendulum.

Across the three categories above, there are a total of 11 interactive diagrams. EDDIE is able to generate nine of these diagrams with very little human interaction, requiring at most five minutes of effort by the authors. Before we show the benefits that EDDIE provides in these nine diagrams compared to the traditional way of building these diagrams, we first explain the two diagrams that EDDIE cannot generate.

The first unsupported diagram is a series spring simulation in which the drag point simultaneously translates and compresses one of the springs. Our system limits updates to a single variable and so does not support this interaction. To support such interactions, we could extend EDDIE to use ranking functions on stay-constraints.

The second unsupported diagram is a spring simulation in which the viewer can drag-and-drop weights onto a platform attached to a spring. This action consists of two different layout configurations, one in which the weight is free-floating and one in which the weight is attached to the spring. Our framework currently assumes the layout constraints always hold for a given diagram and so can't support this functionality. To support such interactions, EDDIE would need to add support for conditional layout constraints and drag-and-drop features.

Cost Savings. We now evaluate the manual effort required by the author to make diagrams in EDDIE compared to the traditional approach. The baseline we use is the PhET implementation of the nine benchmarks we also implemented in EDDIE. In general, PhET benchmarks require a large amount of effort to build, and the builder needs to have a lot of programming experience. At the time of this writing, the project employs four full-time software developers [7]. Each diagram requires between 3800 and 5700 lines of handwritten JavaScript and Flash. Figure 2.6 shows the different benchmarks, along with the lines of codes required to implement each benchmark in PhET (column "PhET LoC").

In EDDIE, authoring a diagram requires two efforts. First, the author generates a static representation of the diagram. We have not formally quantified this effort, but most static diagrams have only a handful of shapes, and creating and aligning those shapes takes on the

Category	**Phet LoC**	**Name**	**P**	**V**	**VPP**
Springs	5278	1 spring	1	3	3
		2 springs	2	6	3
		parallel	1	8	8
Orbits/Gravity	5628	lab	2	4	2
		2 body	4	4	1
		3 body	6	6	1
		4 body	8	8	1
Pendulum	3821	1 mass	1	1	1
		2 masses	2	2	1

Figure 2.6. Comparison of programming effort between PhET and EDDIE for a variety of PhET physics diagrams. "P" is the number of points present in the diagram, "V" is the number of candidate interactions viewed by the user, and "VPP" is our metric of "Views-Per-Point", "V"/"P". EDDIE significantly reduces the effort of authoring diagrams, requiring one or two views per point for most diagrams.

order of minutes.

Second, the author specifies drag points and drag point interactions in the diagram. Figure 2.6 shows the effort required for this in three columns: "P" shows the number of drag points that the author must select; "V" shows the total number of previews (that the author must view across all drag points for that benchmark (note that the V count includes the view that the author accepts for each drag point, meaning that V cannot be smaller than P); "VPP" (which stands for *views per point)* shows the total number of previews that the author must view per drag point, in other words VPP = V/P (VPP cannot be smaller than 1).

The VPP metric captures the effectiveness of this technique's heuristic for ordering interactivity models (through ranking of free-set maps). A VPP of 1 corresponds to EDDIE always choosing the right interactivity model first, while a high VPP corresponds to showing many undesired interactivity models before the author finds the right one.

The VPP results demonstrate that EDDIE ranks interaction modalities well: in most cases, the correct interaction modality is in the top 3. Only one benchmark required 8 views per point, but even in this case, because the previews are self-animating, the viewer can very quickly browse through 8 previews, usually in less than one minute. As a whole, this case study demonstrates that EDDIE can express a variety of real-world diagrams and further, requires little effort by a power user (i.e., one of the authors of this paper).

2.5.2 Usability: Teacher Usage and Insights

To collect feedback and evaluate usability on the target audience of nonprogrammers, we introduced EDDIE during a higher-education computer science pedagogy class for K-12 teachers. We recruited eleven science teachers (three female) aged 25-55 and two male science professors aged 45-60. Participants were first given a brief 15-20 minute presentation on interactive diagrams and the features of EDDIE.

Next, the participants were given an instructional page about EDDIE and two interactive PhET diagrams to replicate within EDDIE. For each diagram, the organizer demonstrated the desired functionality of the specific diagram. The organizer did not directly aid the participants after participants started using EDDIE.

All participants replicated the two diagrams within an hour. The first diagram, consisting of a weight on the end of a spring [8], took the participants roughly 35-45 minutes to replicate. The second diagram was simpler, consisting of two pendulums hanging from a common pivot [6] and took roughly 5-20 minutes to replicate.

After completing both diagrams, the participants answered some qualitative questions about their experience using EDDIE. We asked participants for both positive and negative feedback about the tool, in a free form manner. We did not prompt the participants to discuss any particular feature of EDDIE, and we did not tell them what part of EDDIE was novel.

On the positive side, 8 out of 13 participants mentioned that EDDIE was easy to use and one explicitly mentioned that the side-by-side panes were useful for simultaneously displaying the author's and viewer's perspectives. Finally, 5 out of 13 participants mentioned that they liked EDDIE's static editing capabilities, which are mostly inherited from the existing FabricJS framework.

On the negative side, EDDIE performs a lot of work for the user and as a consequence several participants experienced a steep learning curve—3 out of 13 participants were at times surprised by the output of their actions. However, all three participants overcame the learning curve by referring to the help pages and all three completed both exercises. Furthermore, 3 out of 13 participants were at first confused by the functionality of the left/right buttons in the interface that enables users to select interaction modality. This confusion was resolved after viewing the help page and all three participants completed the diagrams. EDDIE's visualization and presentation of the list is tangential to our research contribution and was picked for ease of implementation. There are other mechanisms to make this interface clearer, for example displaying a swiping animation when previews are switched, replacing the left/right buttons with a single "next" button, or replacing the left/right buttons with a touch/swipe interface.

Finally, the participants used their own machines, which resulted in a wide variety of platforms (e.g., tablets, netbooks, and laptops) and web browsers (e.g., Firefox, Chrome, Safari, Internet Explorer, Edge). A significant number of these environments experienced some performance problems, for example lagging and jittery animated previews. Unfortunately, we did not discover these problems before the study because we had developed and tested EDDIE on a higher-end laptop, on which performance was not an issue. Still, despite these performance problems, EDDIE was ultimately usable and all participants finished their tasks quickly. Furthermore, we have not yet done any significant performance optimizations or extensive cross-browser testing. With further tuning and testing, we believe we can bring good performance to EDDIE across a wide variety of environments.

Insights and Lessons Learned. Through our usage and evaluation of EDDIE, we found that EDDIE's design works well for a variety of reasons.

First, the side-by-side view is useful because it gives immediate feedback on how generated diagrams work. Task-switching is well-known to require measurable cognitive effort [107]. EDDIE's side-by-side panes likely incur less cognitive load than alternating between two different views, because the panes in EDDIE are always visible at the same position.

Second, the self-animated previews are effective at quickly allowing authors to see different interaction modalities, but without having to do any actual manual interaction. This makes the selection process for interaction modalities very quick and easy to use.

Third, the approach of reducing the search space (by pruning underconstrained modalities) and then ordering the top candidates is effective because it narrows the author's attention to only the most promising interaction modalities. While developing EDDIE, prior to adding the ordering heuristics, we experienced frustration at having to navigate many incorrect interactions. After adding the aesthetic heuristics, the experience was much improved.

Limitations and Future Work. There are clear opportunities for future work on expanding our expressiveness and automation to get closer to the expressiveness of more manual tools like Kitty [63] or Apparatus [1], including: continuous locations for drag points, user-defined adjacency relationships, conditional relationships, and nonlinear relationships. Still, despite these limitations on expressiveness, our formalism of shapes and drag points connected by linear adjacency relationships can capture about 67% of the interactivity present in PhET diagrams, a real-world set of interactive diagrams that domain experts actually wanted to build.

In addition, our physics engine is relatively simplistic and as a consequence, there are many diagrams in PhET for which EDDIE can support the interactive portion but not the domain-specific chemical, mathematical, or physical components. To fully support these diagrams, an expert programmer would have to extend EDDIE's implementation by adding domain-specific primitives for these uncovered topics. Once this is done, diagram authors could then use the newly added primitives to reproduce these (currently unsupported) diagrams, without writing any additional code.

Summary. Our case study demonstrates that EDDIE can express and implement a broad variety of real-world diagrams. EDDIE only requires a handful of clicks to generate a complete interactive diagram and does not require any coding experience. In contrast, the original benchmarks require a large amount of programming expertise, which prevents many content experts from directly authoring diagrams. In addition, we demonstrated that EDDIE is usable by non-programming users and gathered some valuable target audience feedback about diagram construction using EDDIE.

2.6 Conclusion and Acknowledgements

We presented USER-GUIDED INTERACTION SYNTHESIS, a technique that transforms a static diagram into an interactive one without requiring any code to be written. We also presented an implementation in a tool called EDDIE, which we show is expressive and usable. By drastically reducing the cost of making interactive diagrams, this line of research opens up the possibility for experts who have domain knowledge (e.g., teachers who know about STEM) to build animated diagrams that they would otherwise not be able to build. This provides an exciting avenue not only for future research, but also for eventual impact on the adoption of interactive diagrams.

Funding: This work was funded by NSF grant 1423517.

This chapter, in full, is a reprint of the material as it appears in CHI Conference on Human Factors in Computing Systems 2017. Sarracino, John; Barrios-Arciga, Odaris; Zhu, Jasmine; Marcus, Noah; Lerner, Sorin; Wiedermann, Ben., ACM Press, 2017. The dissertation author was a primary investigator and author of this paper.

Chapter 3

Mockdown: Inductive Hierarchical Layout Synthesis

3.1 Introduction

Visual layout is the graphic design problem of arranging content on a visual medium. Example domains include document rendering, graphical user-interfaces, data visualization, etc. Layouts are frequently *dynamic* and must adjust to different configurations and sizes: for example, *responsive web design*[83] is a design methodology for designing dynamic web page layouts.

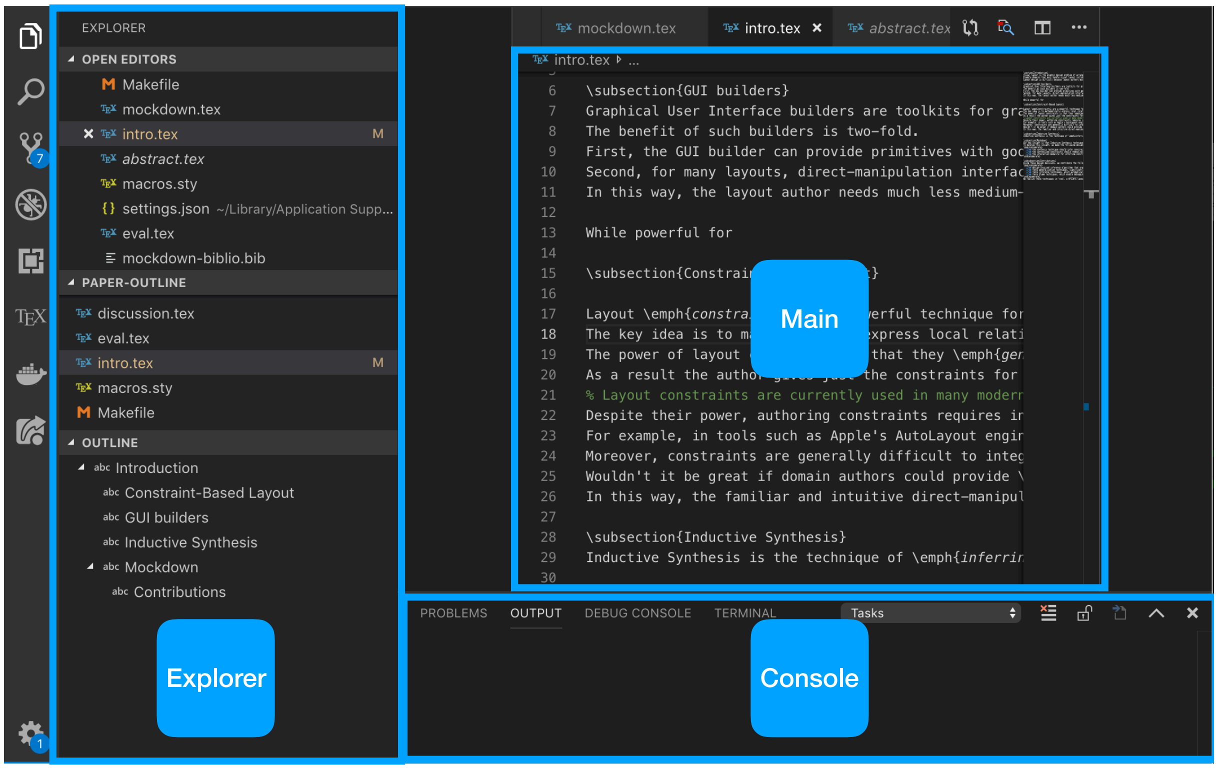
3.1.1 Constraint-Based Layout

Declarative constraint systems, such as Cascading Style Sheets (CSS) [79], iOS's Auto-Layout system [110], and Android's ConstraintLayout system [45], are a powerful and common technique for implementing dynamic layouts. The key idea is to mathematically express local relations between layout elements, such as "adjacent-to" or "child-of" as mathematical equations. The layout author gives just the constraints for a layout, and the rendering system figures out how the layout should *adapt* and *respond* when presented with different graphical settings.

Despite their power, constraint-based layouts are notoriously complicated to create and typically require difficult manual programming and configuration. Inductive program synthesis is a popular new field, which broadly is concerned with inferring a general program from several input-output examples of its behavior. Wouldn't it be great if we could synthesize a constraint-based layout from several examples? In this way the layout author does not need to manually fiddle with constraints and can simply provide several examples of how a layout should behave.

3.1.2 Motivating Problem

Consider the problem of authoring a layout for a code editor shown in Figure 3.1. This layout is dynamic in that it must adjust to different text content, as well as different overall window sizes. Although both of these problems can be handled by constraints, we focus on just the latter.



```
\subsection{GUI builders}
Graphical User Interface builders are toolkits for gra
The benefit of such builders is two-fold.
First, the GUI builder can provide primitives with goo
Second, for many layouts, direct-manipulation interfac
In this way, the layout author needs much less medium-

While powerful for

\subsection{Constrai                  t}

Layout \emph{constra              werful technique for
The key idea is to ma             express local relati
The power of layout               that they \emph{gen
As a result the author gives just the constraints for
% Layout constraints are currently used in many modern
Despite their power, authoring constraints requires in
For example, in tools such as Apple's AutoLayout engin
Moreover, constraints are generally difficult to integ
Wouldn't it be great if domain authors could provide \
In this way, the familiar and intuitive direct-manipul

\subsection{Inductive Synthesis}
Inductive Synthesis is the technique of \emph{inferrin
```

Figure 3.1. Annoted graphic depicting the layout of the VSCode text editor.

In addition, this layout is challenging because it has many different hierarchical components. The *Explorer* pane, highlighted on the left, contains a number of sub-panes each with dynamic content and size. The *Main* pane in the middle displays the actual code. Finally, the *Console* pane in the bottom displays terminal output. In this layout (taken from Visual Studio Code's `Centered Layout` [134]), the content of Main is centered within the root view.

This layout is conceptually straightforward but poses a significant challenge to inductive synthesis techniques due to its sheer complexity. There are many different layout elements, and moreover, the ideal program is relatively complex in that it must completely determine the position of each of the layout elements.

Our key insight is that while the overall layout might be complex, the layout is *nested* and *hierarchical* and each individual *layer* of the layout is amenable to inductive synthesis techniques.

3.1.3 Contributions

We develop this insight into the following contributions:

- An inductive synthesis tool termed MOCKDOWN which takes as input input-output examples of a layout and produces a dynamic constraint-based layout. We develop MOCKDOWN in section 3.2 and section 3.3.
- Several *correctness predicates* for statically ensuring that a dynamic layout will resize for any element of a particular set of top-level dimensions. We develop these predicates in subsection 3.3.2.
- An algorithm of scaling inductive layout synthesis to realistic layouts by leveraging the nested nature of the layouts. We term this algorithm HIERARCHICAL DECOMPOSITION and develop it in subsection 3.3.3.
- A case study in which we find that many popular web pages can be accurately modelled using linear constraints, presented in subsection 3.4.1
- An evaluation of MOCKDOWN and HIERARCHICAL DECOMPOSITION on DuckDuckGo, a popular search engine, in which we find that HIERARCHICAL DECOMPOSITION is necessary for traditional inductive synthesis to scale to realistic layouts, presented in

subsection 3.4.2

The rest of this chapter is structured as follows. In section 3.2 we define a core layout language for dynamic web layouts. Then, in section 3.3, we develop synthesis algorithms for building a dynamic layout from input-output examples of the layout's behavior. Finally in section 3.4 we evaluate HIERARCHICAL DECOMPOSITION through several case studies using MOCKDOWN, and we develop some takeaways and conclude.

3.2 Dynamic Layouts Using Linear Constraints

We first describe our core layout model and how to represent dynamic layouts using sets of linear equations.

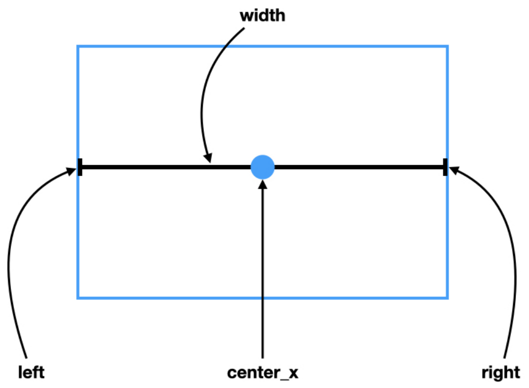
3.2.1 Core Layout Language

We represent visual layouts using a single type of element, rectangles. Although primitive and coarse, this is expressive enough to model (to a first approximation) many realistic layouts. We depict this layout model in Figure 3.2 and also give an annotated example from the DuckDuckGo search engine in Figure 3.2c.

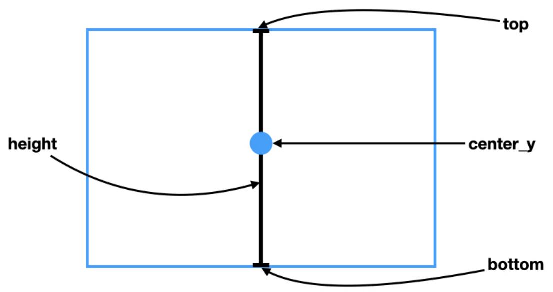
In our model, each element of a layout has eight geometric attributes, termed *anchors*: top, left, right, bottom, width, height, center_x, and center_y. These are standard. We assume that all input layouts are well-formed (i.e. the concrete values of the anchors are consistent with the geometric relationships in Figure 3.2a and Figure 3.2b).

In addition our layouts are also *nested* and *hierarchical*. We model this with an explicit `children` function which takes as input a layout element and returns a list of the children of the input. We again assume that input layouts are well-formed.

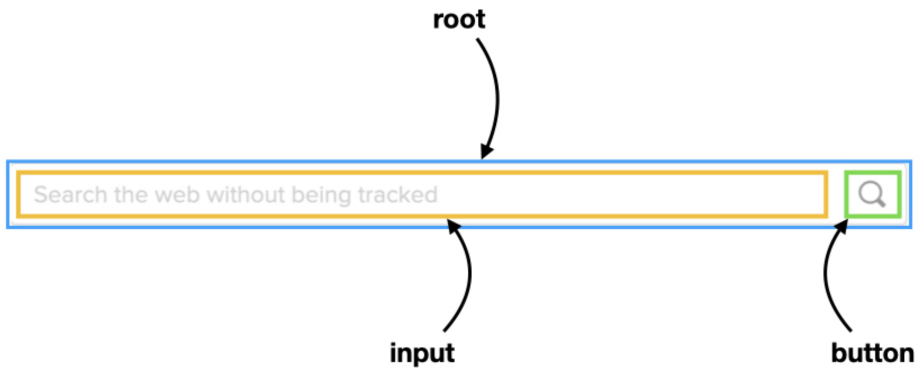
We say a *Model* is a map from anchors to floating point values. If a Model is defined over all of the anchors in a layout, then the visual rendering of the layout can be directly calculated from the values of the Model. In this way our dynamic layout algorithm is to calculate a new Model for the layout given dimensions for the new *root* geometry.



(a) Horizontal layout anchors.



(b) Vertical layout anchors.



(c) Annotated DuckDuckGo search bar example.

Figure 3.2. Anchors in our layout model and DuckDuckGo search bar running example.

3.2.2 Dynamic Layouts

Our layout algorithm is based on linear constraint solvers. A linear constraint solver takes a set of linear equations (termed a *system of constraints*), and either produces a Model that conforms to the equations (termed a *solution*) or throws an error[1].

In MOCKDOWN we use linear constraints in the following way, depicted by example in Figure 3.3. First, we say a *layout equation* is a linear constraint of the form $\mathbf{y} = a \cdot \mathbf{x} + b$ where a and b are (possibly 0 or 1) constants and $\mathbf{y}$ and $\mathbf{x}$ are layout anchors. Then, we represent the layout's geometric structure using a system of layout equations, and we represent the desired root element's dimensions using a set of constant constraints termed *Edit constraints*. Finally, we

[1]It turns out that even overdetermined systems of equations can be solved using a technique called *constraint priorities*. For a more detailed description, see subsection 2.3.1

combine the layout equations with the edit constraints and geometric axioms (which encode the geometric relationships of the layout anchors) into a query for a linear constraint solver. So long as we ensure that the layout equations are solveable, the solver returns a Model that represents the concrete layout values after resizing the root element.

$$\begin{array}{cccc}
input.left = root.left + 5 & button.left = input.right + 5 & button.right = root.right - 5 & button.width = 20 \\
input.center\_y = root.center\_y & button.center\_y = root.center\_y & input.height = 20 & button.height = 20
\end{array}$$

(a) Layout equations for the running example.

Element	**Axioms**	
root	$root.center\_x = root.right - root.left$	$root.center\_y = root.bottom - root.top$
	$root.width = \frac{root.left + root.right}{2}$	$root.height = \frac{root.top + root.bottom}{2}$
input	$input.center\_x = input.right - input.left$	$input.center\_y = input.bottom - input.top$
	$input.width = \frac{input.left + input.right}{2}$	$input.height = \frac{input.top + input.bottom}{2}$
button	$button.center\_x = button.right - button.left$	$button.center\_y = button.bottom - button.top$
	$button.width = \frac{button.left + button.right}{2}$	$button.height = \frac{button.top + button.bottom}{2}$

(b) Layout axioms for the running example.

Edit constraints	
$root.left = 0$	$root.right = 0$
$root.width = 200$	$root.height = 300$

Element	**Model**			
root	$root.left \mapsto 0$	$root.right \mapsto 200$	$root.width \mapsto 200$	$root.center\_x \mapsto 100$
	$root.top \mapsto 0$	$root.bottom \mapsto 300$	$root.height \mapsto 300$	$root.center\_y \mapsto 150$
input	$input.left \mapsto 5$	$input.right \mapsto 170$	$input.width \mapsto 165$	$input.center\_x \mapsto 87.5$
	$input.top \mapsto 130$	$input.bottom \mapsto 170$	$input.height \mapsto 20$	$input.center\_y \mapsto 150$
button	$button.left \mapsto 175$	$button.right \mapsto 195$	$button.width \mapsto 20$	$button.center\_x \mapsto 185$
	$button.top \mapsto 130$	$button.bottom \mapsto 170$	$button.height \mapsto 20$	$button.center\_y \mapsto 150$

(c) Edit constraints and solver model for resizing the *root* width and height to $(200, 300)$.

Figure 3.3. Calculating a dynamic layout for the DuckDuckGo example of Figure 3.2c using layout constraints, axioms, and edits.

This process is well-understood in the community and is not novel. Now that we can dynamically resize layouts, we turn to the problem of *synthesizing* a system of layout constraints from examples of the layout.

3.3 Synthesizing Layout Constraints

In this section we develop several methods for synthesizing a dynamic layout from examples. First, in subsection 3.3.1 we generate a set of candidate equations using templates, in a manner inspired by Daikon [35]. Next, in subsection 3.3.2 we combine the candidates with several correctness predicates that require the output to behave properly accross all possible input solutions. Next, we discharge the overall formula as a maximum-satisfiable subset (MSS) query[2] [81] to an off-the-shell SMT solver (in our implementation Z3 [33]); this results in a set of constraints that determine a valid layout for all possible configurations of the root layout element. Finally, although this process is conceptualy elegant and valid, it does not scale to real-world layouts, and we conclude by describing an algorithm for HIERARCHICAL DECOMPOSITION that drastically reduces the complexity of each MSS instance.

3.3.1 Template Instantiation

In the first phase of our algorithm we enumerate a number of candidate layout constraints. The grammar of our layout constraints is given in Figure 3.4. These constraints encompass a broad variety of common layout relationship and are mostly straightforward, but there are two subtleties with the constraint terms.

Separate dimensions. In all cases the constrained anchors are either both horizontal or both vertical. This is important because it enables our MSS query implementation to solve for the horizontal and vertical dimensions in separate queries.

Single constant variable. All of the constraints have exactly one free constant variable in the constraint. This is important because it enables our constraint generation algorithm to instantiate templates from a single input example.

[2]A MSS problem is a particular type of optimization problem in which the constraint solver gives a satisfiable subset of boolean values maximized according to an objective function. Many popular SAT/SMT solvers support this query.

$$
\begin{array}{lcl}
 & & \texttt{child}, \texttt{parent}, \texttt{lhs}, \texttt{rhs} \in \mathit{View},\ a, b \in \mathbb{R} \\
\text{LayoutConstraint} & ::= & \text{ParentRelative} \ \mid\ \text{SiblingRelative} \ \mid\ \text{Absolute} \\
\text{ParentRelative} & ::= & \texttt{child}.\mathit{left} = \texttt{parent}.\mathit{left} + b \\
 & \mid & \texttt{child}.\mathit{right} = \texttt{parent}.\mathit{right} + b \\
 & \mid & \texttt{child}.\mathit{center\_x} = \texttt{parent}.\mathit{center\_x} + b \\
 & \mid & \texttt{child}.\mathit{width} = a \cdot \texttt{parent}.\mathit{width} \\
 & \mid & \texttt{child}.\mathit{top} = \texttt{parent}.\mathit{top} + b \\
 & \mid & \texttt{child}.\mathit{bottom} = \texttt{parent}.\mathit{bottom} + b \\
 & \mid & \texttt{child}.\mathit{center\_y} = \texttt{parent}.\mathit{center\_y} + b \\
 & \mid & \texttt{child}.\mathit{height} = a \cdot \texttt{parent}.\mathit{height} \\
\text{SiblingRelative} & ::= & \texttt{lhs}.\mathit{left} = \texttt{rhs}.\mathit{left} + b \\
 & \mid & \texttt{lhs}.\mathit{left} = \texttt{rhs}.\mathit{right} + b \\
 & \mid & \texttt{lhs}.\mathit{right} = \texttt{rhs}.\mathit{left} + b \\
 & \mid & \texttt{lhs}.\mathit{right} = \texttt{rhs}.\mathit{right} + b \\
 & \mid & \texttt{lhs}.\mathit{center\_x} = \texttt{rhs}.\mathit{center\_x} + b \\
 & \mid & \texttt{lhs}.\mathit{top} = \texttt{rhs}.\mathit{bottom} + b \\
 & \mid & \texttt{lhs}.\mathit{bottom} = \texttt{rhs}.\mathit{top} + b \\
 & \mid & \texttt{lhs}.\mathit{center\_y} = \texttt{rhs}.\mathit{center\_y} + b \\
\text{Absolute} & ::= & \texttt{child}.\mathit{width} = b \\
 & \mid & \texttt{child}.\mathit{height} = b
\end{array}
$$

Figure 3.4. Syntax for layout constraints.

Constraint Generation

In order to produce layout constraint terms, we use the set of input examples as a source of constraint templates. Our algorithm is conceptually similar to Daikon [35] but the key difference is that we can constrain the space of templates by using the concrete hierarchy and layout of the examples.

In particular we use a geometric adjacency algorithm to eliminate redundant templates and only create templates between elements that are directly adjacent. In addition we use the hierarchy to restrict constraint variables in the following way: in Figure 3.4, when `child` and `parent` are in a production rule, we only generate a template using `parent` and `rhs` when `parent` is a parent of `child` in the concrete example. Similarly when `lhs` and `rhs` are in a rule we only generate a template for `lhs` and `rhs` when they are siblings of the same common parent.

3.3.2 Synthesis Algorithm

Now that we have candidate constraints, our next task is to assemble a subset of constraints that generalizes to other possible top-level dimensions. We take a MSS approach and translate the constraints into a logical formula[3]. In this way, we can leverage the power of SAT/SMT solvers to produce a solution. To do this, we need to express the logic of layout boxes in a logical formula, translate the constraints, and define a notion of correctness.

Our overall synthesis equation is given in Figure 3.6b and we use some helper functions (given in Figure 3.5). Notice that this is similar to the approach taken by InferUI[19]; we term these techniques *symbolic inference* because they calculate layout constraints using a symbolic solver and rich correctness properties.

$$\begin{aligned}
&\texttt{items} && :: \textit{View} \to \textit{List View}\\
&\texttt{items}(box) && = \texttt{children}(box) +\!\!+\ \texttt{flatmap items children}(box)
\end{aligned}$$

(a) `items` function definition: $\texttt{items}(box)$ returns a list of all of the inner elements of box.

$$\begin{aligned}
&\texttt{anchors} && :: \textit{View} \to \textit{List Variable}\\
&\texttt{anchors}(box) && = [box.left, box.right, box.top, box.bottom, box.width, box.height, box.center\_x, box.center\_y]
\end{aligned}$$

(b) `anchors` function definition: $\texttt{items}(anchors)$ returns a list of all of the anchors (control variables) of box.

$$\begin{aligned}
&\texttt{placed} && :: \textit{View} \to \textit{Location} \to \textit{Expression}\\
&\texttt{placed}(box,(top,left,bottom,right)) && = box.top = top \land box.left = left \land box.bottom = bottom \land box.right = right
\end{aligned}$$

(c) `placed` function definition: $\texttt{placed}(box, loc)$ returns a logical expression that specifies that box's concrete location is loc.

$$\begin{aligned}
&\texttt{axiom}_\texttt{x} && :: \textit{View} \to \textit{Expression}\\
&\texttt{axiom}_\texttt{y}(box) && = box.width = box.right - box.left \land box.center\_x = \frac{box.left + box.right}{2}\\
&\texttt{axiom}_\texttt{y} && :: \textit{View} \to \textit{Expression}\\
&\texttt{axiom}_\texttt{y}(box) && = box.height = box.bottom - box.top \land box.center\_y = \frac{box.bottom + box.top}{2}\\
&\texttt{pos} && :: \textit{View} \to \textit{Expression}\\
&\texttt{pos}(box) && = \textstyle\bigwedge_{var \in \texttt{anchors}(box)} var \geq 0
\end{aligned}$$

(d) Layout axiom function definitions: each function returns an Expression encoding layout axioms. $\texttt{axiom}_\texttt{x}$ and $\texttt{axiom}_\texttt{y}$ return horizontal and vertical axioms, respectively, and `pos` requires layout variables to be positive.

Figure 3.5. Helper functions for describing layouts.

Towards this end we formalize two correctness predicates: *Solvability*, which guarantees

[3]Conveniently, we bias the synthesis search towards likely solutions by using the MSS objective function to prioritize candidate constraints with clean, low-denominator constant terms.

$$\texttt{layout\_axioms}(root) = \bigwedge_{box \in \texttt{items}(root) + [root]} \texttt{axiom}_{\texttt{y}}(box) \land \texttt{axiom}_{\texttt{x}}(box) \land \texttt{pos}(box)$$

(a) Layout synthesis axioms.

$$\texttt{synth}(root, dims, constraints) = \texttt{layout\_axioms}(root) \land \bigwedge_{c \in constraints} [\![\, c \,]\!] \land \texttt{correct}(root, dims)$$

(b) Layout synthesis equation.

Figure 3.6. Axioms and synthesis equation for layout synthesis.

that the constraints actually do have a solution when solved; and *Unambiguity*, which guarantees that the constraints force the inner layout elements to have a unique placement for a particular top-level dimension.

$$\texttt{solv}(root, dims) = \forall dim_r \in dims \,.\, \texttt{placed}(root, dim_r) \implies \forall elem \in \texttt{items}(root) \,.\, \exists dim_e \,.\, \texttt{placed}(elem, dim_e)$$

(a) Solvability predicate

$$\begin{aligned}\texttt{unamb}(root, dims) = \quad & \forall dim_r \in dims \,.\, \texttt{placed}(root, dim_r) \implies \forall elem \in \texttt{items}(root), \forall dim_{e1}, dim_{e2} \in \mathbb{R}^4 \,. \\ & \texttt{placed}(elem, dim_{e1}) \land \texttt{placed}(elem, dim_{e2}) \implies dim_{e1} = dim_{e2}\end{aligned}$$

(b) Unambiguous predicate

$$\texttt{correct}(root, dims) = \quad \texttt{solv}(root, dims) \land \texttt{unamb}(root, dims)$$

(c) Overall correctness predicate

Figure 3.7. Correctness predicates for layout synthesis.

Solveable

A set of constraints is *Solveable* if for all possible dimensions of the root layout element, there is a placement of inner elements that conforms with the constraints. We formalize this predicate in Figure 3.7a. Since solvability corresponds to logical satisfiability, we can directly translate this predicate to a MSS query; the set of constraints in the solution is guaranteed to be solveable.

In practice Solvability is very important because in the absence of priorities, dynamic constraint solvers will crash when asked to solve a system that does not contain a solution. Moreover it does not prohibit solutions that are ambiguous. Consider for example $x > 0$. This is

Solveable, but is not necessarily useful because it gives very little insight about x (other than that x is positive); as a consequence 0.1, 1, and 10 are all valid solutions for x.

To address this we rule out *ambiguous* systems of constraints through an Unambiguous predicate.

Unambiguous

A set of constraint is *Unambiguous* if solutions to it are unique. More formally, for all possible dimensions of the root layout element, if there is a placement of inner elements that satisfies the constraints, there cannot be other inner element placements that also satisfy the constraints. We formalize this predicate in Figure 3.7b.

Approximation

Although we have defined layout synthesis, in practice this approach does not scale to realistic layouts. In particular the quantified correctness predicates of Figure 3.7 are intractable. To make headway on this problem, we find *approximate* solutions for solvability and unambiguity.

Solvability. Solvability is easily approximated by dividing the continuous top-level range of dimensions *dims* into several discrete points. In this way, we can create several discrete versions of the problem and find a maximum-satisfiable subset of constraints over all of the subproblems. In practice we found that only a few discrete samples (i.e. five) were needed.

Unambiguity. Unambiguity is harder to approximate because it does not directly translate to a MSS instance. Instead, we must perform explicit search over the space of MSS solutions, check each solution for ambiguity, and repeat until we find a solution that is not ambiguous. We accelerate and bias the search using several heuristics inspired by best-practices for constraint programming (given in Figure 3.8), but it remains an open problem efficiently search this space. Several areas of future work are to use an efficient linear constraint solver (such as Cassowary) to do the checks, to implement an efficient MSS search algorithm (such as MARCO [80]), or to formalize a set of efficient axioms that together imply unambiguity.

Parent-child linked: Constraints that are between a parent and a child are preferable.

Exactly determined dimensions: The layout equations should constrain exactly two of the four anchors in each dimension (i.e. in the horizontal dimension two of left, right, center, and width should be constrained).

Uniquely determined dimensions: Each anchor should be constrained by at most one equation.

Figure 3.8. Heuristics for accelerating unambiguity subset search.

Scaling Synthesis. At the end of the day, though, even with these approximations, the synthesis equation of Figure 3.6b does not scale to realistic problems. The chief issue is the sheer complexity of realistic layouts. Although they might seem small and simple at face value, because the underlying search is exponetial, even a layout with twenty elements poses a challenge for symbolic inference. To address this state space problem, we leverage the nested nature of many layouts to accelerate and scale symbolic search.

3.3.3 Hierarchical Decomposition

Many layouts are complicated and large, but are also subdivided in a natural, hierarchical manner. A common design pattern is to create layers, such that the positioning of a parent is independent from the positioning of its children. Our key insight is that this hierarchical division enables a much more tractable synthesis algorithm, termed HIERARCHICAL DECOMPOSITION: instead of attempting to solve for an optimal set of layout equations in one large query, we leverage the nested layers of the problem and generate a set of layout equations at each layer.

We formalize HIERARCHICAL DECOMPOSITION in Figure 3.9. There are three chief subtleties of this algorithm and we discuss each in turn.

Restricted synthesis quantification. Recall that in the synthesis equations, the predicates quantify over all of the elements of the root view. Now we must modify the equations to quantify just over the direct children of a particular box. This is accomplished in a straightforward manner by simply changing uses of `items`(*root*) to `children`(*root*) in Figure 3.7 and Figure 3.6a.

Subproblem specifications. Because the synthesis equations are parameterized over a root element and possible dimensions, we must calculate possible dimensions for the recursive

```
procedure SynthHier(root, constraints, dims)
    Initialize $worklist$ to [], $output$ to $\{\}$.
    Add $(root, dims)$ to $worklist$.
    while $worklist$ is nonempty do
        $(focus, dims_f) \leftarrow$ the top of $worklist$.
        $constraints_f \leftarrow \texttt{restrict}(focus, constraints)$.
        $output_f \leftarrow \texttt{MaxSAT}(focus, dims_f, constraints_f)$.
        Add $output_f$ to $output$.
        for $child \in \texttt{children}(focus)$ do
            $dims_c \leftarrow \texttt{calc\_dims}(focus, dims_f, child, output)$
            Add $(child, dims_c)$ to $worklist$.
        end for
    end while
    return $output$.
end procedure
```

Figure 3.9. Psuedocode for hierarchical decomposition.

subproblems. To do this we again turn to the MSS capabilities of modern SMT solvers, which enable a query to *maximize* and *minimize* an objective function. Whereas before our objective function was the best set of constraints subject to some notion of niceness of constraints, now our objective function is simply the dimensions of the child.

Relevant constraint focusing. For each particular layer, we must restrict the possible constraints to just those that are relevant to the layout of the layer. This is done by inspecting the syntax of the constraints and restricting the search to parent-relative constraints between the focus and the layer elements, sibling-relative constraints between elements of the layer, and absolute constraints over elements of the layer.

And that's it! Now that we have several candidate algorithms for inferring dynamic layouts, we experimentally evaluate their efficacy and applicability through several studies.

3.4 Evaluation

In this work we are motivated by 4 overall research questions:

(RQ1) Are common websites expressible using linear constraints?

(RQ2) Is symbolic inference useful for automatically calculating layouts?

(RQ3) Do traditional semantic techniques scale to realistic layouts? Does HIERARCHICAL DECOMPOSITION improve performance?

(RQ4) Do disambiguation predicates improve the quality of output solutions?

We quantitatively and quantitatively test these questions by examing a variety of layouts in real-world websites. To do so, we use a web browser to open the website and then capture the web browser's rendering of the website using the web browser's console, JavaScript, and the DOM API [].[4].

We conducted two studies. To answer RQ 1, we performed a survey in which we evaluated a variety of common, popular websites for linearity. To answer RQ 2, RQ 3, and RQ 4, we performed a case study in which we examined a single website in depth. Overall we find strong evidence for RQ 2, RQ 3, and we find significant evidence for RQ 4 and RQ 1.

3.4.1 Linearity Survey

In our linearity study, we considered the top 10 websites on Alexa's website rankings [124]. As of June 18 2020, these websites were: 1. Google [128], a search engine; 2. Youtube [136], a video upload and streaming service; 3. TMall [133], a retail website; 4. Facebook [127], a social media website; 5. QQ [132], an instant-messanger client and and web portal; 6. Baidu [125], a search engine; 7. Sohu [130], a web portal; 8. Taobao [131], a retail website; 9. Qihoo 360 [129], an internet security company; 10. Yahoo [135], a search engine and web portal. In total the Alexa websites comprise a variety of interesting layouts for a broad sample of applications. We found that half of the websites allowed external JavaScript to run from the browser console and were amenable to our quantitative evaluation. With the exception of Facebook, the only social media website, the remaining websites did not appear significantly different.

[4]Our instrumentation method is rather adhoc and indeed didn't succeed for all websites. Several of the websites were not amenable to this study, possibly due to implementation error, website-specific modifications to the DOM API, or other confounding factors.

Linear Layout Model

To answer RQ 1, we developed a general linear model for predicting a layout model given the top-level dimensions. From a rendering perspective the x- and y-coordinates of the top-level root element are fixed, so as the layout is resized, the independent variables of the system are the top-level height and width.

We model and fit each layout variable as a first-order polynomial of the top-level height and width, and the results are shown in Table 3.1. For each website, we took 10 different samples of the website and we measure four metrics:

# Elements: the total number of layout elements. This includes leaves and the root.

Avg Children: the average number of children for non-leaf layout nodes. This measures how nested and hierarchical the website is and corresponds to the average branching factor of the layout tree. A high value indicates a flat website; while a low value indicates a nested website.

Avg Error: the Root-Mean-Square error between the linear fit and the sample, averaged accross all samples. A high value (e.g. 100) indicates a website that is generally not linear, while a low value (e.g. 0) indicates a website that generally is linear.

Max Error: the maximum residual error between the linear fit and the sample, maximized accross all samples.

We chose Root-Mean-Square (RMS) error as an error metric because we wish to test the utility of linear polynomials. Because the errors are squared, this metric tends to favor outliers and in practice tends to be high when the prediction function is even slightly incorrect.

Table 3.1. Complexity and linear function fit for Alexa top 10 websites. Five of the top 10 websites are not amenable to instrumentation and have the error reported as "–". DuckDuckGo is not on the Alexa top 10 and is included for reference.

Website	# Elements	Avg Children	Avg Error	Max Error
Google	36	1.75	0.0	0.0
Youtube	826	1.55	0.0	0.0
TMall	89	2.26	0.14	10.88
Facebook	2130	1.54	–	–
QQ	1124	2.07	0.14	16.40
Baidu	58	2.48	0.57	32.00
Sohu	1210	2.05	–	–
Taobao	958	2.22	–	–
360	476	1.88	–	–
Yahoo	1209	2.20	–	–
DuckDuckGo	76	2.42	0.00	0.01

There are several interesting takeaways from this study. First, the complexity of websites varies very significantly. Many popular websites have less than one hundred distinct elements, while others are on order of several thousand. However all websites are roughly similar in terms of the hierarchical structure of the layout tree; most websites have around 2 children per element.

Second, on average, most website elements are well-modeled by a linear function. The average RMS error is extremely low for all websites, less than 1 in all cases, and provides strong evidence for RQ 1[5].

However, several websites contain elements that are not linear, with relatively large maximum errors. Several potential sources of nonlinear behavior are conditional formats and

[5]In practice we found that completely incorrect layouts had an RMS in the 1000s, and layouts that were slightly off ranged from 5 to 50.

element resizing based on text contents. We did not inspect these errors in detail. During the survey, we did not visually see any nonlinear behavior, so further inspection is needed to suss out these particular outliers.

The Case for Local Constraints

The linearity results are convincing and an inquisitive reader might ask, “Why not use such a model for all layouts? Why is it necessary to synthesize local constraints?” Although the performance of these models is tempting, they have two main weaknesses.

First, because the model has two independent variables (the top-level height and width) for each dependent layout variable, the layout author must provide at least three input-output examples of the layout’s behavior. As a consequence, for the problem of inference from one or two examples, we must develop more sophisticated techniques.

Second, because the model’s equations are inferred with respect to the root variables, it’s extremely common for the resulting layout equations to not be human readable or reusable. For example here are the inferred equations for the x- and y-coordinates of DuckDuckGo’s search bar:

$$\begin{aligned}\texttt{search\_bar.left} &= 0.59 \cdot \texttt{root.width} - 0.16 \cdot \texttt{root.height} + 0.0 \\ \texttt{search\_bar.top} &= -0.04 \cdot \texttt{root.width} + 0.06 \cdot \texttt{root.height} + 0.0\end{aligned}$$

In contrast, the true (relative) equations are given by

$$\begin{aligned}\texttt{search\_bar.center\_x} &= \texttt{logo.center\_x} \\ \texttt{search\_bar.top} &= \texttt{logo.bottom} + 40\end{aligned}$$

Qualitatively these local, relative equations are much more readable and reusable, and it's important for synthesis work to produce results that are consumable by end-users. To this end, we next examine how symbolic inference performs (which produces human-readable constraints).

3.4.2 DuckDuckGo Case Study

For our case study, we examined the DuckDuckGo search engine [] in detail, a screenshot of which is displayed in Figure 3.10.

DuckDuckGo's layout includes several interesting subcomponents. In particular, the layout contains a top-bar with several options, a centered logo, a search bar with an aligned button, and several centered informational boxes below the search bar. As shown in Table 3.1, DuckDuckGo is a representative website layout because both the size and branching factor are comparable with other websites.

We implement three different layout inference algorithms:

Everything: this algorithm represents a greedy approach, potentially taken by conventional user-interface builder, in which the inference takes all of the possible relationships present in the layout (as produced by subsection 3.3.1). Because many of our layout templates conflict with each other, the result is usually overconstrained. As a consequence we configure the linear constraint solver's priorities such that all constraints have equal weight, and let the constraint solver dynamically resolve the conflicting constraints.

Flat: this is the synthesis algorithm of subsection 3.3.2 that does not make use of a hierarchical decompositions during inference. It converts the entire nested layout to a single correctness formula and discharges the formula to Z3.

Hier: this is HIERARCHICAL DECOMPOSITION, the synthesis algorithm of subsection 3.3.3, which hierarchially decomposes the layout into separate subproblems. Each layer of the diagram is inferred separately and the resulting constraints are combined in the end.

For each of these algorithms, we measure how long the algorithm takes to infer constraints for the layout (Synth Time), the number (Output Size) and runtime performance (Solve Time) of constraints produced by the algorithm, and the mean RMS error of the output on items in the test set (Avg Error). The runtime performance is further separated into the preparation and resize times: the preparation time is how long it takes to add all of the layout constraints, and the resize time is how long it takes the layout constraints to adapt to a new root dimension.

Table 3.2. Performance of various synthesis algorithms with one training example, on DuckDuckGo. The metrics are: synthesis time (Synth time, in seconds), number of output constraints (Output Size), average dynamic preparation and resize execution time (Solve time, in seconds), and average root-mean-square error (Avg Error). XXX indicates that the algorithm did not finish in 30 minutes. Algorithms denoted with $*$ have annotated horizontal constraints for a single layout element.

Algorithm	Synth Time	Output Size	Solve Time	Avg Error
Everything	6.65s	719	1.91s, 0.28s	17.51
AmbigFlat	120.30s	548	0.74s, 0.01s	14.56
AmbigHier	19.79s	525	1.19s, 0.01s	15.68
AmbigFlat*	127.40s	552	1.32s, 0.02s	3.69
AmbigHier*	18.6s	524	0.88s, 0.03s	3.69
UnambigFlat	XXX	XXX	XXX	XXX
UnambigHier	332.48s	224	0.38s, 0.02s	15.23
UnambigFlat*	XXX	XXX	XXX	XXX
UnambigHier*	382.91s	224	0.41s, 0.02s	3.69

Table 3.3. Performance of various synthesis algorithms with two training examples, on DuckDuckGo.

Algorithm	Synth Time	Output Size	Solve Time	Avg Error
Everything	13.82s	665	1.48s, 0.21s	0.44
AmbigFlat	126.59s	547	0.79s, 0.02s	0.11
AmbigHier	26.29s	519	1.34s, 0.04s	0.11
AmbigFlat*	122.45s	547	0.90s, 0.02s	0.11
AmbigHier*	29.61s	519	1.15s, 0.02s	0.11
UnambigFlat	XXX	XXX	XXX	XXX
UnambigHier	422.04s	224	0.40s, 0.03s	0.14
UnambigFlat*	XXX	XXX	XXX	XXX
UnambigHier*	387.88s	224	0.32s, 0.01s	0.10

To evaluate the disambiguation predicates from subsubsection 3.3.2, we further modified the Flat and Hier algorithms by implementing versions with the predicate, titled UnambigFlat and UnambigHier, and without the predicate, titled AmbigFlat and AmbigHier. We conducted this experiment with one and two training examples and the results are tabulated in Table 3.2 and Table 3.3.

Takeaways

Overall we find strong support for RQ 2 and RQ 3, and we find evidence for RQ 4 (although the case is not as strong). We discuss the evidence for each in turn.

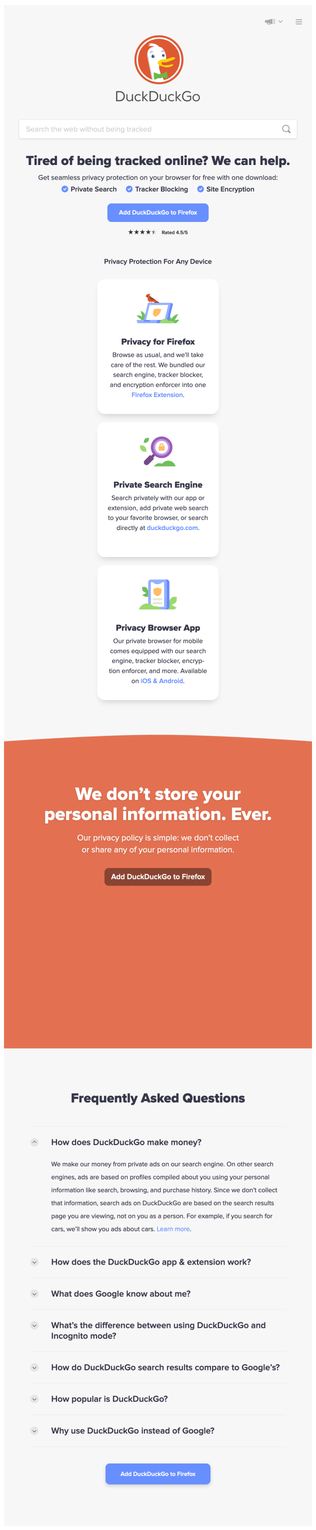
For RQ 2, whether symbolic inference is useful or not, we compare the Everything solver against the other algorithms. While the Everything solver finds a solution much faster than the others (as expected, because the template instantation algorithm does not perform much search), in all other regards the solution is inferior. The system produced by Everything is largest, slowest, and has the highest error in both the one-example study. In the two example study, while Everything still performs the worst the difference is much less pronounced. This is because the template instantiation algorithm discards hypothetical constraints that are not observed in both examples, and so discards many constraints that seem fine for one example but are not observed between examples.

For RQ 3, on the impact of HIERARCHICAL DECOMPOSITION, we conclusively find that HIERARCHICAL DECOMPOSITION scales up the search process. Without disambiguation predicates, Hier is roughly six times as fast as Flat, and with disambiguation predicates, Hier finds a solution while Flat times out after thirty minutes. In all cases the size, performance, and error between Hier and Flat are comparable.

Finally, for RQ 4, on the impact of disambiguation predicates, our findings are mixed. On the one hand, the predicates improve the quality of the output solution. In all cases UnambigHier's output constraint set is smaller, faster, and has lower error than the corresponding AmbigHier output. However, the downside is that disambiguation predicates incurr a signficant performance penalty on the search. The Flat algorithm is unable to finish with disambiguation predicates and the Hier algorithm is between thirteen and eighteen times slower. Overall we find that the disambiguation predicates are useful but expensive.

Acknowledgements

This chapter in part is currently being prepared for submission for publication of the material. Sarracino, John; Lukes, Dylan; Coleman, Cora; Weingarten, Ariel; Lerner, Sorin; Polikarpova, Nadia. The dissertation author was a primary investigator and author of this material.


DuckDuckGo

Search the web without being tracked

Tired of being tracked online? We can help.

Get seamless privacy protection on your browser for free with one download:
Private Search Tracker Blocking Site Encryption

Privacy Protection For Any Device

Privacy for Firefox

Browse as usual, and we'll take care of the rest. We bundled our search engine, tracker blocker, and encryption enforcer into one Firefox Extension.

Private Search Engine

Search privately with our app or extension, add private web search to your favorite browser, or search directly at duckduckgo.com.

Privacy Browser App

Our private browser for mobile comes equipped with our search engine, tracker blocker, encryption enforcer, and more. Available on iOS & Android.

We don't store your personal information. Ever.

Our privacy policy is simple: we don't collect or share any of your personal information.

Frequently Asked Questions

How does DuckDuckGo make money?

We make our money from private ads on our search engine. On other search engines, ads are based on profiles compiled about you using your personal information like search, browsing, and purchase history. Since we don't collect that information, search ads on DuckDuckGo are based on the search results page you are viewing, not on you as a person. For example, if you search for cars, we'll show you ads about cars. Learn more.

How does the DuckDuckGo app & extension work?

What does Google know about me?

What's the difference between using DuckDuckGo and Incognito mode?

How do DuckDuckGo search results compare to Google's?

How popular is DuckDuckGo?

Why use DuckDuckGo instead of Google?

Figure 3.10. DuckDuckGo search engine.

Chapter 4

Spyder: Targeted Synthesis for Programming with Data Invariants

4.1 Introduction

Programmers routinely face the task of enforcing *data invariants*. Prominent examples of data invariants include well-formedness of data structures, model-view relations in interactive GUI applications, and consistency between application data and the database. Failure to properly enforce invariants is a common source of serious bugs and security vulnerabilities [16]. Traditionally, programmers do not state invariants explicitly. Instead, they tacitly maintain invariants by sprinkling invariant-restoring snippets across their code. This ad-hoc practice is error-prone because the programmer must maintain a mental model of which invariants are broken and how to restore them. In addition, these snippets are brittle under software evolution: when data structures and their invariants change, the programmer must go over the entire code base to modify, remove, or add invariant-restoring snippets.

An attractive alternative to this traditional model is to let programmers state the desired invariants explicitly, and have the programming language take responsibility for both *checking* the invariant satisfaction, as well as *enforcing* the invariants by updating the necessary data structures. Static checking of invariants is the subject of much prior work in program verification [18, 74, 17, 76, 91, 88, 118, 104, 23]; these techniques, however, can only identify the code locations where an invariant might be violated, but they do not help the programmer restore the invariant.

Item	Weekly	Daily	Totals
Groceries	-210	-30	-30
Rent	-350	-50	-80
Fellowship	385	55	-25
Salary	700	100	75

(a)

```
var COLA = 1.05;
for (i = 0; i != rows.length; ++i) {
  var r = rows[i];
  if (r > 0) {
    r.day = r.day * COLA;
  }
  // Targeted synthesis:
  // assigns to
  // r.week, r.total
}
// Naive synthesis: after loop
// regenerate rows and preserve days
```

(b)

Figure 4.1. GUI application for building a budget from recurring expenses and incomes.

On the other hand, *declarative constraint programming* [108, 57] automatically adjusts the program state to satisfy the invariant; the downside, however, is that doing so at run time is both unpredictable and inefficient. Wouldn't it be great if instead we could compile declarative constraints into imperative code? Importantly, this would make the semantics of constraints more predictable, since any ambiguity would have to be resolved at compile time, when the compiler can ask the programmer for help. In this work, we propose using *program synthesis* techniques to compile declarative data invariants into imperative invariant-enforcing patches.

Program synthesis is an active area of research [48, 113, 120, 105, 38, 140] that tackles the problem of generating programs from declarative constraints. In particular, synthesis from logical specifications [73, 116, 67, 34, 102] takes as input a logical predicate over a program's inputs and outputs, and searches for a program that satisfies the predicate. We describe how program synthesis enables language support for data invariants through a motivating example.

4.1.1 Motivating Example: Budget Planner

Consider a budget builder application for recurring expenses and incomes, shown in Figure 4.1. The amount for each item in the budget plan is stored in two different formats, `Weekly` and `Daily`, so that the end-user can provide input in the most relevant period. For example, a budget for meals can be given in `Daily` units, rent can be given in `Weekly` units, etc.

To see whether the planned budget is balanced, the daily budget items are added up: a running total stored in `Totals`; the final entry of `Totals` contains the expected overall surplus or deficit per day. Revenues are distinguished from Expenses by rendering Revenues black and Expenses red.

Each of these application properties is a data invariant that the programmer has to maintain: (1) `weekly` and `daily` are unit-conversions of each other, (2) `totals` is a running sum of the `daily` values, and (3) if an entry is negative, its font color is red.

Consider a function that adjusts the income in an existing budget according to a cost-of-living index. This function, shown in Figure 4.1b, multiplies each positive daily item by the Cost-of-Living-Adjustment (COLA) constant. The loop in Figure 4.1b breaks invariants (1) and (2): the `weekly` and `total` values are stale. Our goal is to synthesize an *invariant patch*, *i.e.* a code snippet that, when inserted into the function body, will provably restore the broken data invariants.

At a first glance, it seems natural to insert the patch at the end of the function, using the programmer-provided data invariants as the specification for synthesis. Unfortunately generating such a function-level patch is nontrivial even for this simple example. Since each row of the table is modified, the patch must involve a loop over the rows of the table. Synthesizing loops is challenging, because the synthesis algorithm must generate an inductive loop invariant. Note, that the original data invariant is not suitable because it does not hold on entry to the new loop – the programmer's loop broke the data invariant in the first place. Moreover, even if the synthesis algorithm is clever enough to generate a loop, it must be careful to preserve the programmer's original logic. The simplest solution is to update the `daily` field of each row using the `weekly` values. Such a patch would be disastrous – the data invariant is erroneously "maintained" by undoing the programmer's changes!

More generally, this simple example highlights the two main research problems for synthesis-based language support of data invariants: (a) *complex patches:* even for simple data invariants, the synthesis algorithm must calculate both inductive invariants and complex control flow, and (b) *the frame problem:* without frame conditions, the synthesis algorithm can enforce the invariant by simply reverting the programmer's changes.

4.1.2 Targeted Synthesis and the Spyder Language

The technical contribution of this paper is a solution to the above two research problems. Our solution consists of co-designing a programming language with a novel *targeted synthesis* algorithm, which generates patches *locally* – as close as possible to the invariant violation – as opposed to at the function boundaries.

Targeted synthesis addresses the problem of *complex patches* by generating multiple patches that are as local as possible. For example, in Figure 4.1b, a local patch updates `r.week` and `r.total` inside the loop. Local patches are typically much smaller; moreover, pushing a patch inside a loop often results in preserving the original data invariant between loop iterations, creating an inductive loop invariant. In our example, not only is the desired patch a short, straight-line code snippet, but also it maintains data invariant (1) as an inductive loop invariant.

Targeted synthesis also addresses *the frame problem*: enforcing invariants at basic block boundaries enables a simple syntactic check that disallows patching variables modified by the programmer in that block and thereby ensures that all programmer's changes are preserved.

This paper presents SPYDER, a core language with iterators and data invariants, which is designed to be amenable to targeted synthesis. In particular, SPYDER offers iterator-based loops and iterator-based data invariants, which allows the synthesis algorithm to exploit their structural similarity and push synthesis specifications inside loops, in order to generate local patches.

4.1.3 Main Contributions

The main contributions of this chapter are:

- Programming with data invariants: a new programming model, where the developer explicitly states relational data invariants, and a synthesis engine automatically generates code patches to maintain these invariants.

- TARGETED SYNTHESIS: a sound and efficient algorithm for synthesizing patches for the restricted but useful class of data invariants we call iterator-based invariants.
- SPYDER, a prototype implementation of TARGETED SYNTHESIS; our empirical evaluation shows that SPYDER programs are concise and compositional, and that TARGETED SYNTHESIS generates patches more efficiently than traditional program synthesis techniques.

The remainder of this chapter is structured as follows. We use the domain of web GUI applications to give a high-level overview of SPYDER in section 4.2. In section 4.3 we formalize the semantics of the SPYDER language, and section 4.4 develops our targeted synthesis algorithm for extending SPYDER programs with invariant-preserving patches. As part of our formalisms, we contribute a soundness guarantee that the targeted synthesis algorithm preserves the original invariants; this is briefly summarized in section 4.4. section 4.5 evaluates our SPYDER compiler on a series of benchmark and case studies. Finally, we conclude by giving detailed correctness proofs in section 4.6.

4.2 Overview

We begin with an overview of TARGETED SYNTHESIS on the budgeting application shown in Figure 4.1, in which the programmer uses *data invariants* to author an interactive GUI application. The rendering and logic of the application are relatively easy to express using traditional imperative programming, but this approach does not offer language support for statically enforcing application *data invariants*. We will demonstrate how SPYDER supports data invariants by iteratively building the interactive logic for this example.

4.2.1 Data Invariants

The programmer starts with the logic for the Weekly and Daily columns, shown in Figure 4.2. To do this, the programmer declares a collection of `int`s termed `weeks`, shown on line 1 of Figure 4.2a, as well as a collection of `int`s termed `days` (line 2). These two declarations introduce new mutable global variables `days` and `weeks`.

```
1  data weeks: int[];
2  data days: int[];
3
4  foreach w in weeks, d in days:
5    7 * d.val = w.val
6
7  procedure adjustForCOLA(cola: int):
8    for d in days:
9      if (d > 0):
10       d <- d * cola;
```

(a) Source code in the SPYDER language for the days and weeks columns of the budgeting application.

```
// procedure adjustForCOLA(cola: int):
for d in days, w in weeks:
  if (d > 0):
    d <- d * cola;
    w <- 7 * d;
```

(b) Generated SPYDER code for adjustForCOLA in the budgeting application.

Figure 4.2. Programming with an invariant between Days and Weeks in the budgeting application.

One invariant of the system is the *unit-conversion* invariant (invariant (1) in subsection 4.1.1): each of the elements of `weeks` is 7 times greater than the corresponding element of `days`. This invariant should always hold and in particular, needs to be enforced whenever either `weeks` or `days` is mutated. To specify the unit-conversion invariant, the programmer uses a `foreach` construct on line 4, binding the elements of `weeks` to the local iterator `w` and the elements of `days` to `d`. Using these local bindings, they express the unit-conversion invariant using the formula on line 5: `7 * d.val = w.val`.

Because this unit-conversion invariant is defined over elements of collections, traditional techniques would model collections as arrays and require a *quantified relation* over the indicies of the arrays. Such relations are notoriously tricky to build by hand (and indeed, to verify), but in SPYDER, the programmer can use the `foreach` abstraction. This abstraction builds an element-wise product relation by introducing fresh *iterator* bindings over the abstracted collections.

4.2.2 Maintaining Data Invariants with Spyder

Next, the programmer writes imparative code implementing the desired functionality, without correcting for the violated unit-conversion invariant, as SPYDER will patch to maintain it. In the application, recall that the budget-builder needs to adjust all of the revenues (and only the revenues) in the budget by the Cost-of-Living-Adjustment (COLA). To implement this modification, the programmer writes a *procedure* called `adjustForCOLA` on line 7. This function iterates over the elements of `days` using the `for` loop on line 8, which binds each element of `days` to a local iterator variable `d`.

Since the COLA should only be applied to revenues, the programmer checks the value of the element `d` using a conditional on line 9, and then scales the daily revenue by an iterator update on line 10. The iterator semantics of SPYDER are standard for object-oriented iterators; in particular, notice that the value of the iterator (e.g. `d.val`) is implicitly given by the iterator variable itself (e.g. `d` in the expression `d > 0`).

In this code snippet, the programmer has directly assigned an updated value to `d`, and by extension the values of `days`. On its own, this update breaks the unit-conversion invariant – in particular, the Weekly value of this row of the application depends on the concrete value of `d`. Using traditional techniques, the programmer would have to manually maintain the invariant by setting the corresponding value of `weeks`, i.e. by adding an extra snippet for correctly updating `weeks`.

Fortunately for the programmer, invariants are statically maintained in SPYDER and the compiler synthesizes and inserts a invariant-restoring snippet automatically, as shown in Figure 4.2b. In this case, the compiler *extends* the original loop over `days` with an extra binding over `weeks`; in SPYDER, this has the semantics of a simultaneous iteration (analogous to a functional zip) so that `d` and `w` refer to elements of `days` and `weeks` at the same index.

More generally, in contrast to traditional programming, SPYDER enables the programmer to write modifications that are *agnostic* to the existing invariants. In this case, the programmer simply writes a direct update to the elements of `days` and SPYDER ensures that the overall system's state is correct.

4.2.3 Program Composition Through Data Invariants

In this subsection we demonstrate code evolution with SPYDER. At some later date, the programmer adds a feature to the budget application: a running totals column to help track the state of the budget. To do this, they add a collection for the total values, and the data invriant to populate it, seen in Figure 4.3a, lines 8-9. In order to define the running-sum property, SPYDER provides an iterator method called `prev`, which allows access to the previous value of the iterator. This is useful for defining accumulator properties or enforcing sortedness. SPYDER will generate the implementation of populating the totals column in its entirety.

However, since time has passed since the last change made to the system, the programmer has forgotten about `adjustForCOLA`, which breaks our new totals invariant. In a traditional imparative programming paradigm, it would be the programmer's responsibility to track down every function that breaks the invariant and fix it. However, with SPYDER, the compiler checks the new invariant against all existing functions and generates a new patch to `adjustForCOLA` to ensure it is maintained.

The different invariants are compositional from the user's perspective—in practice, each function is checked against all invariants in the code. It is the responsibility of TARGETED SYNTHESIS to find in a failed set if invariants the actual invariants that have failed, and to reduce those to a local specification that can be used to synthesize a patch. This is shown in subsection 4.4.2.

In evolving the codebase, the programmer later adds another feature, coloring negative values in red. This is done using two sets of invariants: one for totals (lines 12-14), and one for days (lines 16-19). Notice that `adjustForCOLA` does not invalidate the days invariant. SPYDER checks this in compile time, resulting in no changes being made to `adjustForCOLA`—as opposed to dynamic techniques which would generate code that tests this in runtime.

The code generated by SPYDER is seen in Figure 4.3b. The fixes introduced by SPYDER are nontrivial in several ways: (1) the fixes are *extensive*, accounting for the majority of the

```
data weeks: int[];
data days: int[];

foreach w in weeks, d in days:
  7 * d.val = w.val

data totals: int[];
foreach d in days, t in totals:
  t.val = t.prev(0) + d.val

data totalFontColors: int[];
foreach t in totals, c in totalFontColors:
  (t.val >= 0 <=> c.val = black)) &&
  (t.val < 0 <=> c.val = red)

data rowFontColors : int[];
foreach d in days, c in rowFontColors:
  (d.val >= 0 <=> c.val = black)) &&
  (d.val < 0 <=> c.val = red)
```

(a) SPYDER source code for an accumulated Totals invariant.

```
// procedure adjustForCOLA(cola: int):
for d in days, w in weeks, t in totals,
    cr in rowFontColors, ct in totalFontColors:
  t <- t.prev(0) + d;
  if (t < 0):
    ct <- red;
  else:
    ct <- black;
  if (d > 0):
    d <- d * cola;
    w <- 7 * d;
    t <- t.prev(0) + d;
    if (d < 0):
      cr <- red;
    else:
      cr <- black;
    if (t < 0):
      ct <- red;
    else:
      ct <- black;
```

(b) Generated SPYDER code for an update to Days.

Figure 4.3. Programming with an accumulator invariant between `days` and `totals`, as well as a font color invariant. Colors indicate the relationship between the invariant (a) and generated code (b).

code in Figure 4.3b, (2) the fixes are *nonlocal*, meaning that each fix is spread out over (and interleaved with) the original code, (3) the fixes have to *add new variables* just to maintain the invariants, and (4) each invariant requires multiple fixes.

4.2.4 Generalization of Technique

From the programmer's perspective, the process of invariant patching is invisible – SPYDER accepts the original, invariant-oblivious code. More generally, the cognitive load of imperative programming with invariants in SPYDER is significantly less than traditional techniques. Using TARGETED SYNTHESIS, when writing imperative code, the programmer needs to only reason about *local* code properties (e.g. the value of `d`) and does not need to reason about *global* code invariants (e.g. the relation between `days` and `rowFontColors`).

The structure of the remainder of this chapter is as follows. We first describe the SPYDER source language in section 4.3, and show how to translate from SPYDER terms to a well-studied imperative language. We also give a hoare-style axiomatic semantics to SPYDER programs, and provide a formal guarantee that SPYDER verification triples are equivalent to standard triples. In section 4.4, we present synthesis rules for patching and extending SPYDER programs, and provide a formal guarantee that our synthesis rules are sound with respect to our SPYDER verification triples. Finally, we present several case studies and a benchmarking evaluation in section 4.5.

4.3 The Spyder Language

We present the syntax and semantics of the SPYDER source language. In this section, we do not describe how to maintain data invariants. Instead, we just provide the formal framework for both expressing collection-manipulation programs, as well as axiomatically verifying properties over programs. We will build on the results of this section in section 4.4 to show how to maintain data invariants through TARGETED SYNTHESIS.

First, we introduce the core syntax of SPYDER in subsection 4.3.1. Then, we give a semantics to the syntax by translating SPYDER terms to a well-studied standard imperative language in subsection 4.3.2. Next, we demonstrate how to mechanically verify when invariants are maintained or violated by defining a Hoare-style [53, 40] axiomatic logic for SPYDER in subsection 4.3.4. Finally, we give a proof of soundness for this verification logic by reduction to the standard axiomatic semantics for imperative array programs (i.e. Hoare logic) in Theorem 4.3.1.

4.3.1 Surface Syntax for Spyder

At its core, SPYDER is an imperative collection-manipulation language. The focus in SPYDER is to support data invariants for mutable, finite collections. To this end, we formalize and define a core calculus for iterating over and mutating collections, which we present in Figure 4.4a.

Values and Types

SPYDER has three datatypes: *integers*, *collections*, and *iterators*. *Integers* are standard and we denote a variable declaration of type int as `data x: int`.

Collections

Collections hold elements and are analogous to ordered containers, e.g. lists or arrays. For variable declarations, we denote a collection of `T` by `data col: T[]`. Collections are homogeneous and for clarity of presentation, our core syntax and formalisms assume that all collections are 1-dimensional (i.e. collections of integers). In our implementation, however, collections can nest arbitrarily (and extending the formalisms to arbitrary nesting is straightforward). For example, the list `[1,2,3]` is a collection of integers and the list `[[1,2],[3,4]]` is a collection of integer collections. In contrast, the list `[1,[2,3]]` has mixed element types and is not valid. Collections expose a single method, size, which returns the number of elements in the collection.

For simplicity, we assume all collections have a statically known size which does not vary at runtime. We also assume that collection sizes are homogeneous, for example, the list `[[1,2],[3,4,5]]` would not be a valid SPYDER collection.

A key difference between SPYDER collections and traditional arrays is that collections do not support subscription (i.e. `col[idx]` is not a valid SPYDER term). Instead, to access the elements of a collection, SPYDER exposes the for (x, y) statement, which iterates over the values of the collection y. In addition to iteration, the for statement creates a new variable binding for an *iterator* variable.

Iterators

Iterators allow access to the underlying elements of a collection. Iterator variables are not explicitly declared using `data` but are instead created in for loops. SPYDER supports several standard iterator methods: val, which returns the value of the iterated collection; idx, which returns the current iteration index; prev, which returns the previous value; and the $x \leftarrow E$ operator (termed "put"), which destructively *updates* the value of the iterator `x` with the expression E. For example, after the evaluation of the term `for x in xs: x <- x + 1;` each element of `xs` is incremented by exactly one.

Statements

For control-flow, SPYDER has mostly standard imperative statements. A key exception is the for term, which as discussed above, iterates over a collection. In addition, the for loop iterates over multiple collections simultaneously, similar to a zip in function programming. For example the term `for x in xs, y in ys: y <- x.val;` replaces each element in `ys` with the corresponding element in `xs`. Furthermore, iteration is only well-defined when the iterated collections have the same size.

Specifications

To express specifications for SPYDER terms, SPYDER exposes a rich specification language, the Spec terms. To ease the burden of synthesis and verification, we syntactically phrase specifications in a conjunctive normal form. At the top level are conjunctions of specifications using the $\wedge$ operator. Each conjunct can be either a bare expression, or a quantification term.

SPYDER supports two quantifiers: (1) An existential quantifier through the exists keyword. This quantifier is not present in the surface syntax of SPYDER and is only used in the axiomatic semantics, which we present in subsection 4.3.4. (2) A universal quantifier through the foreach keyword, which quantifies over the elements of a collection. For example, the specification `foreach x in xs, y in ys: x.val > y.val` states that each element of `xs` is greater than the corresponding element of `ys`. Similar to the for statement, the foreach term is only well-defined when the bound collections have the same size. We discuss the details of specifications more in subsection 4.3.4.

$$\begin{array}{lcl}
 & & \mathsf{v}, \mathsf{u} \in \mathsf{Vars}, \quad \mathsf{i} \in Z \\
\mathsf{Spec} & ::= & \mathsf{foreach}\ \overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \mathsf{Spec} \\
 & \mid & \mathsf{exists}\ \mathsf{v}\ .\ \mathsf{Spec} \\
 & \mid & \mathsf{Spec} \wedge \mathsf{Spec} \\
 & \mid & \mathsf{Expr} \\
\mathsf{Block} & ::= & \texttt{skip} \mid \mathsf{Stmt}\ ;\ \mathsf{Block} \\
\mathsf{Stmt} & ::= & \mathsf{v} := \mathsf{Expr} \\
 & \mid & \mathsf{v} \leftarrow \mathsf{Expr} \\
 & \mid & \texttt{if}\ \mathsf{Expr}\ \texttt{then}\ \mathsf{Block}\ \texttt{else}\ \mathsf{Block} \\
 & \mid & \mathsf{for}\ \overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \mathsf{Block} \\
\mathsf{Expr} & ::= & \mathsf{v} \mid \mathsf{i} \mid \mathsf{true} \mid \mathsf{false} \\
 & \mid & \mathsf{Expr}\ \mathsf{bop}\ \mathsf{Expr} \\
 & \mid & \mathsf{uop}\ \mathsf{Expr} \\
 & \mid & \mathsf{v.val} \\
 & \mid & \mathsf{v.prev}(\mathsf{Expr}) \\
 & \mid & \mathsf{v.idx} \\
 & \mid & \mathsf{v.size} \\
\mathsf{bop} & ::= & + \mid \times \mid \% \mid \Longrightarrow \mid \Longleftrightarrow \mid \ldots \\
\mathsf{uop} & ::= & \neg \mid !
\end{array}$$

(a) Syntax for the SPYDER language.

$$\begin{array}{lcl}
 & & \mathbf{v}, \mathbf{u} \in \mathbf{Vars}, \quad \mathbf{i} \in \mathbb{Z} \\
\mathbf{Stmt} & ::= & \mathbf{v} := \mathbf{Expr} \\
 & \mid & \mathbf{v}\ [\ \mathbf{Expr}\ ] := \mathbf{Expr} \\
 & \mid & \texttt{if}\ \mathbf{Expr}\ \texttt{then}\ \mathbf{Stmt}\ \texttt{else}\ \mathbf{Stmt} \\
 & \mid & \mathsf{while}\ \ \mathbf{Expr}\ \mathbf{Stmt} \\
 & \mid & \mathbf{Stmt}\ ;\ \mathbf{Stmt} \\
 & \mid & \texttt{skip} \\
\mathbf{Expr} & ::= & \mathbf{v} \mid \mathbf{i} \mid \mathsf{true} \mid \mathsf{false} \\
 & \mid & \mathbf{Expr}\ \mathbf{bop}\ \mathbf{Expr} \\
 & \mid & \texttt{if}\ \mathbf{Expr}\ \texttt{then}\ \mathbf{Expr}\ \texttt{else}\ \mathbf{Expr} \\
 & \mid & \mathbf{uop}\ \mathbf{Expr} \\
 & \mid & \mathbf{Expr}\ [\ \mathbf{Expr}\ ] \\
 & \mid & \mathsf{size}(\mathbf{v}) \\
 & \mid & \forall\ \mathbf{v}\ .\ \mathbf{Expr} \\
 & \mid & \exists\ \mathbf{v}\ .\ \mathbf{Expr} \\
\mathbf{bop} & ::= & + \mid \times \mid \% \mid \Longrightarrow \mid \Longleftrightarrow \mid \ldots \\
\mathbf{uop} & ::= & \neg \mid !
\end{array}$$

(b) Syntax for the IMP-ARRAY language.

Figure 4.4. Syntax for SPYDER and IMP-ARRAY.

4.3.2 Imperative Target Language

We formalize the semantics of SPYDER by translating to an idealized imperative verification language, which we call IMP-ARRAY. The syntax of this verification target language is shown in Figure 4.13. This language is very similar to Boogie [71] and indeed, in our implementation, we compile and synthesize to Boogie.

Although SPYDER and IMP-ARRAY have similar syntax, there are several major differences. Broadly speaking, IMP-ARRAY does not have language support for either collections or iterators. IMP-ARRAY instead offers mutable low-level arrays, which map (integer) indices to values. At the statement level IMP-ARRAY supports mutable updates to both variables and arrays, as well as general while loops. For expressions, IMP-ARRAY enables rich quantification through the $\forall$ quantifier, but in contrast to SPYDER, does not support iterator methods.

To support collections and iterators, the translation from SPYDER to IMP-ARRAY must implement collection and iterator logic in terms of arrays and indices. We show an example of this in Figure 4.5, in which a SPYDER program for calculating a product is translated into IMP-ARRAY. In this case, the integer collections `values` and `product` in SPYDER map 1-to-1 to arrays in IMP-ARRAY, and the `for` loop in SPYDER is desugared into a `while` loop with an

```
data values: int[];
data product: int[];

foreach v in values, fact in product:
  fact.val = fact.prev(1) * v.val

procedure multValues():
  for v in values, fact in product:
    v <- v.val * 1.05;
    fact <- fact.prev(1) * v.val;
```

(a) SPYDER source code for a product invariant.

```
var values: [int]int;
var v: int;
var product: [int]int;
var fact: int;

procedure multValues(){
  v := 0; fact := 0;
  while
    (v < size(values) && fact < size(
    product))
// invariant:
// forall i. 0 <= i < v ==>
//    product[i] == values[i] *
//      (if i == 0 then 1 else product[i
    -1])
  {
    val[v] := val[v] * 1.05;
    if (index == 0) {
      product[fact] := val[v];
    } else {
      product[fact] := product[fact-1] *
     val[v];
    }
    fact := fact + 1; v := v + 1;
  }
}
```

(b) Translated IMP-ARRAY code for a product invariant.

Figure 4.5. Source code and translation maintaining a product invariant. In contrast to the examples in section 4.2, the source code maintains the invariant, and the translation step must soundly produce ImpArray code which also maintains it.

explicit index in IMP-ARRAY. At a high-level, collections in SPYDER correspond 1-to-1 with arrays in IMP-ARRAY, and iterator variables in SPYDER correspond to indices in IMP-ARRAY. This example is similar to a subproblem of the TARGETED SYNTHESIS algorithm (discussed in detail in section 4.4), which reasons about candidate programs like multValues.

4.3.3 Overview of Translation Semantics

We formalize translation as a syntax-directed recursive function over SPYDER terms given in Figure 4.6. Since `for` loops bind iterator variables, the translation must be stateful. We choose to explicitly pass the state using finite mathematical maps, which we term *translation contexts* and we generally denote as Γ. We denote the translation of a term t using the context Γ as the IMP-ARRAY term $\texttt{trans}(t, \Gamma)$; we refer to this as “the translation of t in the context of Γ”.

SPYDER Term		IMP-ARRAY Term
$\texttt{trans}(\mathsf{v}, \Gamma)$	$=$	$\mathbf{v}$
$\texttt{trans}(\mathsf{i}, \Gamma)$	$=$	$\mathbf{i}$
$\texttt{trans}(\mathsf{true}, \Gamma)$	$=$	true
$\texttt{trans}(\mathsf{false}, \Gamma)$	$=$	false
$\texttt{trans}(E_l\ \mathsf{bop}\ E_r, \Gamma)$	$=$	$\texttt{trans}(E_l, \Gamma)\ \mathbf{bop}\ \texttt{trans}(E_r, \Gamma)$
$\texttt{trans}(\mathsf{uop}\ E, \Gamma)$	$=$	$\mathbf{uop}\ \texttt{trans}(E, \Gamma)$
$\texttt{trans}(\mathsf{v}.\mathsf{val}, \Gamma)$	$=$	$\Gamma(\mathsf{v})[\mathbf{v}]$
$\texttt{trans}(\mathsf{v}.\mathsf{prev}(E), \Gamma)$	$=$	$\texttt{if}\ \mathbf{v} > 0\ \texttt{then}\ \Gamma(\mathsf{v})[\mathbf{v}-1]\ \texttt{else}\ \texttt{trans}(E, \Gamma)$
$\texttt{trans}(\mathsf{v}.\mathsf{idx}, \Gamma)$	$=$	$\mathbf{v}$
$\texttt{trans}(\mathsf{v}.\mathsf{size}, \Gamma)$	$=$	$\mathsf{size}(\mathsf{v})$

(a) Translation rules for Spyder Expressions to ImpArray Epressions.

SPYDER Term		IMP-ARRAY Term
$\texttt{trans}(\mathsf{foreach}\ (\mathsf{v},\mathsf{u})\ I, \Gamma)$	$=$	$\forall\, \mathbf{v}\ .\ (0 \le \mathbf{v} \wedge \mathbf{v} < \mathsf{size}(\mathbf{u})) \implies \texttt{trans}(I, \Gamma \oplus \mathsf{v} \mapsto \mathbf{u})$
$\texttt{trans}(\mathsf{exists}\ \mathsf{v}\ I, \Gamma)$	$=$	$\exists\, \mathbf{v}\ .\ \texttt{trans}(I, \Gamma)$
$\texttt{trans}(I_l \wedge I_r, \Gamma)$	$=$	$\texttt{trans}(I_l, \Gamma) \wedge \texttt{trans}(I_r, \Gamma)$
bottomrule		

(b) Translation rules for Spyder Specifications to ImpArray Expressions.

SPYDER Term		IMP-ARRAY Term
$\texttt{trans}(\mathsf{v}{:=}E, \Gamma)$	$=$	$\mathbf{v} := \texttt{trans}(E, \Gamma)$
$\texttt{trans}(\texttt{if}\ E\ \texttt{then}\ B_t\ \texttt{else}\ B_f, \Gamma)$	$=$	$\texttt{if}\ \texttt{trans}(E, \Gamma)\ \texttt{then}\ \texttt{trans}(B_t, \Gamma)\ \texttt{else}\ \texttt{trans}(B_f, \Gamma)$
$\texttt{trans}(\mathsf{v} \leftarrow E, \Gamma)$	$=$	$\Gamma(\mathsf{v})[\mathbf{v}] := \texttt{trans}(E, \Gamma)$
$\texttt{trans}(\mathsf{for}\ (\mathsf{x},\mathsf{y}) B_i, \Gamma)$	$=$	$\mathbf{x} := 0\,;$ $\mathsf{while}\ (\mathbf{x} < \mathsf{size}(\mathbf{y}))\ \texttt{trans}(B_i, \Gamma \oplus \mathsf{x} \mapsto \mathbf{y})\,; \mathbf{x} := \mathbf{x} + 1$

(c) Translation rules for Spyder Statements to ImpArray Statements.

Figure 4.6. Translation rules for Spyder to ImpArray

$$\text{Var-Global}\ \frac{\begin{array}{c}\mathsf{v} \in \texttt{globals}\\ \mathsf{v} \notin \Gamma\end{array}}{\texttt{wf}(\mathsf{v},\ \Gamma)} \qquad \text{Var-Bound}\ \frac{\begin{array}{c}\mathsf{v} \notin \texttt{globals}\\ \mathsf{v} \in \Gamma\end{array}}{\texttt{wf}(\mathsf{v},\ \Gamma)}$$

$$\text{Prim-Int}\ \frac{}{\texttt{wf}(\mathsf{i},\ \Gamma)} \qquad \text{Prim-BT}\ \frac{}{\texttt{wf}(\mathsf{true},\ \Gamma)} \qquad \text{Prim-BF}\ \frac{}{\texttt{wf}(\mathsf{false},\ \Gamma)}$$

$$\text{Bop}\ \frac{\texttt{wf}(E_l,\ \Gamma) \quad \texttt{wf}(E_r,\ \Gamma)}{\texttt{wf}(E_l\ \mathsf{bop}\ E_r,\ \Gamma)} \qquad \text{Uop}\ \frac{\texttt{wf}(E,\ \Gamma)}{\texttt{wf}(\mathsf{uop}\ E,\ \Gamma)} \qquad \text{Elem}\ \frac{\mathsf{v} \in \Gamma}{\texttt{wf}(\mathsf{v}.\mathsf{val},\ \Gamma)}$$

$$\text{Prev}\ \frac{\mathsf{v} \in \Gamma \quad \texttt{wf}(E,\ \Gamma)}{\texttt{wf}(\mathsf{v}.\mathsf{prev}(E),\ \Gamma)} \qquad \text{Idx}\ \frac{\mathsf{v} \in \Gamma}{\texttt{wf}(\mathsf{v}.\mathsf{idx},\ \Gamma)} \qquad \text{Size}\ \frac{\mathsf{v} \in \mathrm{range}(\Gamma)}{\texttt{wf}(\mathsf{v}.\mathsf{size},\ \Gamma)}$$

(a) Well-formedness rules for Spyder Expressions.

$$\text{Foreach}\ \frac{\begin{array}{c}\mathsf{u} \notin \Gamma \quad \mathsf{u} \notin \mathrm{range}(\Gamma) \quad \mathsf{v} \notin \Gamma \quad \mathsf{u} \in \texttt{globals}\\ \texttt{wf}(I,\ \Gamma \oplus \mathsf{v} \mapsto \mathsf{u})\end{array}}{\texttt{wf}(\mathsf{foreach}\ (\mathsf{v},\mathsf{u})I,\ \Gamma)}$$

$$\text{Exists}\ \frac{(\mathit{fresh}\,\mathsf{x}) \quad \texttt{wf}(I,\ \Gamma)}{\texttt{wf}(\mathsf{exists}\ \mathsf{x}\ .\ I,\ \Gamma)} \qquad \text{Conjunct}\ \frac{\begin{array}{c}\texttt{wf}(I_l,\ \Gamma)\\ \texttt{wf}(I_r,\ \Gamma)\end{array}}{\texttt{wf}(I_l \wedge I_r,\ \Gamma)}$$

(b) Well-formedness rules for Spyder Invariants.

$$\text{Blk-Skip}\ \frac{}{\texttt{wf}(\texttt{skip},\ \Gamma)} \qquad \text{Blk-Seq}\ \frac{\texttt{wf}(S,\ \Gamma) \quad \texttt{wf}(B,\ \Gamma)}{\texttt{wf}(S\ ;\ B,\ \Gamma)}$$

$$\text{Stmt-Assign}\ \frac{\mathsf{v} \notin \Gamma \quad \texttt{wf}(E,\ \Gamma)}{\texttt{wf}(\mathsf{v}{:=}E,\ \Gamma)} \qquad \text{Stmt-Put}\ \frac{\mathsf{v} \in \Gamma \quad \texttt{wf}(E,\ \Gamma)}{\texttt{wf}(\mathsf{v} \leftarrow E,\ \Gamma)}$$

$$\text{Stmt-Cond}\ \frac{\texttt{wf}(E,\ \Gamma) \quad \texttt{wf}(B_t,\ \Gamma) \quad \texttt{wf}(B_f,\ \Gamma)}{\texttt{wf}(\texttt{if}\ E\ \texttt{then}\ B_t\ \texttt{else}\ B_f,\ \Gamma)}$$

$$\text{Stmt-For}\ \frac{\begin{array}{c}\mathsf{y} \notin \mathrm{range}(\Gamma) \quad \mathsf{x} \notin \mathrm{assign}(B_i) \quad \mathsf{y} \in \texttt{globals}\\ \texttt{wf}(B_i,\ \Gamma \oplus \mathsf{x} \mapsto \mathsf{y})\end{array}}{\texttt{wf}(\mathsf{for}\ (\mathsf{x},\mathsf{y})B_i,\ \Gamma)}$$

(c) Well-formedness rules for Spyder Statements.

Figure 4.7. Well-formedness rules for Spyder. For exposition, when rules bind a variable we only formalize the well-formedness for a single binding. The extension to multiple bindings is straightforward.

Well-formedness of Translation Contexts

In general, the translation process is only well-defined if the translation context Γ is well-formed. Intuitively, there must be no name-collisions; a collection must not be iterated over multiple times; an iterator variable must not be directly written to (i.e. using assignment $:=$ instead of the iterator $\leftarrow$ operator); etc. We formalize these well-formedness constraints in Figure 4.7, which relates SPYDER terms t to translation contexts that are well-formed for translating t. We denote a well-formed term using $\texttt{wf}(t,\ \Gamma)$ and we say Γ is well-formed with respect to t.

Semantics for Spyder terms

Since IMP-ARRAY is well-studied and has a well-understood semantics (replicated in subsection 4.6.1), we define the semantics of SPYDER by translating into IMP-ARRAY. For details, see subsection 4.6.2.

A keen observer will notice that SPYDER's semantics are focused on alias-free iterator-based programs. IMP-ARRAY has actually been studied in the context of verifying more exotic language features, such as object-oriented invariants [75], concurrency [26], general arrays [72], heap-manipulation [99], etc. Because other systems have verified exotic programs using the rich, low-level semantics of IMP-ARRAY, in the future the semantics of SPYDER can be extended to handle relational invariant maintenance for more complicated languages.

4.3.4 Verification in Spyder and ImpArray

We next define and present an axiomatic semantics for SPYDER that TARGETED SYNTHESIS will use to mechanically verify when invariants are preserved or violated by a statement. In addition, we prove that SPYDER axiomatic semantics are sound with respect to the standard axiomatic semantics for IMP-ARRAY (i.e. Hoare triples).

Hoare Triples for ImpArray

We start by briefly reviewing axiomatic semantics in IMP-ARRAY which are well-studied [53]. The standard approach, called Hoare triples, are deduction rules for relating three terms: a precondition P, a statement S, and a postcondition Q, denoted by $\{P\}\,S\,\{Q\}$. Intuitively, the rules derive a triple if and only if given the precondition P, the postcondition Q holds after executing the statement S. We replicate these rules in Figure 4.16.

Notice that in standard axiomatic semantics, the loop rule requires an inductive invariant I to be maintained on *every* iteration. Furthermore, the axiomatic rules do not contain a notion of termination. As a result, the triple $\{P\}\,S\,\{Q\}$ should only be interpreted as valid if the statement S terminates. In our case, termination is orthogonal. Our well-formedness constraints ensure that all loops over finite collections terminate, and so in practice, this is not an issue for our use of the axiomatic semantics of IMP-ARRAY.

Hoare Triples for Spyder

We next provide a similar axiomatic semantics for SPYDER terms. In this case, we derive a triple $\langle P\rangle\ S\ \langle Q\rangle$, which has the same intuitive interpretation: given P, Q holds after executing S. As part of our contribution, we prove that the logic of Figure 4.8 is *relatively sound*: given a well-formed translation context, the axiomatic rules are sound with respect to Hoare logic. Intuitively, if we prove a triple in the SPYDER semantics, then the corresponding translated triple holds in Hoare's axiomatic semantics.

More formally, let P and Q be SPYDER Expressions, let S be a SPYDER Statement, and let Γ be a translation context. If Γ is well-formed with respect to P, Q, and S, and we derive the triple $\langle P\rangle S \langle Q\rangle$, then there exists a Hoare Triple for the corresponding translated terms in IMP-ARRAY:

Theorem 4.3.1 (Relative Soundness)**.**

$$\forall P, S, Q, \Gamma \,.\, \texttt{wf}(P \wedge Q,\, \Gamma) \wedge \texttt{wf}(S,\, \Gamma) \implies \langle P\rangle\, S\, \langle Q\rangle \implies \{\texttt{trans}(P,\, \Gamma)\}\,\texttt{trans}(S,\, \Gamma)\,\{\texttt{trans}(Q,\, \Gamma)\}$$

We prove this property by induction over the derivation of the SPYDER Triple $\langle P\rangle S \langle Q\rangle$, given in subsection 4.6.3. The key parts of the proof are the soundness of the `Put` and `For` rules which we discuss in detail below.

Strong Iterator Updates

`Put` is interesting because under the hood, the update $\mathtt{x} \leftarrow E$ translates to an array write (namely $\Gamma(\mathtt{x})[\mathbf{x}]{:=}E$). This is potentially problematic because standard array semantics assume indices can alias and so all information about the collection $\Gamma(\mathtt{x})$ is lost after the update. However, SPYDER has no variable aliasing. Moreover, the well-formedness rules ensure that values of

$$\text{Consequence}\ \frac{\begin{array}{c} P \implies P' \qquad Q' \implies Q \\ \langle P' \rangle\ S\ \langle Q' \rangle \end{array}}{\langle P \rangle\ S\ \langle Q \rangle} \qquad \text{Conditional}\ \frac{\begin{array}{c} \langle P \land E \rangle\ B_t\ \langle Q \rangle \\ \langle P \land \neg E \rangle\ B_f\ \langle Q \rangle \end{array}}{\langle P \rangle\ \mathtt{if}\ E\ \mathtt{then}\ B_t\ \mathtt{else}\ B_f\ \langle Q \rangle}$$

$$\text{Assign}\ \frac{(\mathit{fresh}\ v')}{\langle P \rangle\ v{:=}E\ \langle \mathsf{exists}\ v'\,.\,P[v \mapsto v'] \land v = E[v \mapsto v'] \rangle} \qquad \text{Sequence}\ \frac{\begin{array}{c} \langle P \rangle\ S\ \langle Q \rangle \\ \langle Q \rangle\ B\ \langle R \rangle \end{array}}{\langle P \rangle\ S\ ;\ B\ \langle R \rangle}$$

$$\text{Put}\ \frac{(\mathit{fresh}\ v')}{\langle P \rangle\ v \leftarrow E\ \langle \mathsf{exists}\ v'\,.\,P[\mathsf{val}(v) \mapsto v'] \land \mathsf{val}(v) = E[\mathsf{val}(v) \mapsto v'] \rangle}$$

$$\text{For}\ \frac{\begin{array}{c} \mathrm{mod}(B_i) \cap \mathrm{free}(I) = \emptyset \\ \langle \mathrm{weaken\_prev(I)} \land 0 \leq \mathsf{idx}(\mathsf{x}) < \mathsf{size}(\mathsf{y}) \rangle\ B_i\ \langle I \rangle \end{array}}{\langle \mathsf{foreach}\ (\mathsf{x},\mathsf{y})\ I \rangle\ \mathsf{for}\ (\mathsf{x},\mathsf{y}) B_i\ \langle \mathsf{foreach}\ (\mathsf{x},\mathsf{y})\ I \rangle} \qquad \text{Skip}\ \frac{}{\langle P \rangle\ \mathtt{skip}\ \langle P \rangle}$$

Figure 4.8. Hoare-style verification logic for SPYDER. For exposition, we only formalize the relation loops with a single variable binding. Since loops are only well-defined when the iterated collections have the same statically known size, the extension to multiple bindings is straightforward.

the collection $\Gamma(\mathsf{x})$ can only be referenced through exactly one iterator x and one expression x.val.[1] Consequently, in the `Put` rule we reason about the value of x.val while soundly retaining information about the collection $\Gamma(\mathsf{x})$.

Quantifier introduction and maintenance

A key requirement of the axiomatic semantics is to soundly reason about when loops maintain (or violate) universally quantified invariants (i.e. foreach terms). To that end, we provide a `For` rule, which is similar to a standard while rule in that the inductive invariant is on both sides of the statement. Unlike the Hoare while rule, however, the `For` rule for a loop `for x in xs` requires a top-level `foreach x in xs` as well.[2]

In order to show that a `foreach` invariant is maintained by a `for` loop, it suffices to reason about each iteration of the loop in isolation. Due to the well-formedness constraints, the only way to modify the elements of a collection is through the $\leftarrow$ operator. As a consequence the execution of a loop iteration cannot invalidate the results of previous iterations. Since the

[1] In particular the well-formedness relation prohibits a foreach quantifier over a collection y from entering the body of a loop over y.

[2] If a top-level term is not in this form but is equivalent under renaming and quantifier shuffling, the `Consequence` rule can be used to rewrite the term to make progress.

loop is guaranteed to execute for each element of the collection, the rule introduces a `foreach` quantifier after the loop is complete.

Furthermore, it's tempting to assume the specialized invariant as a precondition to verifying the loop body. If the invariant does not contain the prev method, this is completely valid. However, the `prev` method complicates matters because each iteration does not necessarily establish `prev` for the next iteration. To address this situation, we use the weaken_prev helper function to soundly weaken an expression with respect to `prev`. As a result, the `For` rule retains as much information as is soundly possible, and enables automated verification and synthesis by removing a layer of quantification.

4.3.5 Maintaining Data Invariants

With an axiomatic semantics for SPYDER programs, we now consider several techniques for maintaining data invariants. We use a simple midpoint program in Figure 4.9, in which two variables `l` and `r` sum to 10, to demonstrate these techniques.

```
data l: int;
data r: int;

// invariant: l + r = 10


procedure incrL():
  l = l + 1;
  r = r - 1;
procedure incrR():
  r = r + 1;
  l = l - 1;
```

(a) Imperative: program with no additional specifications.

```
data l: int;
data r: int;

// invariant: l + r = 10
l = 10 - r
r = 10 - l

procedure incrL():
  l = l + 1;

procedure incrR():
  r = r + 1;
```

(b) FRP: functional specifications for `l` and `r`.

```
data l: int;
data r: int;

// invariant: l + r = 10

l + r = 10

procedure incrL():
  l = l + 1;

procedure incrR():
  r = r + 1;
```

(c) SPYDER: a single relational specification for `l` and `r`.

Figure 4.9. Three different specification techniques used to implement a midpoint program in which `l` and `r` sum to 10.

Imperative Invariant Maintenance

The most common technique for invariant maintenance is to manually track invariants and provide a patch that maintains the invariant. This is tedious and error prone because the programmer must manually remember 1) what the invariant is, and 2) how to maintain the invariant when it breaks. For example, in the midpoint program (Figure 4.9a), the programmer must remember that `l` must be decremented after `r` is incremented, and vice-versa.

From the programmer's perspective, this is also the least compositional approach to invariant maintenance. If the invariant changes, it is up to the programmer to find all the patches and fix them. However it is also the most performant technique; the runtime system simply executes the code.

Functional Invariant Maintenance

An alternative approach to manual maintenance is the Functional-Reactive programming (FRP) paradigm, in which the programmer provides a functional specification for solving the invariant, and the language runtime detects when the functional specification should be invoked. In this example Figure 4.9b the programmer gives two functional specifications for `l` and `r`, each in terms of the other. In return, the language runtime uses these specifications to perform invariant maintenance, saving the programmer the need to reason about maintenance within the implementation of `incrL()` or `incrR()`. The downside of this approach is that the runtime system must dynamically track data-dependencies, incurring a runtime overhead compared to the imperative approach.

Spyder Invariant Maintenance

Finally, TARGETED SYNTHESIS enables automatic *relational* invariant maintenance. In contrast to a functional specification, a *relational* specification does not easily admit a clear resolution for the specification. From the programmer's perspective, relational specifications are much more clear and concise. Consider in this example the specification in Figure 4.9c; it clearly and unambiguously captures the data invariant that `l` and `r` sum to 10.

The power and expressiveness of relational specifications comes at a cost. One way to handle these rich relational invariants is to dynamically solve the relational specification, similar to FRP. This incurs a significant runtime overhead and moreover, when the specification is erroneous, dealing with the error falls to to the end-user of the code and not the programmer.

Instead, we take the approach of solving these invariants at compile time using program synthesis. In the next section, we detail exactly how TARGETED SYNTHESIS enables the programmer to use relational specifications automatically within the SPYDER language.

4.4 Targeted Synthesis for Spyder

In this section, we detail the automatic enforcing of data invariants. We motivate and formalize the problem in subsection 4.4.1, then, in subsection 4.4.2 we present its solution in the TARGETED SYNTHESIS algorithm. We prove the algorithm sound in subsection 4.4.3.

Recall the budgeting example introduced in subsection 4.1.1 and in particular the specific case of the unit-conversion data invariant, which establishes a unit-conversion invariant between daily and weekly values. Throughout this section we will demonstrate our algorithm on this invariant.

4.4.1 Automatic Enforcement of Data Invariants

Let Π be a Spec term, and S be a SPYDER statement (i.e. a Stmt term). We say that Π is a *data invariant* for S if and only if S maintains Π:

$$\langle \Pi \rangle \; S \; \langle \Pi \rangle.$$

For example, the specification `foreach x in xs: x.val > 0` is a data invariant for a loop which increments each value of `xs`, `for x in xs: x <- x.val + 1`, but it is not a data invariant for decrement loop `for x in xs: x <- x.val - 1`. This definition extends straightforwardly to statement blocks B.

Let B, B' be two SPYDER blocks. We say that a block B' is an *extension* of B ($B \prec B'$) if B and B' have identical semantics on variables modified by B.

An *invariant enforcement* problem is a pair $\langle B, \Pi \rangle$ of a block B and a specification Π.

A solution to the enforcement problem is a block B' such that $B \prec B'$ and $\langle\Pi\rangle\ B'\ \langle\Pi\rangle$. In other words, the goal is to find an extension of B such that Π is a data invariant for the extended block.

In our example, we wish the unit-conversion invariant on lines 4 and 5 to be a data invariant. This means the invariant enforcement problem is to enforce this specification on the body of `adjustForCOLA`.

To find a solution, our algorithm analyses B and insert local patches whenever the invariant needs to be restored. Since there are many candidate patches to explore, the key challenge is to make the search efficient. To this end, our algorithm: (1) a-priori restricts the search to extensions of B, by keeping track of the set of variables that a patch is allowed to modify; (2) *targets* the invariant Π to B, producing a specification for each patch that is as local as possible.

In this example, because `adjustForCOLA` modifies the elements of `days`, our algorithm must find an extension that has an equivalent effect on `days`. Further, since `adjustForCOLA` iterates over `days`, our algorithm will *target* the data invariant on lines 4 and 5 to a local specification, specific to just the loop body on lines 9 and 10. We next explain the details of our algorithm.

4.4.2 Targeted Synthesis Algorithm

We formalize TARGETED SYNTHESIS as a *completion judgment* $\texttt{md} \vdash \langle\Pi\rangle\ B\ \langle\Phi\rangle \hookrightarrow B'$. Intuitively, given a pre- and post-condition Π and Φ, and the set of variables `md` modified so far, an input block B should be completed into B'. In this case, we say that B' is a *completion* for B, and the intension is that B' satisfies the specification ($\langle\Pi\rangle\ B'\ \langle\Phi\rangle$) and does not modify any variables in `md` (i.e. $\text{mod}(B) \cap \texttt{md} = \emptyset$). We present the inference rules for this judgment in Figure 4.10.

$$\textsc{Synth-Base}\;\dfrac{\begin{array}{c}\text{cands} = \{v \mid v \sim_{\Pi} y,\ y \in \mathtt{md}\} \\ \text{mod}(B) \subseteq (\text{cands} \setminus \mathtt{md}),\ \langle\Pi\rangle\ B\ \langle\Phi\rangle\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ \mathtt{skip}\ \langle\Phi\rangle \hookrightarrow B}$$

$$\textsc{Synth-Loop}\;\dfrac{\begin{array}{c}(\textit{fresh}\ \mathsf{v}) \qquad\qquad\qquad\qquad \mathsf{u}_i \in (\mathtt{cn} \cap \mathtt{md}) \\ \mathtt{md} \vdash \langle\Pi\rangle\ \mathsf{for}\{(\mathsf{v} : \mathsf{u}_i)\}\mathtt{skip}\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\phi \wedge \Phi\rangle \hookrightarrow B\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ \mathtt{skip}\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\phi \wedge \Phi\rangle \hookrightarrow B}$$

$$\textsc{Assign}\;\dfrac{\begin{array}{c}(\textit{fresh}\ \mathsf{v}') \\ \mathtt{md} \cup \{\mathsf{v}\} \vdash \langle \exists \mathsf{v}'.\Pi[\mathsf{v} \mapsto \mathsf{v}'] \wedge \mathsf{v} = E[\mathsf{v} \mapsto \mathsf{v}']\rangle\ B\ \langle\Phi\rangle \hookrightarrow B'\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ \mathsf{v} := E\,; B\ \langle\Phi\rangle \hookrightarrow \mathsf{v} := E\,; B'}$$

$$\textsc{Put}\;\dfrac{\begin{array}{c}(\textit{fresh}\ \mathsf{v}') \\ \mathtt{md} \cup \{\mathsf{v}\} \vdash \langle \exists \mathsf{v}'.\Pi[\mathsf{v} \mapsto \mathsf{v}'] \wedge \mathsf{v} = E[\mathsf{v} \mapsto \mathsf{v}']\rangle\ B\ \langle\Phi\rangle \hookrightarrow B'\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ \mathsf{v} \leftarrow E\,; B\ \langle\Phi\rangle \hookrightarrow \mathsf{v} \leftarrow E\,; B'}$$

$$\textsc{Inv}\;\dfrac{\begin{array}{c}\mathtt{md} \vdash \langle\Pi\rangle\ \mathtt{skip}\ \langle\Phi\rangle \hookrightarrow B' \\ \{\} \vdash \langle\Phi\rangle\ B\ \langle\Phi\rangle \hookrightarrow B''\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ B\ \langle\Phi\rangle \hookrightarrow B' \mathbin{+\!\!+} B''}$$

$$\textsc{For-Extend}\;\dfrac{\begin{array}{c}\mathsf{u}_i \sim_{\Pi} \mathsf{u} \qquad\qquad (\textit{fresh}\ \mathsf{v}) \qquad\qquad \mathsf{u} \notin \overline{\mathsf{u}_i} \\ \mathtt{md} \vdash \langle\Pi\rangle\ \mathsf{for}\ \overline{(\mathsf{v}_i, \mathsf{u}_i)} \cup \{(\mathsf{v}, \mathsf{u})\}\ B_i\,;\ B\ \langle\Phi\rangle \hookrightarrow B'\end{array}}{\mathtt{md} \vdash \langle\Pi\rangle\ \mathsf{for}\ \overline{(\mathsf{v}_i, \mathsf{u}_i)}\ B_i\,;\ B\ \langle\Phi\rangle \hookrightarrow B'}$$

$$\textsc{Foreach-Extend}\;\dfrac{\begin{array}{c}\overline{\mathsf{y}_i} \cap \overline{\mathsf{u}_i} \neq \varnothing \\ \phi' = \text{merge}(\mathsf{foreach}\overline{(\mathsf{a}_i, \mathsf{y}_i)}\ \phi_l, \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi_r) \\ \{\} \vdash \langle \phi' \wedge \Phi\rangle\ B\ \langle \phi' \wedge \Phi\rangle \hookrightarrow B'\end{array}}{\begin{array}{c}\{\} \vdash \\ \langle \mathsf{foreach}\overline{(\mathsf{x}_i, \mathsf{y}_i)}\ \phi_l \wedge \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi_r \wedge \Phi\rangle \\ B \\ \langle \mathsf{foreach}\overline{(\mathsf{x}_i, \mathsf{y}_i)}\ \phi_l \wedge \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi_r \wedge \Phi\rangle \\ \hookrightarrow B'\end{array}}$$

$$\textsc{For-Specialize}\;\dfrac{\begin{array}{c}\overline{\mathsf{u}_i} \subseteq \overline{\mathsf{y}_i} \qquad\qquad\qquad \phi' = \text{weaken\_prev}(\phi) \\ \text{mod}(B_i) \vdash \langle \phi'[\mathsf{v}_i \mapsto \mathsf{x}_i] \wedge \Phi\rangle\ \mathtt{skip}\ \langle \phi[\mathsf{v}_i \mapsto \mathsf{x}_i] \wedge \Phi\rangle \hookrightarrow B_{pre} \\ \{\} \vdash \langle\ \phi[\mathsf{v}_i \mapsto \mathsf{x}_i] \wedge \Phi\rangle\ B_i\ \langle \phi[\mathsf{v}_i \mapsto \mathsf{x}_i] \wedge \Phi\rangle \hookrightarrow B_i' \\ \{\} \vdash \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle\ B\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle \hookrightarrow B'\end{array}}{\begin{array}{c}\{\} \vdash \\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle \\ \mathsf{for}\ \overline{(\mathsf{x}_i, \mathsf{y}_i)} B_i\,; B \\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle \\ \hookrightarrow \mathsf{for}\ \overline{(\mathsf{x}_i, \mathsf{y}_i)}(B_{pre} \mathbin{+\!\!+} B_i')\,;\ B'\end{array}}$$

$$\textsc{Conditional}\;\dfrac{\begin{array}{c}\{\} \vdash \langle E \wedge \Phi\rangle\ B_t\ \langle\Phi\rangle \hookrightarrow B_t' \\ \{\} \vdash \langle \neg E \wedge \Phi\rangle\ B_f\ \langle\Phi\rangle \hookrightarrow B_f' \\ \{\} \vdash \langle\Phi\rangle\ B\ \langle\Phi\rangle \hookrightarrow B'\end{array}}{\{\} \vdash \langle\Phi\rangle\ \mathsf{if}\ E\ \mathsf{then}\ B_t\ \mathsf{else}\ B_f\,; B\ \langle\Phi\rangle \hookrightarrow \mathsf{if}\ E\ \mathsf{then}\ B_t'\ \mathsf{else}\ B_f'\,; B'}$$

Figure 4.10. Inference rules for SPYDER algorithm, with explicit blocks.

Patch Generation

The rule `Synth-Base` fires once we reach the end of the input block and performs the actual patch generation. It non-deterministically picks a patch satisfying the specification, and can only update "stale" variables, which are not modified but depend on modified variables via the specification Π (we formalize this dependency in Figure 4.11). Our implementation realizes the non-deterministic choice via constraint-based synthesis in the space of all blocks that only contain assignments and put-statements. `Synth-Loop` is similar to `Synth-Base` but allows generating looping patches when the postcondition contains quantification.

Accumulating Modifications

`Assign` and `Put` simply accumulate modifications made by the input block. In these rules, the variable modified by the current statement is added to `md`, and the precondition of the subproblem is updated to reflect the result of the modification. Note that while the top-level completion problem is always *symmetric* (*i.e.* of the form $\texttt{md} \vdash \langle\Pi\rangle\, B\, \langle\Pi\rangle$, where Π is the data invariant we are trying got enforce), the pre- and the post-condition might become different as a result of applying `Assign` or `Put`. Sometimes these differences must be reconciled, because rules like `For-Specialize` only apply to symmetric goals. The rule `Inv` allow us to do just that: restore the invariant Φ by inserting a patch in the middle of a block.

Targeting

The central rule of our system is `For-Specialize`. If a data invariant and a loop have the same syntactic structure (i.e. iterate over the same collections), this rule *targets* the data invariant to the loop body: *i.e.* strips both loop and quantification from the subgoal. One complication here is the role of prev terms. As discussed in section 4.3, terms with prev cannot be used as an assumption for the body of a targeted loop. In this case, we first patch the current loop iteration into the term B_{pre}, and then continue to the remainder of the loop body.

Alignment

Finally, a crucial necessity for the `For-Specialize` rule is that the data invariant and the loop are syntactically similar. To reach this state, the `Foreach-Extend` and `For-Extend` rules syntactically search for an alignment. Both of these rules are semantics-preserving and are performed so that the Targeting rule can be applied.

Patching the Example

We next give a derivation for a patch for the running example, in which we extend the loop by iterating over `weeks` and introduce a maintenance Put to the new `weeks` iterator.

Recall that we wish the unit-conversion invariant on lines 4 and 5 to be a data invariant for the body of `adjustForCOLA`, lines 8 through 10.

In this case, the pre- and post-conditions are

```
foreach w in weeks, d in days: 7 * d.val = w.val,
```

and the block to be patched is

```
for d in days: if (d.val > 0): d <- d.val * cola;
```

First, to make the loop iterate over the same variables as the `foreach` term, we introduce a new iterator over `weeks` by applying `For-Extend`, producing the new loop

```
for w in weeks, d in days: ...
```

Next, we target the specification to the loop by applying `For-Specialize`, which has the effect of stripping the `foreach` and `for` terms. As a consequence our new data invariant is `7 * d.val = w.val`, and our new block is `if (d.val > 0): d <- d.val * cola`.

We next apply `Conditional` to simplify the loop. The false-branch is empty and so

$$\text{Rel-Expr}\frac{\text{atom}(\pi) \quad \mathsf{x},\mathsf{y} \in \text{free}(\pi)}{\mathsf{x} \sim_\pi \mathsf{y}} \qquad \text{Rel-Left}\frac{\mathsf{x} \sim_L \mathsf{y}}{\mathsf{x} \sim_{L \wedge R} \mathsf{y}}$$

$$\text{Rel-Right}\frac{\mathsf{x} \sim_R \mathsf{y}}{\mathsf{x} \sim_{L \wedge R} \mathsf{y}} \qquad \text{Rel-Trans}\frac{\mathsf{x} \sim_L \mathsf{y} \quad \mathsf{y} \sim_R \mathsf{t}}{\mathsf{x} \sim_{L \wedge R} \mathsf{t}}$$

Figure 4.11. Inference rules for variable data-dependency relation. We relate two variables x and y by $\sim$ if a modification to x might affect y.

satisfies the data invariant. We now only need to patch the true-branch.

Because the statement is a Put, we apply the `Put` rule, which logically embeds the effects of `d <- d.val * cola` into the precondition, and adds `d` to the set of modified variables `md`. At this point, we're left with a logical specification, an empty block, and a set of modified variables with just one member, `md` $= \{d\}$.

Finally, we apply two rules. First, we find a maintenance patch for the data invariants by the `Synth-Base` rule. This produces a snippet B' (in this case `w <- d.val * 7;`) such that if we add B' at line 11, the resulting conditional (and loop) will maintain the invariant. We will discuss this further in a moment, but for now, we will produce an extension from B' and the current block `d <- d.val * cola;` using the `Inv` rule.

Now we demonstrate how to find B' using the `Synth-Base` rule. In this case, because `w` and `d` both appear in the precondition, and `d` is in `md` the candidate variables for a patch are $\{w,d\}$. However, since B' is not allowed to modify any of the variables in `md` (i.e. `d`), it's forced to produce a patch that modifies `w`, which further satisfies the invariant `w.val = d.val * 7`. One such patch is `w <- d.val * 7;`, and so the `Synth-Base` rule calculates this patch for B'.

4.4.3 Soundness of Synthesis Rules

In all cases, if the SPYDER extension rules produce a new program, the program must satisfy the input data invariants. We formalize the synthesis soundness using the axiomatic semantics of section 4.3:

Theorem 4.4.1 (Soundness of TARGETED SYNTHESIS)**.**

$$\forall \Pi, B, B' . \emptyset \vdash \langle \Pi \rangle B \langle \Pi \rangle \hookrightarrow B' \quad \implies \langle \Pi \rangle B' \langle \Pi \rangle$$

We prove this by generalizing to $\mathtt{md} \vdash \langle \Pi \rangle B \langle \Phi \rangle \hookrightarrow B'$ and then by induction on the derivation. More detail is in subsection 4.6.5 and the proof is straightforward.

4.5 Evaluation

In this section, we detail the experiments run to evaluate SPYDER. We assessed SPYDER quantitatively via a set of benchmarks and using several case studies.

Research questions

We test the following questions:

(RQ1) Is programming with SPYDER and data invariant is more succinct (and therefore easier) than maintaining data invariants manually?

(RQ2) Does SPYDER make code evolution easier? We test this by examining the necessary changes to implementation and invariants in order to implement new functionality.

(RQ3) Does TARGETED SYNTHESIS enable fast, scalable synthesis? To test this, we measure the performance of synthesizing with SPYDER.

Implementation

We evaluate SPYDER and TARGETED SYNTHESIS using a prototype compiler that targets Boogie [71]. Our prototype implements the contents of section 4.3 by compiling to Boogie, and we implement the contents of section 4.4 by extending SPYDER terms using our own synthesis and CEGIS algorithms.

4.5.1 Case Studies

We first examine RQ1 and RQ2 using three detailed case studies.

The invariant language of SPYDER, targeted towards expressing relations over collections, is a perfect fit for many useful idioms in web programming. Using SPYDER, we implemented three applications inspired by real-life web programs.

Game of Life

John Conway's Game of Life [27] is a popular visualization of a cellular automaton with applications in Chemistry, Physics, Math, and Computer Science. In this game, a discrete world of cells obeys particular evolutionary behavior. At each time step of the application, the cells in the world change state according to the rules of the game. We looked at several interactive applications of the game of life online, such as [4]. In all of these applications, the programmer manually maintained an invariant between the visual cells of the board and the internal data structure for the cells. To implement this in SPYDER, we encoded the internal state of the game and its visual state as two integer arrays. An element-wise invariant relates the internal state of the game to its visual state. We implemented procedures for 1. making transitions in the internal state according to the rules of the game, 2. interactive logic that allows the user to change the state of a cell by clicking on the board, and 3. a buttom for starting and stopping the game. SPYDER synthesized a patch that re-synchronizes the model and the view for each of these procedures.

Budgeting Application

Our second case study is a spreadsheet-style budgeting application, described in detail in subsection 4.1.1. For this benchmark, the programmer builds a financial application which takes in periodic revenues and deficits. This application takes amounts in three periodic intervals—weekly, monthly, and yearly—and converts between the amounts. In this way, the end-user can input data in the most convenient format.

A difficult feature of this benchmark was summing up the rows of the budget and presenting a total value. In traditional programming, this would require a procedure and would not be easy to compose. In contrast, in SPYDER, this invariant is easily expressible using the `prev` calculus and indeed composes very well with the other invariants of the system.

Shared Expenses Application

Our final case study is an extension of the Budgeting Application. Anecdotally, one of the co-authors actually uses this type of application in real-life. The idea here is that two people who live in the same household want to split shared expenses equally at the end of the month. In this application, each row has 4 entries: in the first two cells store the expenses paid by person A and person B, respectively; in the third cell, stores the average cost for the expense (i.e. the final cost for each person), and in the fourth cell, the amount person A owes to person B (i.e. how much person B over/underpaid on the particular expense). Similar to the budgeting application, we can express each row of this application in SPYDER and further, we can conditionally render the amount owed between the participants.

4.5.2 Quantitative Evaluation

In addition to the qualitative evaluation, we empirically evaluate questions 1-3 on a series of benchmarks and compare them to two traditional techniques, manual invariant maintenance, and dynamic maintenance of functional specifications (i.e. Functional-Reactive Programming, FRP).

To compare SPYDER against these two techniques in a language-agnostic, apples-to-apples way, we implement each benchmark in all three paradigms using SPYDER's syntax. For the imperative paradigm, we manually maintain invariants without using specifications. For the FRP paradigm, we write functional specifications for each variable in the program.

RQ1: Succinctness

We measure the amount of code necessary to implement a set of benchmarks in three different techniques: manually (Imperative), FRP and SPYDER. We show the results in Table 4.1. As expected, the size of manually implemented code for both FRP and SPYDER is considerably smaller than Imparative. However, SPYDER specifications are as much as three times smaller

Table 4.1. Benchmarks comparing implementation in SPYDER to other techniques. *Impl* is the size of implementation (non-invariant) code and *Spec* is the size of invariants. All sizes are in AST nodes, and all implementations are in the SPYDER language. Each benchmark has invariants maintained manually (*Imp*), in the *FRP* paradigm, and with SPYDER. Synthesis times of SPYDER are given compared to Sketch (*SK*), on collections of size *3*, *10*, and *50* (*SK-3*, *SK-10* and *SK-50* resp.). We report a timeout (−) after ten minutes and we use N/A to denote a Sketch program that doesn't use collections. *Patches* reports the size of patches synthesized by SPYDER and the number of patches (*locs*) per benchmark.

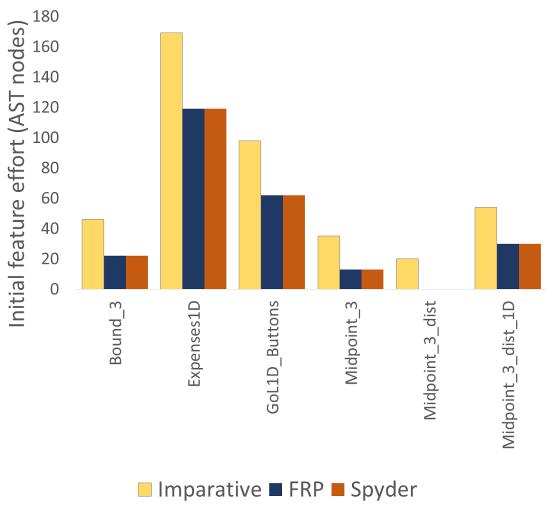
	Benchmark	Imp Size		FRP Size		Spy Size		Synthesis Time				Patches	
		Impl	*Spec*	*Impl*	*Spec*	*Impl*	*Spec*	*SK-3*	*SK-10*	*SK-50*	*Spy*	locs	size
Arithmetic	Midpoint_2	39	0	27	13	27	7	1.04	N/A	N/A	58.01	2	298
	Midpoint_3	64	0	40	25	40	9	1.44	N/A	N/A	27.78	3	123
	Midpoint_3_dist	76	0	40	81	40	22	0.80	N/A	N/A	123.24	3	489
	Midpoint_3_dist_1D	127	0	67	105	67	42	32.52	322.82	303.26	97.79	3	483
	Bound_2	39	0	27	13	27	5	0.36	N/A	N/A	116.20	2	198
	Bound_3	64	0	40	37	44	10	0.92	N/A	N/A	348.68	3	418
Web Apps	GoL1D	279	0	255	25	255	13	0.74	0.84	1.15	38.77	4	30
	GoL1D_Buttons	373	0	317	54	317	46	1.08	0.27	1.29	49.38	8	194
	Expenses	59	0	43	17	43	9	29.09	N/A	N/A	107.58	2	233
	Expenses_1D	170	0	126	37	126	19	-	-	-	105.93	3	190
	Overview	105	0	63	151	63	82	-	-	-	106.25	2	820

than FRP specifications. Additionally, we see that patches are generated in a number of locations. This means manually maintainaing the invariants would have required to keep track of all these locations. We do see that the size of the patches generated by SPYDER is much larger than the size of the manual implementation. There are two main reasions fo this: 1) SPYDER patches are not meant for human consumption and so are unoptimized, and 2) patches are synthesized in the target language (i.e., Boogie), which is not as concise as SPYDER. The results show that SPYDER invariants provide a succinct way of specifying what would otherwise be a much larger piece of enforcement code.

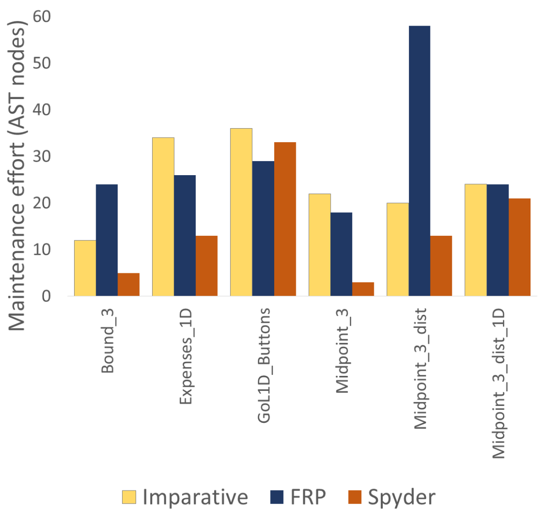
RQ2: Ease of modification

We measure the amount of modification required to evolve existing code, again comparing SPYDER to imperative and FRP invariant maintenance. Figure 4.12a shows for each benchmark the size of the modification (in AST nodes) required to implement the new functionality portion of a new feature, without fixing broken data invariants, and Figure 4.12b shows the size of the modification to invariant-preserving code.

As seen in our discussion of RQ1, we see in Figure 4.12a that writing new functionality



(a) Initial implementation effort: diff size to implement a new feature (in AST nodes) while breaking invariants in all three techniques.



(b) Invariant maintenance effort: effort repairing broken invariants. For FRP and SPYDER indicates updates to invariants and for Imperative this is manually written invariant maintenance code.

Figure 4.12. Effort required to add new features to existing programs.

with SPYDER is more succinct than either writing it imperatively in the SPYDER language. Figure 4.12b shows that the same is true for invariant maintenance: modifications to the invariants in SPYDER are considerably smaller than the manual changes in imp and the changes to code and invariants in FRP.

These benchmarks show that code evolution is also easier in SPYDER.

RQ3: Performance

We evaluated the scalability of the SPYDER compiler (and by extension, the TARGETED SYNTHESIS algorithm) by compiling our benchmarks and comparing the performance against a standard synthesis technique, Sketch [114]. For each of our benchmarks, we reimplemnted the benchmark in Sketch and compared the performance. In contrast to TARGETED SYNTHESIS, Sketch performs bounded enumeration for verification, and as a consequence, quantified invariants scale in proportion to the size of the verified array. To measure the scalability of bounded verification, for Sketch programs with arrays we varied the number of elements in the concrete

Sketch arrays from 3 elements to 50 elements.

Overall, we find that Sketch outperforms SPYDER on the synthetic benchmarks but does not complete within the time limit in two of of our three case studies. For the case studies, Sketch could solve these problems if the programmer wrote a synthesis template tailored to the specific study. In contrast, SPYDER programmers do not have to develop an application-specific sketch.

4.6 Spyder Appendix

4.6.1 ImpArray Syntax and Semantics

$$
\begin{array}{lcl}
 & & \mathbf{v}, \mathbf{u} \in \mathbf{Vars}, \quad \mathbf{i} \in \mathbb{Z} \\
\mathbf{Stmt} & ::= & \mathbf{v} := \mathbf{Expr} \\
 & | & \mathbf{v}\,[\,\mathbf{Expr}\,] := \mathbf{Expr} \\
 & | & \texttt{if } \mathbf{Expr} \texttt{ then } \mathbf{Stmt} \texttt{ else } \mathbf{Stmt} \\
 & | & \mathsf{while}\ \ \mathbf{Expr}\ \mathbf{Stmt} \\
 & | & \mathbf{Stmt}\ ;\ \mathbf{Stmt} \\
 & | & \texttt{skip} \\
\mathbf{Expr} & ::= & \mathbf{v} \mid \mathbf{i} \mid \mathsf{true} \mid \mathsf{false} \\
 & | & \mathbf{Expr}\ \mathbf{bop}\ \mathbf{Expr} \\
 & | & \texttt{if } \mathbf{Expr} \texttt{ then } \mathbf{Expr} \texttt{ else } \mathbf{Expr} \\
 & | & \mathbf{uop}\ \mathbf{Expr} \\
 & | & \mathbf{Expr}\,[\,\mathbf{Expr}\,] \\
 & | & \mathsf{size}(\mathbf{v}) \\
 & | & \forall\ \mathbf{v}\ .\ \mathbf{Expr} \\
 & | & \exists\ \mathbf{v}\ .\ \mathbf{Expr} \\
\mathbf{bop} & ::= & + \mid \times \mid \% \mid \implies \mid \iff \mid \ldots \\
\mathbf{uop} & ::= & \neg \mid !
\end{array}
$$

Figure 4.13. Syntax for the IMP-ARRAY language.

4.6.2 Spyder Semantics

Let σ be a IMP-ARRAY state, E a SPYDER Expression, and Γ a well-formed translation context with respect to E. We define the denotational semantics of E as the denotational semantics of the corresponding IMP-ARRAY expression:

$$\begin{array}{lcc}
[\![\mathbf{v}]\!]_\sigma & = & \sigma[\mathbf{v}] \\
[\![\mathbf{i}]\!]_\sigma & = & \mathbf{i} \\
[\![\mathsf{true}]\!]_\sigma & = & \top \\
[\![\mathsf{false}]\!]_\sigma & = & \bot \\
[\![E_l\,\mathbf{bop}\,E_r]\!]_\sigma & = & [\![E_l]\!]_\sigma\mathbf{bop}[\![E_r]\!]_\sigma \\
[\![\mathbf{uop}\,E_i]\!]_\sigma & = & \mathbf{uop}[\![E_i]\!]_\sigma \\
[\![\mathbf{v}[E]]\!]_\sigma & = & [\![\mathbf{v}]\!]_\sigma[[\![E]\!]_\sigma] \\
[\![\mathsf{size}(E)]\!]_\sigma & = & \|[\![E]\!]_\sigma\| \\
[\![\forall\mathbf{v}\,.\,E]\!]_\sigma & = & \forall x \in \sigma\,.\,[\![E[v \mapsto x]]\!]_\sigma = \top
\end{array}$$

Figure 4.14. Denotational semantics for IMP-ARRAY expressions

Definition 1 (Spyder Expression Semantics)**.**

$$[\![E]\!]_\sigma \quad ::= \quad [\![\texttt{trans}(E,\,\Gamma)]\!]_\sigma$$

We similarly define the operational semantics of a SPYDER statement S as the operational semantics of the corresponding IMP-ARRAY statement $\texttt{trans}(S,\,\Gamma)$:

Definition 2 (Spyder Statement Semantics)**.**

$$\frac{\texttt{trans}(S,\,\Gamma)\,,\,\sigma \rightsquigarrow \sigma'}{S\,,\,\sigma \rightsquigarrow \sigma'}$$

4.6.3 Proofs: Soundness of Spyder Triples

Lemma 4.6.1 (Bindings)**.**

$$\forall P,\,B,\,\Gamma\,.\texttt{wf}(\mathsf{for}(\mathsf{x},\mathsf{y})B,\,\Gamma) \wedge \texttt{wf}(P,\,\Gamma) \implies \mathsf{x} \notin \mathrm{free}(P)$$

Proof. Induction over the derivation of $\texttt{wf}(P,\,\Gamma)$. □

Lemma 4.6.2 (Assignment)**.**

$$\forall B,\,\Gamma\,.\texttt{wf}(\mathsf{for}(\mathsf{x},\mathsf{y})B,\,\Gamma) \implies x \notin \mathrm{assign}(B)$$

$$\frac{}{\mathbf{v} := \mathbf{Expr},\ \sigma \ \rightsquigarrow \sigma[\mathbf{v} \mapsto [\![\mathbf{Expr}]\!]_\sigma]}$$

$$\frac{}{\mathbf{v}\,[\,\mathbf{Expr}_1\,] := \mathbf{Expr}_2\,,\ \sigma \ \rightsquigarrow \sigma[\mathbf{v} \mapsto \mathbf{v}[[\![\mathbf{Expr}_1]\!]_\sigma \mapsto [\![\mathbf{Expr}_2]\!]_\sigma]}$$

$$\frac{\begin{array}{c}[\![\mathbf{Expr}]\!]_\sigma \ = \top \\ \mathbf{Stmt}_1\,,\ \sigma \rightsquigarrow \sigma'\end{array}}{\texttt{if}\ \mathbf{Expr}\ \texttt{then}\ \mathbf{Stmt}_1\ \texttt{else}\ \mathbf{Stmt}_2\,,\ \sigma \ \rightsquigarrow \sigma'}$$

$$\frac{\begin{array}{c}[\![\mathbf{Expr}]\!]_\sigma \ = \bot \\ \mathbf{Stmt}_2\,,\ \sigma \rightsquigarrow \sigma'\end{array}}{\texttt{if}\ \mathbf{Expr}\ \texttt{then}\ \mathbf{Stmt}_1\ \texttt{else}\ \mathbf{Stmt}_2\,,\ \sigma \ \rightsquigarrow \sigma'}$$

$$\frac{[\![\mathbf{Expr}]\!]_\sigma \ = \top}{\mathsf{while}\ \ \mathbf{Expr}\ \mathbf{Stmt}\,,\ \sigma \ \rightsquigarrow \sigma}$$

$$\frac{\begin{array}{c}[\![\mathbf{Expr}]\!]_\sigma \ = \bot \\ \mathbf{Stmt}\,;\ \mathsf{while}\ \mathbf{Expr}\ \mathbf{Stmt}\,, \sigma \ \rightsquigarrow \sigma'\end{array}}{\mathsf{while}\ \ \mathbf{Expr}\ \mathbf{Stmt}\,,\ \sigma \ \rightsquigarrow \sigma}$$

$$\frac{\mathbf{Stmt}_1\,,\ \sigma \ \rightsquigarrow \ \sigma'\mathbf{Stmt}_2\,,\ \sigma' \ \rightsquigarrow \ \sigma''}{\mathbf{Stmt}_1\ ;\ \mathbf{Stmt}_2\,,\ \sigma \ \rightsquigarrow \sigma''}$$

$$\frac{}{\texttt{skip}\,,\ \sigma \ \rightsquigarrow \sigma}$$

Figure 4.15. Operational semantics for IMP-ARRAY statements

Proof. Induction over the derivation of $\texttt{wf}(\mathsf{for}(\mathsf{x},\mathsf{y})B,\ \Gamma)$. □

Lemma 4.6.3 (Array substitution)**.**

$$\forall P, \mathsf{x}, \mathsf{y}, \Gamma\,.\,\texttt{wf}(P,\ \Gamma) \wedge \Gamma(\mathsf{x}) = \mathsf{y} \implies$$
$$\forall \sigma\,.\,\sigma(\texttt{trans}(P,\ \Gamma)[\mathbf{y} \mapsto \mathbf{y}']) \implies \sigma(\texttt{trans}(P,\ \Gamma)[\mathbf{y}[\mathbf{x}] \mapsto \mathbf{x}'])$$

Proof. Structural induction over P. □

Theorem 4.6.4 (Relative Soundness)**.**

$$\forall P, S, Q, \Gamma\,.\,\texttt{wf}(P \wedge Q,\ \Gamma) \wedge \texttt{wf}(S,\ \Gamma) \implies$$
$$\langle P \rangle\, S\, \langle Q \rangle \implies \{\texttt{trans}(P,\ \Gamma)\}\, \texttt{trans}(S,\ \Gamma)\, \{\texttt{trans}(Q,\ \Gamma)\}$$

$$\text{Consequence}\ \frac{P \implies P' \quad Q' \implies Q \quad \{P'\}\, S\, \{Q'\}}{\{P\}\, S\, \{Q\}}$$

$$\text{Skip}\ \frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\text{Sequence}\ \frac{\{P\}\, S_1\, \{Q\} \quad \{Q\}\, S_2\, \{R\}}{\{P\}\, S_1\, ;\, S_2\, \{R\}}$$

$$\text{Conditional}\ \frac{\{P \land e\}\, S_t\, \{Q\} \quad \{P \land \neg\, e\}\, S_f\, \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ S_t\ \texttt{else}\ S_f\ \{Q\}}$$

$$\text{Assign-Var}\ \frac{(\mathit{fresh}\, v')}{\{P\}\ v := E\ \{\exists v' . P[v \mapsto v'] \land v = E[v \mapsto v']\}}$$

$$\text{Assign-Array}\ \frac{(\mathit{fresh}\, v')}{\{P\}\ v[\,E_i\,] := E_r\ \{\exists v' . P[v \mapsto v'] \land v = v'[E_i[v \mapsto v'] {:=} E_r[v \mapsto v']]\}}$$

$$\text{While}\ \frac{\{I \land E\}\, S\, \{I\}}{\{I\}\ \mathsf{while}\, E\, S\ \{I \land \neg E\}}$$

Figure 4.16. Standard axiomatic semantics (Hoare logic) for IMP-ARRAY.

Proof. By induction over the derivation of $\langle P\rangle\, S\, \langle Q\rangle$; for each case of S, we build a corresponding derivation for $\{\texttt{trans}(P,\ \Gamma)\}\, \texttt{trans}(S,\ \Gamma)\, \{\texttt{trans}(Q,\ \Gamma)\}$.

In all cases we start by assuming $\texttt{wf}(P \land Q,\ \Gamma) \land \texttt{wf}(S,\ \Gamma)$.

Cases of S:

1. Base case, in which the last step of the derivation is `Skip`: $\langle P\rangle\, \texttt{skip}\, \langle Q\rangle$. From the structure of `Skip`, it must be the case that P and Q are structurally identical, i.e. the derivation is $\langle P\rangle\, \texttt{skip}\, \langle P\rangle$. Since `trans` is a function, it maps `skip` to exactly one statement (namely `skip`), and P to exactly one expression $\texttt{trans}(P,\ \Gamma)$. Finally, we apply the `Skip` Hoare rule to obtain $\{\texttt{trans}(P,\ \Gamma)\}\, \texttt{skip}\, \{\texttt{trans}(P,\ \Gamma)\}$.

2. Inductive case, in which the last step of the derivation is `Consequence`: $\langle P \rangle \, S \, \langle Q \rangle$. We will use the corresponding `Consequence` rule of Hoare logic to build a derivation for $\{\texttt{trans}(P,\,\Gamma)\}\,\texttt{trans}(S,\,\Gamma)\,\{\texttt{trans}(Q,\,\Gamma)\}$.

   Since the case is `Consequence`, there must be P' and Q' such that $P \implies P'$, $Q' \implies Q$, and $\langle P' \rangle \, S \, \langle Q' \rangle$. From Definition 2, we know that

   $$\texttt{trans}(P,\,\Gamma) \implies \texttt{trans}(P',\,\Gamma), \quad \text{and} \quad \texttt{trans}(Q',\,\Gamma) \implies \texttt{trans}(Q,\,\Gamma).$$

   From the inductive hypothesis, we have the ImpArray triple

   $$\{\texttt{trans}(P',\,\Gamma)\}\,\texttt{trans}(S,\,\Gamma)\,\{\texttt{trans}(Q',\,\Gamma)\},$$

   and so we apply the `Consequence` ImpArray rule to obtain

   $$\{\texttt{trans}(P,\,\Gamma)\}\,\texttt{trans}(S,\,\Gamma)\,\{\texttt{trans}(Q,\,\Gamma)\}.$$

3. Inductive case, in which the last step of the derivation is `Conditional`:

   $$\langle P \rangle \; \texttt{if} \; E \; \texttt{then} \; S_t \; \texttt{else} \; S_f \; \langle Q \rangle.$$

   This follows from the inductive hypothesis applied to E, S_t, and S_f, as well as the `Conditional` ImpArray rule.

4. Inductive case, in which the last step of the derivation is `Sequence`: $\langle P \rangle \; S_1 \; ; \; S_2 \; \langle R \rangle$. This follows from the inductive hypothesis applied to S_1, and S_2, as well as the `Sequence` ImpArray rule.

5. Inductive case, in which the last step of the derivation is `Assign`: $\langle P \rangle \; \texttt{v} := E \; \langle Q \rangle$. In this case, the translation produces a Imp assignment to `v`.

Since the last step is `Assign`, there must be a fresh variable v' such Q is the strongest postcondition of the assignment to v:

$$\exists \mathsf{v}' . P[\mathsf{v} \mapsto \mathsf{v}'] \wedge \mathsf{v} = E[\mathsf{v} \mapsto \mathsf{v}']$$

From the inductive hypothesis, we know that translating the Spyder triple produces an equivalent Imp Hoare triple

$$\{\mathtt{trans}(P, \Gamma)\} \ \mathbf{v} := \mathtt{trans}(E, \Gamma) \ \{\mathtt{trans}(\exists \mathsf{v}' . P[\mathsf{v} \mapsto \mathsf{v}'] \wedge \mathsf{v} = E[\mathsf{v} \mapsto \mathsf{v}'], \Gamma)\}.$$

If you consider the translated term $\mathtt{trans}(\exists \mathsf{v}' . P[\mathsf{v} \mapsto \mathsf{v}'] \wedge \mathsf{v} = E[\mathsf{v} \mapsto \mathsf{v}'], \Gamma)$, using Lemma 4.6.3 and the definition of translation, you'll find that it is exactly the ImpArray postcondition for assignment with $\mathtt{trans}(P, \Gamma)$ as a precondition:

$$\exists \mathbf{v}' . \mathtt{trans}(P, \Gamma)[\mathbf{v} \mapsto \mathbf{v}'] \wedge \mathbf{v} = \mathtt{trans}(E, \Gamma)[\mathbf{v} \mapsto \mathbf{v}'].$$

So, we apply `Assign` with P as a precondition to obtain

$$\{\mathtt{trans}(P, \Gamma)\}\mathbf{v} := \\ \mathtt{trans}(E, \Gamma) \{\exists \mathbf{v}' . \mathtt{trans}(P, \Gamma)[\mathbf{v} \mapsto \mathbf{v}'] \wedge \mathbf{v} = \mathtt{trans}(E, \Gamma)[\mathbf{v} \mapsto \mathbf{v}']\},$$

6. Inductive case, in which the last step of the derivation is `Put`: $\langle P \rangle \ \mathsf{v} \leftarrow E \ \langle Q \rangle$. For this, we will show that the translation of the put $\mathsf{v} \leftarrow E$ takes the precondition $\mathtt{trans}(P, \Gamma)$ to the translation of the Spyder post-condition $\exists \mathsf{v}' . \mathrm{weaken\_foreach}(\mathrm{P}, \mathrm{v}, \Gamma)[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}'] \wedge \mathsf{val}(\mathsf{v}) = E[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}']$.

   Consider the translation of $\mathsf{val}(\mathsf{v})$ in the context of Γ. Since Γ is well-formed with respect to the Put to v, it must be the case that $\mathsf{v} \in \Gamma$ and $\Gamma(\mathsf{v}) = \mathbf{y}$ for some variable $\mathbf{y}$. Furthermore, the Spyder expressions $\mathsf{val}(\mathsf{v})$ and $\mathsf{iter}(\mathsf{v})$ are translated to $\mathbf{y}[\mathbf{v}]$ and $\mathbf{v}$ respectively.

Next, consider the Hoare postcondition of the translated put statement. The *Put* statement is translated to $\mathbf{y}[\mathbf{v}] := \mathtt{trans}(E, \Gamma)$, and we can apply the `Assign-Array` rule to obtain the postcondition of $\mathtt{trans}(P, \Gamma)$:

$$\{\mathtt{trans}(P, \Gamma)\}\, \mathbf{y}[\mathbf{v}] := \mathtt{trans}(E, \Gamma)\, \{ \\ \exists \mathbf{y}' . \mathtt{trans}(P, \Gamma)[\mathbf{y} \mapsto \mathbf{y}'] \wedge \mathbf{y} = \mathbf{y}'[\mathbf{v} := \mathtt{trans}(E, \Gamma)[\mathbf{y} \mapsto \mathbf{y}']]\},$$

where $\mathbf{y}'$ is some fresh variable.

Because the case is `Put`, we have just derived the Spyder triple

$$\langle P \rangle\ \mathsf{v} \leftarrow E\ \langle \exists \mathsf{v}' . P[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}'] \wedge \mathsf{val}(\mathsf{v}) = E[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}'] \rangle,$$

where v' is some free variable.

Let σ be a ImpArray state such that

$$[\![\exists \mathbf{y}' . \mathtt{trans}(P, \Gamma)[\mathbf{y} \mapsto \mathbf{y}'] \wedge \mathbf{y} = \mathbf{y}'[\mathbf{v} := \mathtt{trans}(E, \Gamma)[\mathbf{y} \mapsto \mathbf{y}']]]\!]_\sigma = t$$

.

Consider the Hoare term P', $\exists \mathsf{v}' . \mathtt{trans}(P[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}'] \wedge \mathsf{val}(\mathsf{v}) = E[\mathsf{val}(\mathsf{v}) \mapsto \mathsf{v}'], \Gamma)$, or equivalently,

$$\exists \mathbf{v}' . \mathtt{trans}(P, \Gamma)[\mathbf{y}[\mathbf{v}] \mapsto \mathbf{v}'] \wedge \mathbf{y}[\mathbf{v}] = \mathtt{trans}(E, \Gamma)[\mathbf{y}[\mathbf{v}] \mapsto \mathbf{v}'].$$

We claim that $[\![P']\!]_\sigma = t$. Since P is well-formed with respect to Γ, and $\Gamma(\mathsf{x}) = \mathsf{y}$, it must be the case that the substitution of $\mathbf{y} \mapsto \mathbf{y}'$ only affects translations of $\mathsf{val}(\mathsf{v})$. As a result, if

$\mathbf{y}'$ is an (array) witness for

$$[\![\exists \mathbf{y}'.\,\mathtt{trans}(P,\,\Gamma)[\mathbf{y}\mapsto\mathbf{y}'] \wedge \mathbf{y} = \mathbf{y}'[\mathbf{v}{:=}\mathtt{trans}(E,\,\Gamma)[\mathbf{y}\mapsto\mathbf{y}']]]\!]_\sigma,$$

we can use the value $\mathbf{y}[\mathbf{v}]$ as a (variable) witness for P'.

Since $[\![P']\!]_\sigma = t$, we can apply `Consequence` to obtain the triple

$$\{\mathtt{trans}(P,\,\Gamma)\}\mathtt{trans}(\mathsf{v} \leftarrow E,\,\Gamma)\,\{P'\}.$$

7. Inductive case, in which the last step of the derivation is `For`: $\langle P\rangle$ for $(\mathsf{x},\mathsf{y})B_i\ \langle P\rangle$, where P is of the form $\mathsf{foreach}(\mathsf{x},\mathsf{y})P_i$. S

   At a high-level, this rule is introducing a quantification over the elements of y. This is sound because the body B_i can only adjust the elements at the current iteration, because the loop cannot modify variables captured in I, and because the translated loop is guaranteed to execute exactly once for every element of y.

   Let Γ' be Γ extended with the loop binding $\mathsf{x}\mapsto\mathbf{y}$. Since Γ is well-formed with respect to the loop, it must be the case that Γ' is well-formed as well. Recall that the translated loop is

$$\mathbf{x}{:=}0;\,\mathsf{while}\,(\mathbf{x} < \mathsf{size}(\mathbf{y}))\,\mathtt{trans}(B_i,\,\Gamma');\,\mathbf{x}{:=}\mathbf{x}+1.$$

   Consider the translated foreach predicate I

$$\forall \mathbf{x}'.\,0 \le \mathbf{x}' < \mathsf{size}(\mathbf{y}) \implies \mathtt{trans}(P_i,\,\Gamma')[\mathbf{x}\mapsto\mathbf{x}'].$$

   We will use the `While` rule with three helper predicates: intuitively, we keep three predicates around to 1) quantify I for previous iterations 2) safely weaken I for the current iteration and 3) quantify I for future iterations. Let I_{pre} restrict I up to the current iteration,

$$\forall \mathbf{x}'.\, 0 \le \mathbf{x}' < \mathbf{x} \implies \texttt{trans}(P_i,\, \Gamma')[\mathbf{x} \mapsto \mathbf{x}'].$$

Let I_{post} weaken I using $\mathrm{weaken\_prev(P_i)}$ for future iterations:

$$\forall \mathbf{x}'.\, \mathbf{x} < \mathbf{x}' < \mathsf{size}(\mathbf{y}) \implies \texttt{trans}(\mathrm{weaken\_prev(P_i)},\, \Gamma')[\mathbf{x} \mapsto \mathbf{x}'].$$

Finally, let I_{curr} be the weakening of I for the current iteration: $\texttt{trans}(\mathrm{weaken\_prev(P_i)},\, \Gamma')$.

We will use the `While` rule with the combined predicate $I_{pre} \land I_{post} \land I_{curr}$ as the loop invariant, and in particular, we will show the following Hoare triple holds:

$$\{I_{pre} \land I_{post} \land I_{curr} \land 0 \le \mathbf{x} < \mathsf{size}(\mathbf{y})\}\, \texttt{trans}(B_i,\, \Gamma');\, \mathbf{x} := \mathbf{x} + 1 \{I_{pre} \land I_{post} \land I_{curr}\}.$$

From the inductive hypothesis we have the triple

$$\{I_{curr} \land 0 \le \mathbf{x} < \mathsf{size}(\mathbf{y})\}\, \texttt{trans}(B_i,\, \Gamma')\, \{\texttt{trans}(P_i,\, \Gamma')\}.$$

Since $I_{pre} \land I_{post} \land I_{curr} \implies I_{curr}$, we apply `Consequence` on the precondition to obtain

$$\{I_{pre} \land I_{post} \land I_{curr} \land 0 \le \mathbf{x} < \mathsf{size}(\mathbf{y})\}\, \texttt{trans}(B_i,\, \Gamma')\, \{\texttt{trans}(P_i,\, \Gamma')\}.$$

From Lemma 4.6.2, since the loop with B_i is well-formed with respect to Γ', it must be the case that $\mathbf{x}$ is not assigned within B_i. As well, since B_i is restricted from writing to free variables of B_i, the only way for I_{pre} and I_{post} to be invalidated by $\texttt{trans}(B_i,\, \Gamma')$ is through a tt Put. Since Spyder does not have aliasing, each `Put` within B_i with x as a target only writes to the current iteration (i.e. each `Put` only invalidates I_{curr}). Furthermore, since B_i does not have nested loops over $\mathtt{y}$, x is the only possible target to write to $\mathtt{y}$, and so it must be the case that I_{pre} and I_{post} are not invalidated by $\mathtt{trans}(B_i,\, \Gamma')$.

As a result, we can safely strengthen the postcondition of this triple with I_{pre} and I_{post}:

$$\{I_{pre} \wedge I_{post} \wedge I_{curr} \wedge 0 \leq \mathbf{x} < \mathsf{size}(\mathbf{y})\}\, \mathtt{trans}(B_i,\, \Gamma')\, \{I_{pre} \wedge I_{post} \wedge \mathtt{trans}(P_i,\, \Gamma')\}.$$

Finally, consider the increment of $\mathbf{x}$ after the loop. Given the precondition $I_{pre} \wedge I_{post} \wedge \mathtt{trans}(P_i,\, \Gamma')$, we apply the `Assign` rule to obtain

$$\{I_{pre} \wedge I_{post} \wedge \mathtt{trans}(P_i,\, \Gamma')\}\, \mathbf{x} := \mathbf{x} + 1\, \{\exists\, \mathbf{v}\,.\, (I_{pre} \wedge I_{post} \wedge \mathtt{trans}(P_i,\, \Gamma'))[\mathbf{x} \mapsto \mathbf{v}] \wedge \mathbf{x} = \mathbf{v} + 1\},$$

where $\mathbf{v}$ is a fresh variable. This postcondition is logically equivalent to $I_{pre} \wedge I_{post} \wedge I_{curr}$, and so we apply `Consequence` and `Sequence` to obtain

$$\{I_{pre} \wedge I_{post} \wedge I_{curr} \wedge 0 \leq \mathbf{x} < \mathsf{size}(\mathbf{y})\}\, \mathtt{trans}(B_i,\, \Gamma')\, ;\, \mathbf{x} := \mathbf{x} + 1 \{I_{pre} \wedge I_{post} \wedge I_{curr}\}.$$

Finally, we apply `While` with the condition $0 \leq \mathbf{x} < \mathsf{size}(\mathbf{y})$ to obtain the triple

$$\{I_{pre} \wedge I_{post} \wedge I_{curr}\}\, \mathsf{while}\, (0 \leq \mathbf{x} < \mathsf{size}(\mathbf{y}))\, \mathtt{trans}(B_i,\, \Gamma')\, \{I_{pre} \wedge I_{post} \wedge I_{curr} \wedge \mathbf{x} = \mathsf{size}(\mathbf{y})\}.$$

From here, it remains to use `Consequence`, and `Sequence` to build a triple for the loop initialization.

□

4.6.4 Proofs: Soundness of Targeted Synthesis

Lemma 4.6.5 (Block Append)**.**

$$\forall B, B', P, Q, R.\, \langle P\rangle\, B\, \langle Q\rangle \wedge \langle Q\rangle\, B'\, \langle R\rangle \quad \Longrightarrow \langle P\rangle\, B \mathbin{+\!\!+} B'\, \langle R\rangle.$$

Proof. By structural induction over the arguments of $+\!\!+$. □

Theorem 4.6.6.

$$\forall \Pi, \Phi, B, B'.\, \mathtt{cn}\,;\mathtt{md} \vdash \langle \Pi\rangle\, B\, \langle \Phi\rangle \hookrightarrow B' \quad \Longrightarrow \langle \Pi\rangle\, B'\, \langle \Phi\rangle$$

Proof. Induction over the derivation of $\mathtt{cn}\,;\mathtt{md} \vdash \langle \Pi\rangle\, B\, \langle \Phi\rangle \hookrightarrow B'$. In all cases we show that $\langle \Pi\rangle\, B'\, \langle \Phi\rangle$.

1. Base case, in which the last step is `Synth-Base`: $\mathtt{cn}\,;\,\mathtt{md} \vdash \langle \Pi\rangle\ \mathtt{skip}\ \langle \Phi\rangle \hookrightarrow B$ Because a side-condition for `Synth-Base` is $\langle \Pi\rangle\, B\, \langle \Phi\rangle$, this is trivially true.

2. Base case, in which the last step is `Synth-Loop`: $\mathtt{cn}\,;\,\mathtt{md} \vdash \langle \Pi\rangle\ \mathtt{skip}\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\phi \wedge \Phi\rangle \hookrightarrow B$. This is true from the inductive hypothesis.

3. Recursive case, in which the last step is `Consequence`: $\mathtt{cn}\,;\,\mathtt{md} \vdash \langle \Pi\rangle\ B\ \langle \Phi\rangle \hookrightarrow B' \mathbin{+\!\!+} B''$. From Lemma 4.6.5 and the inductive hypothesis, it is the case that $\langle \Pi\rangle\, B' \mathbin{+\!\!+} B''\, \langle \Phi\rangle$.

4. Recursive case, in which the last step is `Assign`: $\mathtt{cn}\,;\,\mathtt{md} \vdash \langle \Pi\rangle\ \mathsf{v} := E\ ;B\ \langle \Phi\rangle \hookrightarrow \mathsf{v} := E\ ;B'$. We apply the Hoare rule for `Assign` to the inductive hypothesis.

5. Recursive case, in which the last step is `Put`: $\texttt{cn}\,;\texttt{md} \vdash \langle\Pi\rangle\, \mathsf{v} \leftarrow E\,; B\, \langle\Phi\rangle \hookrightarrow \mathsf{v} \leftarrow E\,; B'$. This is analogous to `Assign`.

6. Recursive case, in which the last step is one of the Extension rules. These are all trivially sound from the inductive hypothesis.

7. Recursive case, in which the last step is `Foreach-Specialize`:

$$\{\} \vdash \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle\ \mathsf{for}\ \overline{(\mathsf{x}_i, \mathsf{y}_i)} B_i\,; B\, \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle \hookrightarrow$$
$$\mathsf{for}\overline{(\mathsf{x}_i, \mathsf{y}_i)}(B_{pre} \mathbin{+\!\!+} B'_i)\ ;\ B'.$$

In this case, we use the inductive hypothesis to establish the triple for B'_i. Next, we use the inductive hypothesis and Lemma 4.6.5 to establish the triple for B'_i and B_{pre}:

$$\langle \text{weaken\_prev}(\Phi)\rangle\, (B_{pre} \mathbin{+\!\!+} B'_i)\, \langle\Phi\rangle.$$

On this, we apply the `For` Hoare logic rule to introduce the foreach term, and we appeal to the inductive hypothesis for the remainder B'.

8. Recursive case, in which the last step is `Conditional`: $\{\} \vdash \langle\Phi\rangle\ \mathsf{if}\ E\ \mathsf{then}\ B_t\ \mathsf{else}\ B_f\ ; B\ \langle\Phi\rangle \hookrightarrow \mathsf{if}\ E\ \mathsf{then}\ B'_t\ \mathsf{else}\ B'_f\ ; B'$. This follows from the inductive hypothesis and the `Conditional` Hoare rule.

□

4.6.5 Proofs: Soundness of Targeted Synthesis

Theorem 4.6.7.

$$\forall \Pi, \Phi, B, B'.\,\texttt{cn}\,;\texttt{md} \vdash \langle\Pi\rangle\, B\, \langle\Phi\rangle \hookrightarrow B' \quad \Longrightarrow \langle\Pi\rangle\, B'\, \langle\Phi\rangle$$

Proof. Induction over the derivation of $\texttt{cn}; \texttt{md} \vdash \langle\Pi\rangle\, B\, \langle\Phi\rangle \hookrightarrow B'$. In all cases we show that $\langle\Pi\rangle\, B'\, \langle\Phi\rangle$.

1. Base case, in which the last step is `Synth-Base`: $\texttt{cn} ; \texttt{md} \vdash \langle\Pi\rangle\ \texttt{skip}\ \langle\Phi\rangle \hookrightarrow B$ Because a side-condition for `Synth-Base` is $\langle\Pi\rangle\, B\, \langle\Phi\rangle$, this is trivially true.

2. Base case, in which the last step is `Synth-Loop`: $\texttt{cn} ; \texttt{md} \vdash \langle\Pi\rangle\ \texttt{skip}\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\phi \wedge \Phi\rangle \hookrightarrow B$. This is true from the inductive hypothesis.

3. Recursive case, in which the last step is `Consequence`: $\texttt{cn} ; \texttt{md} \vdash \langle\Pi\rangle\ B\ \langle\Phi\rangle \hookrightarrow B' +\!\!+ B''$. From Lemma 4.6.5 and the inductive hypothesis, it is the case that $\langle\Pi\rangle\, B' +\!\!+ B''\, \langle\Phi\rangle$.

4. Recursive case, in which the last step is `Assign`: $\texttt{cn} ; \texttt{md} \vdash \langle\Pi\rangle\ \mathsf{v} := E\ ; B\ \langle\Phi\rangle \hookrightarrow \mathsf{v} := E\ ; B'$. We apply the Hoare rule for `Assign` to the inductive hypothesis.

5. Recursive case, in which the last step is `Put`: $\texttt{cn} ; \texttt{md} \vdash \langle\Pi\rangle\ \mathsf{v} \leftarrow E\ ; B\ \langle\Phi\rangle \hookrightarrow \mathsf{v} \leftarrow E\ ; B'$. This is analogous to `Assign`.

6. Recursive case, in which the last step is one of the Extension rules. These are all trivially sound from the inductive hypothesis.

7. Recursive case, in which the last step is `Foreach-Specialize`:

$$\{\} \vdash \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle\ \mathsf{for}\ \overline{(\mathsf{x}_i, \mathsf{y}_i)} B_i\ ; B\ \langle \mathsf{foreach}\overline{(\mathsf{v}_i, \mathsf{u}_i)}\ \phi \wedge \Phi\rangle \hookrightarrow \\ \mathsf{for}\ \overline{(\mathsf{x}_i, \mathsf{y}_i)}(B_{pre} +\!\!+ B'_i)\ ;\ B'.$$

   In this case, we use the inductive hypothesis to establish the triple for B'_i. Next, we use the inductive hypothesis and Lemma 4.6.5 to establish the triple for B'_i and B_{pre}:

$$\langle \text{weaken\_prev}(\Phi)\rangle\, (B_{pre} +\!\!+ B'_i)\, \langle\Phi\rangle.$$

On this, we apply the `For` Hoare logic rule to introduce the foreach term, and we appeal to the inductive hypothesis for the remainder B'.

8. Recursive case, in which the last step is `Conditional`: $\{\} \vdash \langle\Phi\rangle$ $\mathsf{if}\ E\ \mathsf{then}\ B_t\ \mathsf{else}\ B_f\ ; B\ \langle\Phi\rangle \hookrightarrow \mathsf{if}\ E\ \mathsf{then}\ B'_t\ \mathsf{else}\ B'_f\ ; B'$. This follows from the inductive hypothesis and the `Conditional` Hoare rule.

□

Acknowledgements

This chapter in part is currently being prepared for submission for publication of the material. Sarracino, John; Barke, Shraddha; Peleg, Hila; Lerner, Sorin; Polikarpova, Nadia. The dissertation author was a primary investigator and author of this material.

Chapter 5

Conclusion and Future Work

The common insight in all of the projects presented in this thesis is to automate programming in a domain in which people already understand how to write programs, but where programming is difficult, important, or needed by people without programming knowledge. For the future I will further explore the ideas in this thesis, as well as develop this style of research in other areas.

Two ripe directions for future work are to flesh out and expand the applicability of MOCKDOWN for visual layouts, and to explore the area of *parsers*.

Visual Layouts. While MOCKDOWN has great potential for easing the burden of authoring visual layouts, there are several large barriers to broader applicability. There are many layout systems present in the wild, and it's unclear how to integrate MOCKDOWN with an existing layout system. In addition, there are plenty of common and useful layouts that do not fit the formal machinery of MOCKDOWN[1]. Finally, the algorithms used by MOCKDOWN are not robust to user-input noise (i.e. small variations between input examples). In the future I will work on techniques for addressing all of these issues.

Parsers. The problem of *parsing* is a rich, storied research area with many different tools and techniques [46]. Many programmers frequently use low-level, left-to-right parsing technique due to their speed, robust error messages, and intuitive semantics [68]; such parsers are ubiquitious.

[1]Several difficult-to-formalize techniques are conditional formats, text-wrapping, and layouts for which the underlying data is dynamic (e.g. a social media feed).

Unfortunately though left-to-right parsers are difficult to verify correct and are prone to implementation errors. Due to the broad use of these parsers, their implementations are ripe targets for security exploits[2] and indeed a recent push in the security community has been to fix so-called *shotgun parsers* [89].

In the future, I will develop techniques for taking a high-level parser specification and automatically generating a correct-by-construction low-level implementation of the parser.

[2]For example, the recent Heartbleed bug was a result of a left-to-right parser error [21].

Bibliography

[1] Apparatus: a hybrid graphics editor and programming environment for creating interactive diagrams. http://aprt.us. Accessed: 2016-04-12.

[2] Fabricjs javascript canvas library. http://fabricjs.com/. Accessed: 2016-09-19.

[3] Interactive mathematics: Learn math while you play with it. http://www.intmath.com. Accessed: 2016-04-08.

[4] John Conway's Game of Life. https://bitstorm.org/gameoflife/. Accessed: 2018-07-12.

[5] Overconstrained: Cassowary projects and its community. http://overconstrained.io. Accessed: 2016-04-11.

[6] Pendulum lab – motion, pendulum, simple harmonic motion – phet. https://phet.colorado.edu/en/simulation/legacy/pendulum-lab. Accessed: 2016-09-19.

[7] Phet: Interactive simulations for science and math. https://PhET.colorado.edu. Accessed: 2016-04-11.

[8] Resonance – resonance, harmonic motion, oscillator – phet. https://phet.colorado.edu/en/simulation/legacy/resonance. Accessed: 2016-09-19.

[9] Wendy K. Adams, Sam Reid, Ron LeMaster, Sarah McKagan, Katherine Perkins, Michael Dubson, and Carl E. Wieman. A study of educational simulations part ii – interface design. *Journal of Interactive Learning Research*, 19(4):551–577, October 2008.

[10] Wendy K. Adams, Sam Reid, Ron LeMaster, Sarah B. McKagan, Katherine K. Perkins, Michael Dubson, and Carl E. Wieman. A study of educational simulations part i - engagement and learning. *Journal of Interactive Learning Research*, 19(3):397–419, July 2008.

[11] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV (Part II)*, volume 9780 of *LNCS*, pages 934–950. Springer, 2013.

[12] Christine Alvarado and Randall Davis. Sketchread: A multi-domain sketch recognition engine. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, pages 23–32, New York, NY, USA, 2004. ACM.

[13] Christine Alvarado and Randall Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *ACM SIGGRAPH 2006 Courses*, page 24. ACM, 2006.

[14] Greg J Badros, Alan Borning, and Peter J Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[15] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306, dec 2001.

[16] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1327–1342, 2015.

[17] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[18] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3, 2004.

[19] Pavol Bielik, Marc Fischer, and Martin Vechev. Robust relational layout synthesis from examples for android. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[20] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *LISP and symbolic computation*, 5(3):223–270, 1992.

[21] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. Heartbleed 101. *IEEE security & privacy*, 12(4):63–67, 2014.

[22] Julia M. Chamberlain, Kelly Lancaster, Robert Parson, and Katherine K. Perkins. How guidance affects student engagement with an interactive simulation. *Chem. Educ. Res. Pract.*, 15:628–638, 2014.

[23] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - providing ocaml with a formal specification language. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, pages 484–501. Springer, 2019.

[24] Salman Cheema and Joseph J LaViola Jr. Applying mathematical sketching to sketch-based physics tutoring software. In *International Symposium on Smart Graphics*, pages 13–24. Springer, 2010.

[25] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices*, 51(6):341–354, 2016.

[26] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[27] John Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[28] Gregory H Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308. Springer, 2006.

[29] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration*. MIT press, 1993.

[30] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[31] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[32] Richard C. Davis, Brien Colwell, and James A. Landay. K-sketch. In *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*, page 413, New York, New York, USA, apr 2008. ACM Press.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[34] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*, pages 689–700. ACM, 2015.

[35] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[36] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, chapter Babelsberg/JS, pages 411–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[37] Tim Felgentreff, Todd Millstein, Alan Borning, and Robert Hirschfeld. Checks and balances: constraint solving without surprises in object-constraint programming languages. In *ACM SIGPLAN Notices*, volume 50, pages 767–782. ACM, 2015.

[38] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, pages 422–436. ACM, 2017.

[39] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[40] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[41] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas W. Reps, Phillip A. Porras, Hassen Saïdi, and Vinod Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *CAV*, 2012.

[42] Bjorn N Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[43] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 66:1–66:9, New York, NY, USA, 2009. ACM.

[44] Martin Grabmüller and Petra Hofstedt. Turtle: A constraint imperative programming language. In *Research and Development in Intelligent Systems XX*, pages 185–198. Springer, 2004.

[45] Dawn Griffiths and David Griffiths. *Head first Android development: A brain-friendly guide*. " O'Reilly Media, Inc.", 2017.

[46] Dick Grune and Ceriel JH Jacobs. Parsing techniques. *Monographs in Computer Science. Springer,*, page 13, 2007.

[47] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[48] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, August 2012.

[49] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[50] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, page 62, New York, New York, USA, jun 2011. ACM Press.

[51] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. *ACM SIGPLAN Notices*, 46(6):50, jun 2011.

[52] CAR Hoare and II Chapter. Notes on data structuring, structured programming, 1972.

[53] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[54] Markus Hohenwarter and Karl Fuchs. Combination of dynamic geometry , algebra and calculus in the software system GeoGebra. *Computer algebra systems and dynamic geometry systems in mathematics teaching conference 2004*, 2002(July):1–6, 2005.

[55] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*, pages 231–241, New York, New York, USA, oct 2014. ACM Press.

[56] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In *Acm siggraph 2007 courses*, page 21. ACM, 2007.

[57] Joxan Jaffar, Spiro Michaylov, Peter J Stuckey, and Roland HC Yap. The clp (r) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[58] Joaquim Jorge and Faramarz Samavati. *Sketch-based interfaces and modeling*. Springer Science & Business Media, 2010.

[59] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 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, 2011. ACM.

[60] Solange Karsenty, Chris Weikart, and James A Landay. Inferring graphical constraints with rockit. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*, page 531, 1993.

[61] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[62] Alan Kay and Adele Goldberg. Personal dynamic media. *Computer*, 10(3):31–41, 1977.

[63] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. Kitty: sketching dynamic and interactive illustrations. *Proceedings of the 27th annual ACM symposium on User interface software and technology - UIST '14*, pages 395–405, 2014.

[64] Rubaiat Habib Kazi, Fanny Chevalier, Tovi Grossman, Shengdong Zhao, and George Fitzmaurice. Draco: Bringing Life to Illustrations with Kinetic Textures. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 351–360, 2014.

[65] Rubaiat Habib Kazi, Tovi Grossman, Nobuyuki Umetani, and George Fitzmaurice. Skuid: Sketching dynamic illustrations using the principles of 2d animation. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4599–4609. ACM, 2016.

[66] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *CAV*, 2015.

[67] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, pages 407–426. ACM, 2013.

[68] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[69] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[70] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services - MobiSys '13*, page 193, New York, New York, USA, jun 2013. ACM Press.

[71] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[72] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[73] K Rustan M Leino and Aleksandar Milicevic. Program extrapolation with jennisys. In *ACM SIGPLAN Notices*, volume 47, pages 411–430. ACM, 2012.

[74] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

[75] K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004.

[76] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, pages 80–94, 2007.

[77] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728. ACM, 2008.

[78] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[79] Hakon Wium Lie and Bert Bos. *Cascading style sheets: Designing for the web, Portable Documents*. Addison-Wesley Professional, 2005.

[80] Mark H Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 160–175. Springer, 2013.

[81] Mark H Liffiton and Karem A Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[82] Hao Lü and Yang Li. Gesture coder: A tool for programming multi-touch gestures by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2875–2884, New York, NY, USA, 2012. ACM.

[83] Ethan Marcotte. *Responsive web design: A book apart n 4*. Editions Eyrolles, 2017.

[84] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[85] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[86] Micha Meier. Debugging constraint programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 204–221. Springer, 1995.

[87] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[88] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.

[89] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 45–52, 2016.

[90] Emily B. Moore, Timothy A. Herzog, and Katherine K. Perkins. Interactive simulations as implicit support for guided-inquiry. *Chem. Educ. Res. Pract.*, 14:257–268, 2013.

[91] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[92] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *ICLR*, 2018. To appear.

[93] B. A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 59–66, New York, NY, USA, 1986. ACM.

[94] Brad A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.*, 12(2):143–177, April 1990.

[95] Dan R. Olsen, Jr. and Kirk Allan. Creating interactive techniques by symbolically solving geometric constraints. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '90, pages 102–107, New York, NY, USA, 1990. ACM.

[96] Stephen Oney, Brad Myers, and Joel Brandt. Constraintjs: programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 229–238. ACM, 2012.

[97] Eleanor O'Rourke, Erik Andersen, Sumit Gulwani, and Zoran Popović. A framework for automatically generating interactive instructional scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1545–1554, New York, NY, USA, 2015. ACM.

[98] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630. ACM, 2015.

[99] Edgar Pek, Xiaokang Qiu, and Parthasarathy Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *ACM SIGPLAN Notices*, volume 49, pages 440–451. ACM, 2014.

[100] Katherine Perkins, Wendy Adams, Michael Dubson, Noah Finkelstein, Sam Reid, Carl Wieman, and Ron LeMaster. PhET: Interactive Simulations for Teaching and Learning Physics. *The Physics Teacher*, 44(1):18, dec 2006.

[101] Noah S. Podolefsky, Katherine K. Perkins, and Wendy K. Adams. Computer simulations to classrooms: tools for change. *AIP Conference Proceedings*, 1179(1):233–236, 2009.

[102] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016.

[103] NADIA POLIKARPOVA, DEIAN STEFAN, JEAN YANG, SHACHAR ITZHAKY, TRAVIS HANCE, and ARMANDO SOLAR-LEZAMA. Liquid information flow control.

[104] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *FM*, pages 514–530, 2014.

[105] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *OOPSLA*, pages 107–126. ACM, 2015.

[106] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[107] Robert D Rogers and Stephen Monsell. Costs of a predictible switch between simple cognitive tasks. *Journal of experimental psychology: General*, 124(2):207, 1995.

[108] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[109] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 661–676, 2014.

[110] Erica Sadun. *iOS Auto Layout Demystified*. Addison-Wesley Professional, 2013.

[111] Christian Schulte. Oz explorer: A visual constraint programming tool. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 477–478. Springer, 1996.

[112] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *PLDI*, pages 326–340. ACM, 2016.

[113] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[114] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[115] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, October 2006.

[116] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326. ACM, 2010.

[117] CACM Staff. React: Facebook's functional turn on writing javascript. *Communications of the ACM*, 59(12):56–62, 2016.

[118] Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In *VMCAI*, pages 328–344, 2010.

[119] Ivan E Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[120] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014.

[121] Brad Vander Zanden. Languages for developing user interfaces. chapter An Active-value&Mdash;Spreadsheet Model for Interactive Languages, pages 183–209. A. K. Peters, Ltd., Natick, MA, USA, 1992.

[122] Bradley T. Vander Zanden, Richard Halterman, Brad A. Myers, Rob Miller, Pedro Szekely, Dario A. Giuse, David Kosbie, and Rich McDaniel. Lessons learned from programmers' experiences with one-way constraints: Research articles. *Softw. Pract. Exper.*, 35(13):1275–1298, November 2005.

[123] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Acm sigplan notices*, volume 35, pages 242–252. ACM, 2000.

[124] Wikipedia contributors. Alexa internet — Wikipedia, the free encyclopedia, 2020. [Online; accessed 28-June-2020].

[125] Wikipedia contributors. Baidu — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Baidu&oldid=964490513, 2020. [Online; accessed 28-June-2020].

[126] Wikipedia contributors. Duckduckgo — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DuckDuckGo&oldid=964046557, 2020. [Online; accessed 28-June-2020].

[127] Wikipedia contributors. Facebook — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Facebook&oldid=964800204, 2020. [Online; accessed 28-June-2020].

[128] Wikipedia contributors. Google search — Wikipedia, the free encyclopedia, 2020. [Online; accessed 28-June-2020].

[129] Wikipedia contributors. Qihoo 360 — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Qihoo_360&oldid=961339374, 2020. [Online; accessed 28-June-2020].

[130] Wikipedia contributors. Sohu — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sohu&oldid=950549005, 2020. [Online; accessed 28-June-2020].

[131] Wikipedia contributors. Taobao — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Taobao&oldid=956683522, 2020. [Online; accessed 28-June-2020].

[132] Wikipedia contributors. Tencent qq — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tencent_QQ&oldid=964322805, 2020. [Online; accessed 28-June-2020].

[133] Wikipedia contributors. Tmall — Wikipedia, the free encyclopedia, 2020. [Online; accessed 28-June-2020].

[134] Wikipedia contributors. Visual studio code — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=964424545, 2020. [Online; accessed 28-June-2020].

[135] Wikipedia contributors. Yahoo! — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Yahoo!&oldid=963547691, 2020. [Online; accessed 28-June-2020].

[136] Wikipedia contributors. Youtube — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=YouTube&oldid=964752628, 2020. [Online; accessed 28-June-2020].

[137] Haijun Xia, Bruno Araujo, Tovi Grossman, and Daniel Wigdor. Object-oriented drawing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4610–4621. ACM, 2016.

[138] Jun Xing, Rubaiat Habib Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. Energy-brushes: Interactive tools for illustrating stylized elemental dynamics. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 755–766. ACM, 2016.

[139] Jun Xing, Li-Yi Wei, Takaaki Shiratori, and Koji Yatani. Autocomplete hand-drawn animations. *ACM Transactions on Graphics (TOG)*, 34(6):169, 2015.

[140] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[141] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 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, 2013. ACM.

[142] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The auckland layout editor: An improved gui layout specification process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, NY, USA, 2013. ACM.

[143] Bo Zhu, Michiaki Iwata, Ryo Haraguchi, Takashi Ashihara, Nobuyuki Umetani, Takeo Igarashi, and Kazuo Nakazawa. Sketch-based Dynamic Illustration of Fluid Systems. *ACM Transactions on Graphics*, 30(6):1, dec 2011.