2018

The Three Pillars of Machine Programming

Justin E. Gottschlich

Armando Solar-Lezama

Nesime Tatbul

Michael Carbin

Martin Rinard

See next page for additional authors

The Three Pillars of Machine Programming

Abstract

In this position paper, we describe our vision of the future of machine programming through a categorical examination of three pillars of research. Those pillars are:(i) intention,(ii) invention, and (iii) adaptation. Intention emphasizes advancements in the human-to-computer and computer-to-machine-learning interfaces. Invention emphasizes the creation or refinement of algorithms or core hardware and software building blocks through machine learning (ML). Adaptation emphasizes advances in the use of ML-based constructs to autonomously evolve software.

Keywords

program synthesis, machine programming, software development, software maintenance, intention, invention, adaptation

Author(s)

Justin E. Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Timothy Mattson

The Three Pillars of Machine Programming

Justin Gottschlich
Intel Labs, USA
justin.gottschlich@intel.com

Armando Solar-Lezama
MIT, USA
asolar@csail.mit.edu

Nesime Tatbul
Intel Labs and MIT, USA
tatbul@csail.mit.edu

Michael Carbin
MIT, USA
mcarbin@csail.mit.edu

Martin Rinard
MIT, USA
rinard@csail.mit.edu

Regina Barzilay
MIT, USA
regina@csail.mit.edu

Saman Amarasinghe
MIT, USA
saman@csail.mit.edu

Joshua B. Tenenbaum
MIT, USA
jbt@mit.edu

Tim Mattson
Intel Labs, USA
timothy.g.mattson@intel.com

Abstract

In this position paper, we describe our vision of the future of machine programming through a categorical examination of three pillars of research. Those pillars are: *(i)* intention, *(ii)* invention, and *(iii)* adaptation. Intention emphasizes advancements in the human-to-computer and computer-to-machine-learning interfaces. Invention emphasizes the creation or refinement of algorithms or core hardware and software building blocks through machine learning (ML). Adaptation emphasizes advances in the use of ML-based constructs to autonomously evolve software.

Keywords program synthesis, machine programming, software development, software maintenance, intention, invention, adaptation

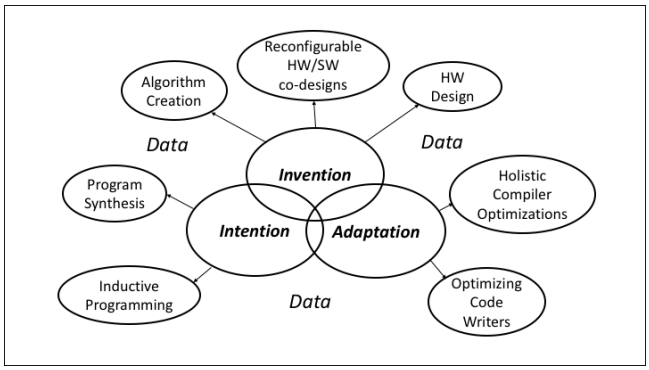

Figure 1. The Three Pillars of Machine Programming: Intention, Invention, and Adaptation. Each pillar in the diagram includes a few example sub-domains generally related to them.

1 Introduction

Programming is a cognitively demanding task that requires extensive knowledge, experience and a large degree of creativity, and is notoriously difficult to automate. Machine learning (ML) has the capacity to reshape the way software is developed. At some level, this has already begun, as machine-learned components progressively replace complex hand-crafted algorithms in domains such as natural-language understanding and vision. Yet, we believe that it is possible to move much further. We envision machine learning and automated reasoning techniques that will enable new programming systems; systems that will deliver a significant degree of automation to reduce the cost of producing secure, correct, and efficient software. These systems will also enable non-programmers to harness the full power of modern computing platforms to solve complex problems correctly and efficiently. We call such efforts *machine programming*.

1.1 Why Now?

Programming is the process of turning a problem definition (the *intent*) into a sequence of instructions that when executed on a computer, produces a solution to the original problem. Over time, a program must be *maintained* as it adapts to changes in the program's goals, errors in the program, and the features in new computer platforms. A *machine programming system* is any system that automates some or all of the steps of turning the user's intent into an executable program and maintaining that program over time.

The automation of programming has been a goal of the programming systems community since the birth of Fortran in the 1950s. The first paper on *"The FORTRAN Automatic Coding System"* made it clear that its goal was "for the 704 [IBM's next large computer] to code problems for itself and produce as good programs as human coders (but without the errors)" [8]. The broader AI community has also been interested in automatic programming dating back to the Programmer's Apprentice Project back in the late 1970s [64]. A number of technological developments over the past few years, however, are creating both the *need* and the *opportunity* for transformative advances in our ability to use machines to help users write software programs.

Opportunity Humans interact through speech, images, and gestures; so-called "natural inputs". Advances in deep

learning and related machine learning technologies have dramatically improved a computer's ability to associate meaning with natural inputs. Deep learning also makes it possible to efficiently represent complex distributions over classes of structured objects; a crucial capability if one wants to automatically synthesize a program using probabilistic or direct transformation techniques. In parallel to advances in machine learning, the programming systems community has been making notable advances in its ability to reason about programs and manipulate them. Analyzing thousands of lines of code to derive inputs that expose a bug has moved from an intractable problem into one that is routinely solved due to advances in automated reasoning tools such as SAT and SMT solvers. Data, a key enabler for learning-based strategies, is also more available now than at any time in the past. This is the byproduct of at least two factors: *(i)* the emergence of code repositories, such as GitHub, and *(ii)* the growing magnitude of the web itself, where it is possible to observe and analyze the code (e.g., JavaScript) powering many web applications. Finally, the advent of cloud computing makes it possible to harness large-scale computational resources to solve complex analysis and inference problems that were out of reach only a few years ago.

Need The end of Dennard scaling means that performance improvements now come through increases in the complexity of the hardware, with resulting increases in the complexity of compilation targets [27]. Traditional compilation techniques rely on an accurate model of relatively simple hardware. These techniques are inadequate for exploiting the full potential of heterogeneous hardware platforms. The time is ripe for techniques, based on modern machine learning, that learn to map computations onto multiple platforms for a single application. Such techniques hold the promise of effectively working in the presence of the multiple sources of uncertainty that complicate the use of traditional compiler approaches. Moreover, there is a growing need for people with core expertise outside of computer science to program, whether for the purpose of data collection and analysis, or just to gain some control over the growing set of digital devices permeating daily life.

1.2 The Three Pillars

Given the opportunity and the need, there are already a number of research efforts in the direction of machine programming in both industry and academia. The general goal of machine programming is to remove the burden of writing correct and efficient code from a human programmer and to instead place it on a machine. The goal of this paper is to provide a conceptual framework for us to reason about machine programming. We describe this framework in the context of three technical pillars: *(i)* intention, *(ii)* invention, and *(iii)* adaptation. Each of these pillars corresponds to a class of capabilities that we believe are necessary to transform the programming landscape.

Intention is the ability of the machine to understand the programmer's goals through more natural forms of interaction, which is critical to reducing the complexity of writing software. *Invention* is the ability of the machine to discover how to accomplish such goals; whether by devising new algorithms, or even by devising new abstractions from which such algorithms can be built. *Adaptation* is the ability to autonomously evolve software, whether to make it execute efficiently on new or existing platforms, or to fix errors and address vulnerabilities. As suggested in Figure 1, intention, invention, and adaptation intersect in interesting ways. When advances are made in one pillar, another pillar may be directly or indirectly influenced. In Section 5 we show that sometimes such influence can be negative. This further emphasizes the importance for the machine programming research community to be cognizant of these pillars moving forward and to understand how their research interacts with them.

The remainder of this report is organized as follows. In Sections 2, 3, 4, we provide a detailed examination of intention, invention, and adaptation, respectively. We also discuss the interactions between them, throughout. In Section 5, we provide a concrete analysis of verified lifting [36] and how it interacts with each of the three pillars (in some cases, disruptively). We close with a discussion on the impact of data, as it is the cornerstone for many ML-based advances.

2 Intention

Intention corresponds to the class of challenges involved in capturing the user's intent in a way that does not require substantial programming expertise. One of the major challenges in automating programming is capturing the user's intent; describing any complex functionality to the level of detail required by a machine quickly becomes just programming by another name. Table 1 provides a brief overview of existing research in the space of intention. It consists of three columns: *Research Area*, *System*, and *Influence*. The *Research Area* column includes subdomains of research for the given pillar. The *System* column includes a non-exhaustive list of examples of systems for that subdomain. The *Influence* column lists the different pillars, other than intention, that are influenced by the system listed in the corresponding *System* column. [1] This table structure is also used for the invention and adaptation pillar sections.

It is useful to contrast programming with human-human interactions, where we are often able to convey precise intent by relying on a large body of shared context. If we ask

[1] This table is not meant as an exhaustive survey of all the research in the intention pillar. Rather, it is meant as an example of work in subdomains of intention.

a human to perform a complex, nuanced task such as generating a list of researchers in the ML domain, we can expect the hidden details not provided in our original description to be implicitly understood (e.g., searching the internet for individuals both in academia and industry with projects, publications, etc., in the space of ML). By contrast, writing a computer program to do this would be a more significant undertaking because we would have to explicitly detail how to accomplish each step of the process. Libraries provide some assistance, but libraries themselves are difficult to write, can be difficult to use, and, in many cases, even difficult to find.

Table 1. Examples of Research in the Intention Pillar.

Research Area	System	Influence
Examples	Domain-Specific Input-Output ML [18]	–
	FlashFill [32]	Invention
Generalizability	Recursion [15]	Adaptation
Natural Language	Babble Labble [63]	–
	SQLNet [81]	–
	NL2P [39]	–
Partial Implementations	Sketch [72]	Adaptation
	AI Programmer [13]	Adaptation

For the last decade, the program synthesis community has struggled with this problem, both in the broad context of synthesizing programs from specifications, as well as in the narrower context of inductive programming or programming by example [20, 49]. There are at least two major observations that have emerged from prior work. The first observation is that by tightly controlling the set of primitives from which programs can be built and imposing strong biases on how these primitives should be connected, it is possible to cope with significant ambiguities in the specification. For example, the work on FlashFill [32] demonstrated that it is possible to synthesize a desired string manipulation from a small number of examples, often only one, by restricting the space of programs to a carefully crafted domain specific language (DSL) with carefully tuned biases. Moreover, subsequent work showed that such biases could be learned, rather than having to be tailored by hand [25, 69]. Similar observations have been made in other contexts, from the synthesis of SQL queries, to the synthesis of Java APIs [50, 83]. The Bayou project [50], for example, shows that it is possible to use deep neural networks to learn complex conditional distributions over programs, allowing a user to generate complex Java code from concise traces of evidence. Models not based on neural networks can also be used to learn distributions over programs for this purpose [14].

The second major observation is that multi-modal specification can help in unambiguously describing complex functionality. One of the earliest examples of multi-modal synthesis was the Storyboard Programming Tool (SPT), which allowed a user to specify complex data-structure manipulations through a combination of abstract diagrams, concrete examples, and code skeletons [70, 71]. The observation was that fully specifying a transformation via any of these modalities on its own, code, examples or abstract diagrams, was difficult, but in combination each of these formalisms could cover for the shortcomings of the other. A similar observation was made by the Transit project, which showed that it was possible to synthesize complex cache coherence protocols from a combination of temporal properties, concrete and symbolic examples [75].

Addressing the intention challenges more fully, however, will require additional breakthroughs at the intersection of machine learning and programming systems. One of the major opportunities is exploiting the ability modern learning techniques to extract meaning from high-dimensional unstructured inputs, such as images or speech. Recent work, for example on converting natural-language to programs, has shown the potential for exploiting this in the context of narrow domains [35, 39, 83]. Similarly, recent work on extracting programmatic representations from hand-drawn images has demonstrated the possibilities of using visual data as a basis for conveying intent [26]. Many of these systems, however, are one-off efforts targeted at narrow domains; one of the major questions is how to support this kind of functionality while maintaining the versatility of modern programming systems, and how to scale such high-level interactions to richer more complex tasks, including tasks that may require input from more than one person to fully describe.

3 Invention

Invention emphasizes the creation or refinement of algorithms or core hardware and software building blocks. For program construction, invention usually involves generating the series of steps that a machine would have to execute to fulfill a user's intent; in essence, it is the process of generating algorithms. This may require discovering new algorithms that are unique and different from prior contributions within the same space. In many instances, however, invention will be accomplished by identifying how to combine and adapt known data structures and algorithmic primitives to solve a particular problem. Both the program synthesis and the machine learning communities have made notable progress in this space in recent years, but there remain many open problems to be solved. See Table 2 for highlights of existing research in the space of invention. [2]

[2]This table is not meant as an exhaustive survey of all the research in the invention pillar. Rather, it is meant as an example of work in subdomains of invention.

Table 2. Examples of Research in the Invention Pillar.

Research Area	System	Influence
Explicit Search	λ^2 [28]	Intention
	SynQuid [57]	Intention
Constraint-Based	Sketch [72]	Intention
	PTS [74]	Intention
Symbolic Version Space	FlashFill [32]	Intention
Deductive	Paraglide [77]	Adaptation
	Fiat [22]	Adaptation
	Spiral [60]	Adaptation
Learning Directed	DeepCoder [9]	Intention
	Bayou [50]	Intention
Learning to Learn	Learning to Optimize [40]	Adaptation

3.1 Program Synthesis

For program synthesis, the modern approach has been to frame invention as a search problem where, given a space of candidate programs, the goal is to search for one that satisfies a set of constraints on the desired behavior [4]. This type of research has focused on questions of *(i)* how to represent the search space, *(ii)* how to explore it efficiently by exploiting knowledge of the semantics of the underlying building blocks, as well as *(iii)* understanding and advancing the structure of the semantic constraints themselves.

At a high-level, researchers have explored at least two major classes of approaches to this problem. The first class involves search techniques that explicitly try to build a syntactic representation of each program in the search space—abstract syntax trees (ASTs) are common as a representation. These techniques achieve efficiency by ruling out large sets of possible programs without exploring them one-by-one, usually by discovering that particular sub-structures can never be part of a correct solution [3, 28, 52, 57, 75]. The second class involves symbolic search techniques, where the entire program space is represented symbolically, either using a special purpose representation [32, 58], or, in the case of *constraint-based synthesis*, by reducing it to a set of constraints whose solution can be mapped to a concrete program, which can be solved using a SAT or SMT solver [33, 72, 74], or in some cases a numerical optimization procedure [17].

Many of these techniques, especially those designed to support rich specifications instead of simply input-output examples, have been enabled by the ability to automatically reason about the correctness of candidate programs, and in some cases, the ability to use static analysis to rule out large sets of candidate programs all at once [52, 57].

These techniques have made tremendous progress in recent years; for example, in the domain of bit-level manipulation, the most recent winner of the Syntax Guided Synthesis competition (SyGuS Comp) was able to automatically discover complex bit-level manipulation routines that were considered intractable only a few years earlier [5]. In the case of string manipulations, program synthesis is now robust enough to ship as part of commercial products (e.g. Flashfill in Excel [32]). In the context of data-structure manipulations, routines such as red-black tree insertion and complex manipulations of linked lists can now be synthesized and verified in the context of both imperative and functional languages [57], and in the functional programming realm, routines that were once considered functional pearls can now be synthesized from a few examples [28].

That said, there are fundamental limitations to the recent program synthesis approach to invention. Even with a restrictive set of primitives, the search-space grows exponentially with the size of the code-fragments that one aims to discover, making it difficult to scale beyond a dozen or so lines of code. There are some instances of systems that have been able to discover more complex algorithms, either by building them incrementally [54], or by breaking down the problem into smaller pieces—either by providing the synthesizer with the interfaces of sub-components to use [57], or by leveraging some domain-specific structure of the problem to decompose it into a large number of independent sub-problems [34].

Deductive synthesis techniques are another class of approaches to the Invention problem, where the idea is to start with a high-level specification and refine it to a low-level implementation by applying deductive rules or semantics-preserving transformations. This class of techniques has proven to be successful, for example, in automating the development of concurrent data-structures [77] or signal processing pipelines [60]. The growing power of interactive theorem provers such as Coq have also made it possible to get strong correctness guarantees from code developed through this approach [22]. The main drawback of this class of techniques is that while they tend not to suffer from the same scalability problems as the search-based techniques—because they break the problem into a number of small local reasoning steps—they tend to be domain specific, because they rely on carefully engineered deductive rules for the particular problem domain to operate effectively.

3.2 Machine Learning

Parallel to these efforts, the ML community has been exploring similar ideas in a different context. At one level, machine learning itself can be considered a form of invention. Many ML algorithms, including support vector machines (SVMs) and deep learning, can be seen as a form of constraint-based synthesis, where the space of programs is restricted to the set of parameters for a specific class of parameterized functions, and where numerical optimization is used to solve

the constraints, which in this case involve minimizing an error term. By focusing on a narrow class of parameterized functions, machine-learning techniques are able to support search spaces that are larger than what the aforementioned synthesis techniques can support. Neural networks with a million real-valued parameters are becoming standard for many applications, whereas the largest problems solved by the SMT-based techniques have on the order of a thousand boolean parameters. This allows neural-networks to capture significantly more complexity.

More recently, there have been significant efforts to capture more general program structure with neural networks, either by encoding differentiable Turing Machines [31], or by incorporating the notion of recursion directly into the neural network [15]. However, when it comes to synthesizing programs with significant control structure and that require more discrete reasoning, the techniques from the synthesis community tend to outperform the neural network based techniques [30].

3.3 New Directions

A major opportunity for breakthroughs in the invention problem lies at the intersection of the two lines of research. One important idea that is beginning to emerge is the use of learning-based techniques to learn distributions over the space of programs that are conditioned on features from the stated goals of the desired program. These distributions can be used to narrow the space of programs to something tractable. For example, DeepCoder [9] uses a neural network to map from the intention (given as a set of examples) to a restricted set of components that it has learned to recognize as useful when satisfying similar intentions. This allows it to then use an off-the-shelf synthesizer to solve the synthesis problem on this restricted program space. The Bayou project uses a more sophisticated network architecture to learn much more complex conditional distributions, allowing it to automatically determine, for example, how to use complex Java and Android APIs [50].

One of the open challenges in this space is to develop systems that can solve large-scale invention challenges, moving beyond simple algorithms and data-structure manipulations, by solving problems at the scale of an ACM programming competition or a collegiate programming course. This requires systems that can better mimic the way programmers approach these problems today. That is, using knowledge accumulated through practice and directed study to identify the core algorithmic building blocks needed to solve a problem. This also includes reasoning at a high-level of abstraction about how those building blocks fit together, and only then reasoning at the code level in a targeted fashion.

An important set of challenges in solving this problem is that while there are extensive resources to help humans learn to program, from tutorials to textbooks to stackoverflow.com, most of those resources are not suitable for data-hungry ML methods, such as deep learning. Applying ML in this domain may require a combination of new ML methods that can learn from data-sources aimed at humans, with novel solutions to exploit large-scale data sources, such as code repositories like GitHub, or synthetic data-sources such as randomly generated programs and datasets.

As architectures continue to evolve and become more complex and reconfigurable, some of the responsibility for coping with this complexity will fall on the invention layer, either because it will have to discover algorithms that map well to the constraints imposed by the hardware, or in the case of architectures that include FPGAs, the invention layer may need to derive the hardware abstractions themselves that align for a given algorithm. There is already some precedent in using constraint-based synthesis to handle non-standard architectures, ranging from exploiting vector instructions [11], to synthesizing for complex low-power architectures [56], but significantly more research is needed for this problem to be fully addressed.

4 Adaptation

Determining the algorithmic steps to solve a problem is only one part of the software development process. The resulting algorithms must be made to run efficiently on one or more target platforms, and after the code is deployed in the field, the code must be maintained as users expose bugs or corner cases where the system does not behave as expected. Moreover, as workloads on the system evolve, it may be necessary to re-evaluate optimization decisions to keep the software running at its peak performance. Together, these capabilities make up the *Adaptation* pillar. See Table 3 for highlights of existing research in the space of adaptation. [3]

4.1 Pre-deployment Optimization

In recent years, there have been significant efforts in automating the work to adapt an algorithm to perform optimally on a particular platform. To some extent, the entire field of compiler optimization is dedicated to this goal, but recently there has been a strong push to move beyond the traditional application of pre-defined optimization steps according to a deterministic schedule and to embrace learning-based techniques and search in order to explore the space of possible implementations to find a truly optimal one.

A turning point for the field came with the advent of auto-tuning, first popularized by the ATLAS [79] and FFTW [29] projects in the late 90s. The high-level idea of auto-tuning is to explicitly explore a space of possible implementation choices to discover the one that works most efficiently on a particular architecture. The PetaBricks language pushed this

[3] This table is not meant as an exhaustive survey of all the research in the adaptation pillar. Rather, it is meant as an example of work in subdomains of adaptation.

Table 3. Examples of Research in the Adaptation Pillar.

Research Area	System	Influence
Autotuning	OpenTuner [7]	–
	PetaBricks [6]	–
Code-to-Code	Verified Lifting [36]	Intention
	Tree-to-Tree Translation [19]	Intention
Correctness	ACT [2]	Intention
	CodePhage [66]	Intention
Data Structures	Learned Index Structures [37]	Invention
Mathematics	SPIRAL [60]	–
	Self-Adapting Linear Algebra Algorithms [23]	–

idea all the way to the language level, allowing the programmer to explicitly provide implementation choices throughout the code, replacing all compiler heuristics with reinforcement learning to discover close to optimal implementations [6].

Starting in the mid 2000s, there was also a realization that domain specific languages (DSLs) offered an important opportunity for automation. By eliminating a lot of the complexity of full-featured programming languages and exploiting domain specific representations, DSLs enabled aggressive symbolic manipulation of the computation, allowing the system to explore a much wider range of implementation strategies than what a human could possibly consider. Spiral [60] and the Tensor Contraction Engine (TCE) [12] were early examples of this approach. More recently, Halide has demonstrated the potential of this approach to bridge the "ninja-gap" by generating code that significantly outperforms expert-tuned codes with only a small amount of high-level guidance from the developer [61].

Despite this successes, there is significant scope for advances in this direction. For example, how do we enable transfer learning, so that the N-th program can be optimized faster by leveraging learned optimizations from the previous $(N-1)$ programs? Could a system learn new optimization strategies by analyzing a corpus of existing hand-optimized programs? Could learning help reduce the cost of developing high-performance DSLs, for example by reducing the need for custom heuristics or optimizations?

4.2 Post-deployment Maintenance

One of the most important maintenance tasks today is the repair of bugs and vulnerabilities. Fixing software bugs is currently an entirely manual process. A programmer must diagnose, isolate, and correct the bug. While the bug remains in place, it can impair program behavior or even open up security vulnerabilities. Recent research has demonstrated the feasibility of automating many aspects of repair. Two early systems include ClearView [55] and GenProg [78]. ClearView uses learned invariants that characterize correct execution to generate patches (which can be applied to a running program) that repair a range of execution integrity defects. GenProg uses genetic programming to search for patches for defects exposed by input/output pairs. More recently, Fan Long and Martin Rinard have pioneered machine learning mechanisms for automatically generating correct patches for large software systems [41, 66].

This recent research has highlighted the importance of learning and statistical techniques for program repair. At a fundamental level, program repair is an underdetermined problem, so a repair system must be able to select among all the possible patches that eliminate the symptoms of the bug to select the one that actually eliminates the bug without introducing other undesired behaviors. This can be done by automatically learning invariants that characterize correct behavior so that the generated patch can be required to maintain these invariants [55], by using machine learning over a large corpus of real bug fixes to build a model of successful patches [43], using change statistics from past human patches [38], or even leveraging a large corpus of bug fixes to learn how to generate successful patches [41].

Other successful program repair techniques focus on specific classes of defects such as incorrect conditionals [24, 80]. Here constraint solving can play an important role [16, 21, 42, 45, 51]. Automating repetitive source code edits can also eliminate or correct errors introduced by developer mistakes when working with similar code patterns [46, 47, 65, 73]. Code transfer, potentially augmented with machine learning to find appropriate code to transfer, can automatically work with code across multiple applications to eliminate defects and security vulnerabilities [10, 66, 68].

These demonstrated techniques lay out the initial case for the feasibility of automated bug detection and correction, autonomously without programmer involvement. Many existing successful techniques focus largely on surviving execution integrity bugs (bugs that can cause crashes or open up security vulnerabilities). Future directions include the additional incorporation of machine learning to enhance current latent bug detection techniques and to generate more sophisticated corrections for larger classes of bugs.

In addition to bug fixing, there are a number of other post-deployment maintenance tasks that could benefit from learning. In general these fall into the category of bit-rot prevention, and include, for example, upgrading to new versions of APIs and web-services, porting to new platforms, such as new cloud or mobile environments, or specializing code for particular uses. We envision the eventual development of systems that continuously monitor program execution, incorporate user feedback, and learn from large code repositories to deliver a system of autonomous and continuous program correction and improvement [66, 67].

5 The Interplay Between Pillars

Systems for machine programming will most likely be composed of a set of tools each of which focuses on a particular pillar. We anticipate that, in most cases, an individual tool cannot be fully understood in terms of a single pillar. The machine programming problem is multifaceted and issues concerning one pillar will inevitably impact the other pillars. Hence, we need to understand machine programming systems in terms of the interplay between the three pillars.

We expect this interplay to expose a tension between features of a tool that are supportive of the needs of any given pillar and those that are disruptive to the needs of the other pillars. The challenge in designing a machine programming system is to understand this interplay and reach an effective resolution of those tensions. As an example to explore this interplay, consider verified lifting [36].

Verified lifting tools input code written in one language, translate the code into a new language, and then formally verify that the new code produces results that are consistent with the original code. The prototypical example [36] takes stencil codes written in an imperative language, translates them into a modern DSL such as Halide, and then uses theorem proving technology to verify that the original and generated DSL codes are functionally the same. The newer code defines an abstract representation of the problem that can adapt onto a wide range of computer systems. Therefore, we see that the verified lifting problem is primarily used to support the *adaptation* pillar.

Verified lifting, however, goes well beyond the adaptation pillar. Consider the early steps in the verified lifting problem. A verified lifting tool must first understand the problem as represented in the input code. It discovers the intent of the program and produces an internal high-level representation of the problem often in mathematical or functional terms. This phase of the verified lifting process is firmly grounded in the *intention* pillar. From the high-level representation of the original problem, the verified lifting system can explore a range of algorithms appropriate to the target language; therefore working within the *invention* pillar. It then synthesizes the new code (the *adaption* pillar) and verifies that it is consistent with the high level representation of the problem. Hence, a verified lifting tool, while nominally focused on adaption, touches, in a supportive way, all three pillars.

It is important, however, to consider ways that a tool disrupts analysis within the different pillars. For example, when verified lifting translates low-level code into a compact representation in a DSL it is making the intent behind the code more apparent. Yet, the transformation can also interfere with other tools at the intention layer. For example, if the lifting transformation is not careful to preserve variable names, it may hamper the performance of intention layer tools that focus on names in the code to estimate whether a piece of code is relevant for a particular task. In general, when adaptation layer tools modify code, it is important to think about how the change may impact the ability of intention and invention layer tools to use that code.

While we have discussed the interplay between pillars in terms of just one machine based programming technique (verified lifting) we expect this complex interplay to be a common feature of machine programming systems. As researchers in machine program apply the three pillars in their own research, it is essential to consider the interplay between the three pillars and how this interplay is supportive or disruptive to the overall programming process.

6 Data

Nearly all machine programming systems require *data* to drive their algorithms. More specifically, every *Research Area* listed in Tables 1, 2, and 3 requires data (in some form) to function properly. The data required by these subdomains comes in a variety of forms (e.g., code, input/output examples, DSLs, etc.), but is ever-present. This dependency on data makes it essential that we consider the open problems and emerging uses around data when reasoning about machine programming and the systems that implement it.

The various approaches to address the three machine programming pillars have different needs in terms of the type and size of data they require. Moreover, there is a wide spectrum in terms of the quality of the data that a project might use. We discuss some of these emerging data uses and issues for the remainder of this section.

Code Repositories Large version control repositories, such as GitHub, offer the promise of access to full revision histories for all the code necessary to build and run a project, as well as its accompanying documentation. The code available in these public repositories has grown exponentially over the last several years and show no indication of stopping. Many projects in these repositories have long commit histories with detailed commit logs which could be of notable value to machine programming systems [76]. However, recent analysis of public repositories has shown that a large fraction of the projects are duplicates, making a significant portion of the data less useful [44].

One use of code repositories is to use their version control histories to identify code changes that correspond to the introduction of performance or correctness bugs. This type of data utilization has been explored to train models for program repair [1, 43]. Additionally, the presence of complete codebases makes it possible to run whole program analyses on the code. In some contexts, it has been shown that augmenting the code with features discovered from program analysis can help train more effective models [62]. However, complete codebases may not always be available, and running whole program analysis may not always be feasible.

Incomplete and Synthetic Code + Natural Language Sources such as stackoverflow provide a wealth of information beyond code, which can be used to correlate code and natural language. There has been some work in the community in extracting information from code that comes in the form of code snippets like those usually found in stackoverflow. This requires assembling information gathered from multiple different snippets into a coherent model of the behavior of a code component [48, 53]. There have even been some efforts aimed at extracting code from video tutorials, which offers the possibility of correlating the code with the accompanying narration [82]. In some contexts, the data needed to train a model does not even have to come from real code; synthetic data generated from random combinations of components can be useful, as demonstrated by DeepCoder [9].

Data Privacy One of the open issues of machine programming data is that of privacy. In the context of code, machine programming systems will eventually have to work with and protect intellectual property as well as software licensing agreements. As we move toward a future where data will be more openly shared, used, and traded, new models and tools for secure and privacy-preserving exchange will become increasingly important [59, 84]. In the case of models learned from code, there are important open cyclic questions surrounding the copyright status of code generated from models trained from copyrighted code.

Lifecycle Management Machine programming systems will require lifecycle management practices, similar in scope to those used in traditional software engineering. Much of this is due to the need to fulfill the goals of the adaptation pillar. These lifecycle management efforts will be long-lasting and will require support for continued monitoring and improvement around changing software needs and advances in an increasingly complex and heterogeneous hardware ecosystem. A significant portion of this lifecycle management will be centered on managing the data that are required for such an adaptive machine programming system, as these data will help ensure the stability and maturity of the system. As machine programming systems evolve so will the data they ingest to baseline and advance the system. Because of this, proper data management is likely to be a key enabler to calibrate any machine programming system's lifecycle.

7 Conclusion

In the post Dennard scaling world, where performance comes from architectural innovation rather than increased transistor count with constant power density, hardware complexity will only increase. Heterogeneous computing will become more widely used and more diverse than it is today. Over the next several years, specialized accelerators will play an increasingly important role in the hardware platforms we depend on. At the same time, the nature of programming is changing. Instead of computer scientists trained in the low level details of how to map algorithms onto hardware, programmers are more likely to come from a broad range of academic and business backgrounds. Moreover, rather than programming in low level languages that interface almost directly to hardware, programmers are more likely to use higher level abstractions and scripting languages. This will fundamentally change how we write software. We believe this change is already well underway.

We envision a future where computers will participate directly in the creation of software, which we call *machine programming*. This paper presents a framework to organize work on this problem. We call this framework *the three pillars of machine programming*. The three pillars are intention, invention, and adaptation.

Intention focuses on the interface between the human and the machine programming system; i.e., techniques to discover what a program needs to do from input that is natural to express by the human. A system grounded in the intention pillar meets human programmers on their terms rather than forcing them to express code in computer/hardware notations. *Invention* emphasizes machine systems that create and refine algorithms or the core hardware and software building blocks from which systems are built. *Adaptation* focuses on ML-based tools that help software adapt to changing conditions; whether they are bugs or vulnerabilities found in an application or new hardware systems.

Data is at the foundation of the modern renaissance in artificial intelligence (AI). Without vast amounts of data, it is unlikely that AI would have had significant impact outside specialized academic circles. In this paper, we explored the impact of data as it pertains to machine programming. Software repositories in systems such as GitHub and the vast amount of software embedded in countless webpages is the raw material that will likely support a large majority of the emergence of machine programming.

Finally, there are numerous open problems that must be solved to make machine programming a practical reality. We outlined some of these open problems in this paper. It will take a large community of researchers years of hard work to solve this problem. If we can agree on a conceptual framework to organize this research, it will help us advance the field and more quickly bring us to a world where everyone programs computers; on human-terms with machine systems handling the low level details of finding the right algorithm for the right hardware to solve the right problem.

References

[1] Mohammad Mejbah Ul Alam, Justin Gottschlich, and Abdullah Muzahid. 2017. *AutoPerf: A Generalized Zero-Positive Learning System to Detect Software Performance Anomalies.* Technical Report. http://arxiv.org/abs/1709.07536/

[2] Mohammad Mejbah ul Alam and Abdullah Muzahid. 2016. Production-run Software Failure Diagnosis via Adaptive Communication Tracking. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 354–366. https://doi.org/10.1109/ISCA.2016.39

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[4] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

[5] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

[6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1542476.1542481

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN Automatic Coding System. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability (IRE-AIEE-ACM '57 (Western))*. ACM, New York, NY, USA, 188–198. https://doi.org/10.1145/1455567.1455599

[9] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. *ICLR* (2017). https://arxiv.org/abs/1611.01989

[10] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 257–269.

[11] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. 123–134. https://doi.org/10.1145/2442516.2442529

[12] Gerald Baumgartner, David E. Bernholdt, Daniel Cociorva, Robert J. Harrison, So Hirata, Chi-Chung Lam, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. 2002. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*. 33:1–33:10. https://doi.org/10.1109/SC.2002.10056

[13] Kory Becker and Justin Gottschlich. 2017. AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. *CoRR* abs/1709.05703 (2017). arXiv:1709.05703 http://arxiv.org/abs/1709.05703

[14] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2933–2942. http://jmlr.org/proceedings/papers/v48/bielik16.html

[15] Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making Neural Programming Architectures Generalize via Recursion. *CoRR* abs/1704.06611 (2017). arXiv:1704.06611 http://arxiv.org/abs/1704.06611

[16] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11')*.

[17] Swarat Chaudhuri and Armando Solar-Lezama. 2010. Smooth interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 279–291. https://doi.org/10.1145/1806596.1806629

[18] Xinyun Chen, Chang Liu, and Dawn Song. 2017. Towards Synthesizing Complex Programs from Input-Output Examples. *CoRR* abs/1706.01284 (2017). arXiv:1706.01284 http://arxiv.org/abs/1706.01284

[19] Xinyun Chen, Chang Liu, and Dawn Song. 2017. Tree-to-tree Neural Networks for Program Translation. *CoRR* abs/1712.01208 (2017). https://arxiv.org/abs/1712.01208

[20] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA.

[21] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer-Aided Verification (CAV)*.

[22] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 689–700. https://doi.org/10.1145/2676726.2677006

[23] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. 2005. Self-Adapting Linear Algebra Algorithms and Software. *Proc. IEEE* 93, 2 (Feb 2005), 293–312. https://doi.org/10.1109/JPROC.2004.840848

[24] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. *CoRR* abs/1505.07002 (2015). http://arxiv.org/abs/1505.07002

[25] Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. 1638–1645. https://doi.org/10.24963/ijcai.2017/227

[26] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2017. Learning to Infer Graphics Programs from Hand-Drawn Images. *CoRR* abs/1707.09627 (2017). arXiv:1707.09627 http://arxiv.org/abs/1707.09627

[27] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[28] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

[29] Matteo Frigo and Steven G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*. 1381–1384. https://doi.org/10.1109/ICASSP.1998.681704

[30] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). arXiv:1608.04428 http://arxiv.org/abs/1608.04428

[31] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *CoRR* abs/1410.5401 (2014). arXiv:1410.5401 http://arxiv.org/abs/1410.5401

[32] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[33] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 62–73. https://doi.org/10.1145/1993498.1993506

[34] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. 2016. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 302–320. https://doi.org/10.1007/978-3-319-40970-2_19

[35] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. 963–973. https://doi.org/10.18653/v1/P17-1089

[36] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

[37] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *CoRR* abs/1712.01208 (2017). https://arxiv.org/abs/1712.01208

[38] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*. 213–224.

[39] Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. 2013. From Natural Language Specifications to Program Input Parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*. 1294–1303.

[40] Ke Li Li and Jitendra Malik. 2017. Learning to Optimize. *ICLR* (2017). https://arxiv.org/abs/1606.01885

[41] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 727–739.

[42] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*.

[43] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 298–312.

[44] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *PACMPL* 1, OOPSLA (2017), 84:1–84:28. https://doi.org/10.1145/3133908

[45] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 691–701.

[46] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11')*. 329–342.

[47] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 502–511.

[48] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 997–1016. https://doi.org/10.1145/2384616.2384689

[49] Stephen Muggleton. 1991. *Inductive logic programming*. Vol. 8. 295–318 pages. https://doi.org/10.1007/BF03037089

[50] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR* abs/1703.05698 (2017). arXiv:1703.05698 http://arxiv.org/abs/1703.05698

[51] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13')*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

[52] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 619–630. https://doi.org/10.1145/2737924.2738007

[53] Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. 2013. Symbolic Automata for Static Specification Mining. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 63–83. https://doi.org/10.1007/978-3-642-38856-9_6

[54] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 408–418. https://doi.org/10.1145/2594291.2594297

[55] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. 87–102.

[56] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodík. 2014. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 396–407. https://doi.org/10.1145/2594291.2594339

[57] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 522–538. https://doi.org/10.1145/2908080.2908093

[58] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 107–126. https://doi.org/10.1145/2814270.2814310

[59] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 85–100. https://doi.org/10.1145/2043556.2043566

[60] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. 2004. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *IJHPCA* 18, 1 (2004), 21–45. https://doi.org/10.1177/1094342004041291

[61] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 519–530. https://doi.org/10.1145/2462156.2462176

[62] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[63] Chris Re. 2017. *Babble Labble*. Technical Report. Department of Computer Science, Stanford, Stanford, CA. https://hazyresearch.github.io/snorkel/blog/babble_labble.html

[64] Charles Rich and Richard C. Waters. 1988. The Programmer's Apprentice: A Research Overview. *Computer* 21, 11 (Nov. 1988), 10–25. https://doi.org/10.1109/2.86782

[65] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*.

[66] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 43–54.

[67] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin C. Rinard. 2015. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 473–486.

[68] Stelios Sidiroglou-Douskos, Eric Lantinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

[69] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 398–414. https://doi.org/10.1007/978-3-319-21690-4_23

[70] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 289–299. https://doi.org/10.1145/2025113.2025153

[71] Rishabh Singh and Armando Solar-Lezama. 2012. SPT: Storyboard Programming Tool. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 738–743. https://doi.org/10.1007/978-3-642-31424-7_58

[72] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

[73] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS*.

[74] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 313–326. https://doi.org/10.1145/1706299.1706337

[75] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 287–296. https://doi.org/10.1145/2462156.2462174

[76] Martin Vechev and Eran Yahav. 2016. Programming with "Big Code". *Found. Trends Program. Lang.* 3, 4 (Dec. 2016), 231–284. https://doi.org/10.1561/2500000028

[77] Martin T. Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 125–135. https://doi.org/10.1145/1375581.1375598

[78] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[79] R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*. 38. https://doi.org/10.1109/SC.1998.10004

[80] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*.

[81] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. *CoRR* abs/1711.04436 (2017). arXiv:1711.04436 http://arxiv.org/abs/1711.04436

[82] Shir Yadid and Eran Yahav. 2016. Extracting Code from Programming Tutorial Videos. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 98–111. https://doi.org/10.1145/2986012.2986021

[83] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26. https://doi.org/10.1145/3133887

[84] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 283–298. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng